Self-Organized Specialization and Controlled Emergence in Organic Computing Systems

Von der Fakultät für Mathematik und Informatik
der Universität Leipzig
angenommene

DISSERTATION

zur Erlangung des akademischen Grades

DOCTOR RERUM NATURALIUM
(Dr. rer. nat.)

im Fachgebiet
Informatik

vorgelegt

von Dipl.-Inf. Alexander Scheidler
geboren am 20. Dezember 1977 in Eisenach

Die Annahme der Dissertation wurde empfohlen von:

Prof. Dr. Martin Middendorf, Universität Leipzig
Prof. Dr. Hartmut Schmeck, Universität Karlsruhe

Die Verleihung des akademischen Grades erfolgt mit Bestehen
der Verteidigung am 11. Februar 2010 mit dem Gesamtprädikat magna cum laude.

Danksagung

Mein Dank gilt in erster Linie Prof. Martin Middendorf für die Ermutigung zu dieser Arbeit und das mir über deren gesamte Dauer entgegengebrachte Vertrauen. Durch seine stetige Anregung und kompetente Anleitung hat er einen großen Teil zum Gelingen dieser Arbeit beigetragen.
Besonders danken möchte ich auch Konrad Diwold, Matthias Bernd und Thomas Hühn für das gründliche Korrekturlesen der Arbeit und ihre hilfreichen Kommentare. Weiter bedanke ich mich bei meinen Kollegen Stephanie Keller-Schmidt, Stefan Janson, Daniel Merkle, Sebastian Lange und Kai Ramsch für den produktiven Austausch und die gute Arbeitsatmosphäre.
Meiner gesamten Familie, insbesondere meinen Eltern Volkmar und Margitta Scheidler, danke ich für ihre andauernde Unterstützung und Bestärkung. Ganz besonders bedanke ich mich bei Christin Albin, die mir immer verständnisvoll und aufmunternd zur Seite stand.

Contents

1 Introduction

In everyday life we are surrounded by a multitude of computing devices, including notebooks, PDAs, mobile phones, navigation systems, MP3-players and so on. As they become increasingly powerful, cheaper and smaller, our environment will be filled with large collections of these devices. Most of them are equipped with wireless communication interfaces, allowing them to interact and collaborate. The formed networks of autonomous intelligent systems open fascinating application areas, but also result in increasingly complex and potentially unreliable systems. This opens new challenges to designers and users. Due to the increasing complexity it will not be possible to explicitly design and manage these systems in every detail and to anticipate every possible configuration. Thus, our technical systems will have to act more independently, flexibly, and autonomously. They should organize themselves and adapt to an uncertain and dynamic environment while maintaining a requested functionality.

Flexibility, self-organization, autonomy, and adaptivity are features ubiquitous in natural systems. Organic Computing is a new field of computer science that has the vision to make technical systems more life-like in order to address the challenging requirements raised by an increasing complexity. Organic Computing systems shall show so called self-x properties, i.e., they should be self-organizing, self-optimizing, self-healing, self-protecting, self-configuring, self-explaining and so on. Like in natural systems these properties shall become apparent on the level of the whole system through the properties and interactions of their components.

The self-x feature which is considered to be most important for Organic Computing systems is self-organization, i.e., the adaptive and dynamic process that allows systems to establish and maintain structure and function without external control. Self-organizing systems bear several advantages over classical, centrally controlled systems. For instance, failures of single components usually do not cause global malfunctions and self-organizing systems typically scale well with their size. Therefore one objective of Organic Computing is the identification and technical usage of the principles underlying the wide variation of different self-organizing natural systems (Rochner and Müller-Schloer, 2005).

Closely related to self-organization is the concept of emergence (emergere, lat.: to appear, being produced, come into existence). Emergence relates to the occurrence of novel properties on a higher system level based on the interaction of the lower level parts of the system. Organic Computing tries to deepen the understanding of emergent behavior in self-organizing systems and to utilize this knowledge for technical applications (Branke et al., 2006).

Self-organization and resulting emergent effects also give rise to new problems that are unknown in classical technical systems. Organic Computing must be able to deal with the possibility of emerging global behavior due to unanticipated interactions between the active components and find ways to control them (Müller-Schloer, 2004).

This thesis covers all of these aspects: the application of self-organization principles found in nature to technical systems, the utilization of emergence for solving problems in a decentralized and robust way, and the introduction and investigation of methods to control emergent behavior.

1.1 Contribution of this Work

To emphasise the value of this thesis, this section enumerates our main contributions together with the papers in which they are published. First this thesis briefly sketches the concepts of self-organization and emergence, and gives a slightly deeper introduction to the problem of engineering self-organizing systems with emergent properties. In the second and third chapter we deal with emergence. We give examples of how to utilize emergent effects and investigate how emergent behavior can be controlled. In the fourth and fifth chapter we discuss how to create Organic Computing Systems whose components are able to specialize for certain tasks and to cooperate in a self-organized way.

Chapter 2 investigates two decentralized systems that exploit emergent effects. In the first part of the chapter so called Emergent Sorting Networks are introduced. These directed networks consist of router agents with fixed positions and buffer sites that the agents can use to store objects of different types. New objects are inserted randomly into the network and are moved by the agents using simple local rules. At the outflow of the networks the objects appear sorted, i.e., batches of objects of the same type can be observed. As a macro-level property of the system the sortedness emerges from the local rules of the agents. We study different local routing rules on varying network topologies in terms of sorting performance and fairness. The presented results are published in Scheidler et al. (2008).

In the second part of the chapter a decentralized algorithm for packet clustering in networks is presented. This algorithm, which is inspired by the chemical recognition system in ants, runs in networks in a self-organized way without any central control. Packets that need to be grouped (clustered) according to an inherent data vector “meet” in the routers of the networks and exchange information. From these local, decentralized interactions of the packets the clustering emerges as a global system property. The algorithm was published in Merkle et al. (2005, 2004) and Merkle et al. (2005).

Chapter 3 investigates an important question when dealing with self-organizing systems, that is, the question how to control (unwanted) emergent effects. A new approach called swarm controlled emergence is introduced as a way to deal with this problem. A proof of concept is given by applying the method to control an emergent effect in a nature inspired test system. This work is submitted to an international journal and has been partially published in Merkle et al. (2007).

A second work presented in this chapter deals with the question of controlling emergent congestion effects in groups of ant like moving agents. The presented control methods do not (or only slightly) alter the internal functioning of the agents, instead the environment is modified in order to avoid the congestion. This work is published in Scheidler et al. (2008).

Chapter 4 introduces so called Organic Support Systems as an approach to deal with the execution of the necessary support and system care tasks in Organic Computing systems. Organic Support Systems consist of autonomous components which exhibit reconfigurable hardware in order to be able to adapt to the actual needs of the supported systems by specializing for the required types of support tasks. Several aspects are treated in the chapter. First a self-organized, social insect inspired mechanism for the allocation of the service tasks to the support components is introduced, a work that is published in Merkle et al. (2008, 2006). Second, the stability and performance of ant queue inspired methods for the partitioning and the sequential execution of the support tasks is studied. The content of this part of the chapter is published in Scheidler et al. (2008c, 2007). Finally, as a third aspect, we investigate how to allocate the support tasks if the supported and the supporting components are interconnected via a network. It is studied how the decentralized clustering algorithm presented in Chapter 2 can be used to solve this problem. This work is submitted to an international journal and is partially published in Merkle et al. (2006).

Chapter 5 presents a work that uses interacting Pittsburgh-style Learning Classifier Systems to evolve rule sets for solving classification problems on computing systems consisting of distributed, autonomous, memory constrained components. Using this approach the components become specialists for parts of the classification problem and learn to solve the whole problem in cooperation. The chapter takes a deeper look at the structure and properties of the evolved rule sets and the way the components share their knowledge. The influence of different communication topologies and communication costs on the emerging patterns of cooperation, on the obtained classification performance of the whole system and on the distribution of knowledge within the system is studied. The work presented in this chapter is published in Scheidler and Middendorf (2009) and Scheidler and Middendorf (2009).

1.2 Self-Organization and Emergence

Self-organization and emergence are highly debated entities in many research fields such as philosophy (Bedau and Humphreys, 2008), physics (Licata and Sakaji, 2008), biology (Johnson, 2001), sociology (Luhmann, 1997), and several other fields of science (Corning, 2002). For example physicists used the concepts of self-organization and emergence to explain Bénard convection cells, psychologists to explain consciousness, economists to explain stock market behavior, and organization theorists to explain informal networks in large companies. The collective movement behavior of animal groups, such as swarms of bees, schools of fish, flocks of birds, and herds of mammals can be explained as emerging from the local movement rules of the individuals. A traffic jam is the emergent result of the interactions between drivers that make decisions based on their local observations of the actual traffic situation. The evolution itself can be seen as a self-organizing, emergent phenomenon. While highlighting the importance of the concepts, their broad use has the disadvantage that depending on the domain the concepts are associated with different and sometimes opposite ideas and interpretations.

We are aware that the following characterizations of the terms are too coarse to account for some intricacies of the many different views which can be found in the literature. Especially we will not try to give exact definitions, the interested reader is referred to Degueta et al. (2006) for a review on some definitions of emergence, respectively to Anderson (2002) for definitions on self-organization. However the following descriptions will be sufficient in the context of this thesis and relates to some common views especially from the computer science literature (see, e.g., Banzhaf, 2009; Cakar et al., 2007; Correia, 2006; De Wolf and Holvoet, 2004; Fromm, 2005a,b; Marzo et al., 2006; Mühl et al., 2007; Müller-Schloer and Sick, 2006; Rochner and Müller-Schloer, 2005; Yamins, 2005).

This thesis is dealing with systems consisting of a large number of interacting agents or components that have no central control and hence are based only on local rules and interactions. We will call such a system self-organising if it autonomously acquires and maintains its structure in order to display a coherent behavior. That is, in response to external circumstances and under appropriate conditions the system has the ability to spontaneously arrange its components in a purposeful (non-random) manner. Such a system adapts to the environment in order to be able to provide its primary functionality. Self-organization increases the order in the system structure, i.e., the order within the organization and interactions of its components. Self-organizing systems are autonomous, that is, the increase in order is reached without any external control.

When a large number of entities interact, the resulting system can show features and behaviors which are not possessed by the individual constituents. That is, novel properties can emerge on a higher system level. For the concept of emergence the most important property is the micro-macro effect. It refers to the fact that the emergent effects (behavior, structures, properties or patterns) that can be observed on a higher level of the system (also called macro-level) are caused by the interactions of individual entities at the lower level of the system (also called micro-level).

For the relation between the concepts of self-organization and emergence different points of view can be found in the field of computer science: emergence and self-organization are completely different concepts (De Wolf and Holvoet, 2004), both concepts are synonyms (Fromm, 2006), emergence is a main property of self-organization (Holzer et al., 2008), system showing emergence must be self-organizing (Minati and Pessa, 2006). Since in most cases that are interesting for Organic Computing emergence and self-organization come together, we will stick with the view of Mnif and Müller-Schloer (2006), that is, emergence is self-organized order. In this view a self-organization process increases the order of a system and this increased order establishes as an emergent effect that is observable on a higher system level.

Two points are important regarding this concept of emergence. First, the role of the observer, since the concept of order depends strongly on the (human) observer's selected system attributes (Mnif and Müller-Schloer, 2006). Second, emergent effects are very hard to predict, because they are novel, not possessed by the parts of the system and are constituted through self-organized interactions between the parts of the whole system. Some authors even define emergent effects as effects for which the optimal way of prediction is simulation (Bedau, 1997; Darley, 1994).

1.3 Engineering Self-Organizing Systems with Emergent Properties

Self-organizing technical systems have many attractive features. First of all they are robust. The emergent properties on the macro-level are insensitive to fluctuations on the micro-level processes and independent of individual components. Because self-organizing systems are continuously re-organizing, their performance is robust against the loss of single components, i.e., the systems show a graceful degradation behavior. Losses of components can be tolerated because each component is simple and probably inexpensive. Simple entities also have the advantage that such components are easy to program and to prove correct at the level of individual behavior. They can be implemented as (or on) simple devices like sensors or RFID-chips. Another important feature of self-organizing systems is their scalability. In fact, often the performance of such systems improves when increasing the number of individual components. Last but not least these systems are adaptive, i.e., they can deal with changing environment by re-organizing themselves. All these properties make self-organizing systems and the resulting emergent properties very interesting for system designers. An increasing number of researcher concern with the question how to engineer self-organizing systems and the resulting emergent behavior. They deal with the problem to find appropriate micro-level specifications that lead to a desired global emergent behavior.

Traditional methodologies to design and engineer systems usually follow a top-down approach. The design process is organized strictly hierarchical. A sequence of modelling steps starting with a high level specification leads through several refinements finally to a model which directly specifies the entities and their interactions. But when it comes to design self-organizing systems and especially emergent behavior, this top-down process conflicts with the fact that emergence is a bottom-up phenomenon. The unpredictable bottom-up micro-macro direction in self-organizing systems makes any pure top-down attempt useless.

These days there exists no common formal methodology for engineering self-organizing systems with emergent properties, although some first steps are done in this direction. Many of the proposed methodologies which shall guide developers through the process of engineering emergent solutions are from the field of Multi Agent Systems and Agent Oriented Software Engineering (see, e.g., Bernon et al., 2002; Gleizes et al., 2007; Sudeikat et al., 2009; Wolf and Holvoet, 2005). Like stated before, a pure top-down approach can not solve the problem and a formal design methodology must contain some kind of round-trip process based on stepwise iterative enhancements that can bridge the micro-macro distinction (Edmonds and Bryson, 2004; Fromm, 2006).

Within such a process the main question to be answered is how to find appropriate micro-level rules that lead to a desired behavior on the macro-level. One way to derive these micro-level specification is to imitate and adapt existing solutions. In the last years many natural and artificial systems showing emergent effects have been investigated, mostly using (individual based) simulations. For instance, simple computational models can reproduce the mentioned collective moving behavior of animal groups in a realistic way (Giardina, 2008). In these models each individual has a position, a current direction, and a current speed. While moving every individual attempts to maintain a minimum distance representing the personal space of the individual between itself and others, i.e., the individual avoids collisions. If there is no individual within the personal space of an individual it steers to the average position of local neighbours and towards the average heading of them, i.e., it gets attracted by its local neighbors and tries to align with them. These simple interactions between the individuals are sufficient to reproduce the complex adaptive patterns observable at the level of the group.

Cellular Automata have been used to model several emergent phenomenas. Cellular Automatas are spatially and temporally discrete dynamical systems composed of a lattice of extremely simple elements ("cells"). Each cell's state is fully determined by the states of its neighboring cells, and updated repeatedly using simple local rules. Despite the relative simplicity of the model it can produce complex spatial and temporal patterns (Symons, 2008). For instance the model can be used to reproduce the mentioned emergence of traffic jams (Nagel and Schreckenberg, 1992) or it can be used to model the emergent formation of patterns in biological systems, like the pigmentation patterns of some seashells (Deutsch and Dormann, 2005).

Even though, such extensive computational studies of emergent behavior will fail to shed light on general metaphysical questions concerning the nature of emergence (Symons, 2008), they nonetheless can provide plausible explanations of particular cases of self-organization and the resulting emergent behavior. The gathered experience can thus be cumulated and a future engineer might use a large collection of basic "design patterns for emergent effects" to apply and, if necessary, adapt them to build a desired system behavior. First steps to systematically collect such design patterns and guidelines for designing self-organizing systems are already done (see, e.g., De Wolf and Holvoet, 2006; Gardelli et al., 2007; Mamei et al., 2006; Parunak and Brueckner, 2004; Sudeikat and Renz, 2008).

Most of these patterns are inspired by natural systems. A very prominent example of a nature phenomenon that served as a "design pattern" for several technical applications is the pheromone trail laying behavior of ants. Through the local interactions of the individuals with their environment ant colonies can form and maintain relatively short trails between their nests and food sources. In the following a brief explanation how this works is given. Individual ants deposit a chemical substance called pheromone on the ground when they move from a food source to their nest. Other ants follow the pheromone trail and reinforce it if they eventually find food. Over time the pheromone evaporates, thus reducing its attractive strength. In an experimental setup, offering two possible paths to the food that have different length, ants which followed the shorter path will return faster and thus the pheromone on the shorter path will be stronger and attract more ants that start new trips to the food. Eventually all ants will use the short path. This behavior has inspired the well known Ant Colony Optimization metaheuristic that is used to solve combinatorial optimization problems (see, e.g., Dorigo et al., 1996) but also, for example, network traffic routing protocols (Di Caro, 2004) or the distributed control of robot swarms (Hauert et al., 2008).

Another emergent effect found in insects that was applied to several technical systems is a synchronization behavior found in fireflies. In South-East Asia huge swarms of tropical fireflies synchronously emit light flashes to attract mating partners (Buck, 1988). The ability of the insects to synchronize their flashing can be modelled using so called pulse-coupled oscillators. Pulse-coupled oscillators influence each others only during short, periodic pulses. Every oscillator exhibits an activation state that increases over time. If the activation level reaches a certain threshold the oscillator fires and the activation is set back to zero. Neighbored oscillators that observe the firing increase their activation by a small amount, which can lead to firing and a setback of the activation level, too. In this way almost always a state emerges in which all oscillators are firing synchronously (Mirollo and Strogatz, 1990). Inspired by this model, several distributed clock synchronization algorithms for sensor, overlay, and ad-hoc networks have been developed (see, e.g., Leidenfrost and Elmenreich, 2009; Tyrrell et al., 2007). Whereas most applications can be found in the field of network synchronization, there are also examples of using the firefly synchronization method in other fields, for instance, Christensen et al. (2009) proposes a completely distributed algorithm to detect non-operational individuals in multi-robot systems.

Many other applications of self-organizing systems discovered in nature can be found, for example, the emergent pattern formation in reaction diffusion systems has been applied to sensor networks (see, e.g., Wakamiya et al., 2008), the behavior of ants to sort their larvae has inspired the design of clustering algorithms (see, e.g., Handl and Meyer, 2007), models of task allocation in wasp colonies were applied in different industrial settings (see, e.g., Cicirello and Smith, 2004), the movement rules found in herds, flocks, and schools are used to simulate crowds in movies or computer games (see, e.g., Azahar et al., 2008), models of the spread of epidemics have led to several robust and highly resilient algorithms for propagating information in networks (Eugster et al., 2004), cell adhesion processes have inspired strategies for topology forming in Peer-to-Peer networks (Jelasity et al., 2009), and so on.

Nature inspired "emergence design patterns" can thus be very helpful for designing system with emergent properties. But often for a given problem no such pattern exists or an existing one has to be modified. Because of the inherent unpredictability of the emergent effects, the only reliable way to find an appropriate micro-level specification is then to follow, at least partly, a trial-and-error process. This means specific local rules are implemented via simulation or directly in an existing system and it has to be tested if the wanted macro-effects emerge. If not, the rules must be modified in a reasonable way until the desired goal is reached. Since a pure bottom up process that tests all possible combinations and configurations on the micro-level is not feasible, more elaborate search methods must be applied.

The search for simple rules that generate a complex pattern resembles to the problem of science in general (Fromm, 2006). Science tries to explain complexity by describing complex natural phenomena using a minimum of primary principles, laws, and rules. The scientific method uses experiments to gather evidence and numerical data in order to validate hypotheses and theories. A continuous round-trip from the concrete world phenomenas to the abstract model and back results in an iterative refinement of the theory or model. The formulation of the hypotheses, principles, theories, and laws is the non-trivial step in science. This hard step requires a lot of personal experience, creativity, intuition, and curiosity.

Several approaches to engineer self-organizing systems somehow equal this well-known scientific method applied to an artificial world instead to the natural world (e.g. Edmonds, 2004; Gershenson, 2007). Interchanging top-down and bottom-up phases shall lead to a stepwise iterative enhancement of a considered solution. The way up corresponds to experiments in science, this means synthesis and simulation of the individual actions on the micro level. The way down corresponds to theory and means to create testable hypotheses and to analyse how collective forces influence and constrain individual actions. A human designer is directly incorporated in the trial-and-error based iterative enhancement of the micro-level rules. Like in the scientific method experience and intuition are helpful.

In cases of systems with high complexity and large parameter spaces involving a human designer in the time-consuming trial-and-error process is often not efficient or even unfeasible. In this case automated processes that search for local rules based on a desired global behavior or a given global goal can help.

Since the complexity of the considered systems grows heavily with the number of possible local states and rules, a full search over the design space is not feasible. A more suitable methodology for the automated (simulation based) search and design of emergent behavior is to use Evolutionary Algorithms (EAs) (Branke and Schmeck, 2008; Zapf and Weise, 2007). This mimics the way the observable emergent behavior found in many natural systems has developed. Because EAs are black box algorithms they can be applied to any problem where a quality (or fitness) can be assigned to a solution and thus the algorithm does not need any internal knowledge about the simulated system. Other advantages are that EAs can run in parallel and cope with the uncertainty of stochastic simulation models. Since they maintain populations of solutions EAs are able to handle multiple objectives and uncertainty about the user preferences.

Several examples for the use of Evolutionary Algorithms to evolve local rules for self-organizing systems exist. In the field of evolutionary robotics they have been applied to generate self-organising behaviors in groups of autonomous robots (see, e.g., Fehervari and Elmenreich, 2009; Trianni, 2008). In the field of cellular automata they are used, for instance, to search for glider guns (Sapin and Bull, 2007) or to evolve automata that can detect edges in images (Batouche et al., 2006). In Komann et al. (2009) and Komann and Fey (2009) EAs evolve appropriate micro-level rules to deal with the so called Creatures' Exploration Problem. In this problem agents are situated in a regular grid with blocked and unblocked cells. Based on the locally perceived neighborhood final state machines determine the next movement steps of the agents. A Genetic Algorithm is used in order to evolve final state machines which steer the agents to visit as many unblocked cells as possible in a given time.

In these examples the automatic trial-and-error search for appropriate micro-level rules takes place in an offline, simulated environment. In this way a large number of different combinations of rules can be tested with respect to a given goal. The downside of a simulation based approach is that it is sometimes hard or even impossible to model all influences a changing environment might have on the developed self-organized system. Such influences can lead to unforeseen and therefore not simulated systems states. A way to deal with this problem is to shift the search for the local behavior leading to a desired global (emergent) behavior from the design time to the run time of the system.

A general form of systems that are able to deal with unforeseen system states and environmental changes are so called *self-adaptive* systems (Cheng et al., 2009). The "self" prefix indicates that the individual system reasons about its state and environment and decides autonomously how to adapt and self-organize to accommodate changes in its context and environment. In this way a self-adaptive system is able to deal with a changing environment and emerging requirements that may be unknown at design-time. The comparison of the actual state of the system with given higher-level objectives guides the self-adaption process. From control engineering such a circular dependence is known as a feedback loop. Even if, adding automation such as feedback loops can itself lead to unexpected behavior (Brown and Hellerstein, 2005), feedback loops are seen to be one of the main aspects that enable self-adaptive systems to deal with unforeseen system states (Brun et al., 2009).

A first architecture for self-adaptive systems that explicitly exposes the feedback control loop is the Autonomic Element introduced by Kephart and Chess (2003) and popularized through IBM's architectural blueprint for *Autonomic Computing* (IBM Corporation, 2006). IBM uses the metaphor of the autonomic nervous system, which runs our body for us without need for conscious intervention. In the same way autonomic computing systems shall function largely independently from human interventions, adapting, correcting, and repairing themselves whenever problems occur. Autonomic Elements are the basic building blocks of autonomic systems. They contain and manage resources and deliver services to humans or other Autonomic Elements. An Autonomic Element consists of an autonomic manager and one or more managed elements building a feedback loop. The autonomic manager consists of sensors, effectors, and an analysis and planning engine. This engine contains a monitor that filters the data collected from the sensors and stores them in a knowledge base, an analysis component that compares the collected data against desired values, a planning component that develops strategies to correct the trends identified by the analysis component and an execution engine that finally adjusts parameters of the managed element by means of effectors. Autonomic Elements manage their own internal states and their interactions with the environment and other Autonomic Elements driven by the goals and policies the designers have built into the system.

IBM's Autonomic Computing was mainly developed for monitoring and adapting enterprise server applications. An approach that is more general in its focus of application is the so called *generic observer/controller architecture* (see, e.g. Branke et al., 2006; Richter et al., 2006; Schöler and Müller-Schloer, 2006). It was proposed by several researchers in the field of Organic Computing in order to assess the behavior of self-organizing technical systems consisting of large collections of intelligent devices and to introduce a regulatory feedback to control the dynamics of such systems. The main objective of the architecture is to achieve a controlled self-organised behavior, that is, to influence the self-organizing processes on the micro-level of a system in order to control the resulting emergent processes on the macro-level. Compared to classical system design Organic Computing Systems relaying the observer/controller architecture have the ability to adapt and to cope with some emergent behavior for which they have not been designed explicitly. At least three possible objectives which can be realized with an observer/controller architecture exist. The objectives can be to influence the system such that a desired emergent behavior appears, to disrupt an undesired emergent behavior as quickly and efficiently as possible or to construct the system in a way such that no undesired emergent behavior can develop.

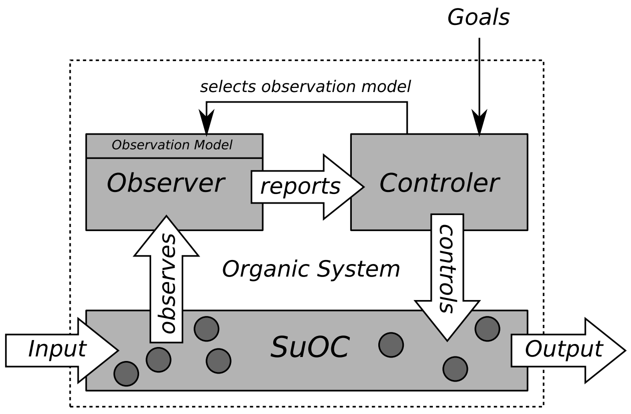


Figure 1.1: The observer/controller architecture

In Figure 1.1 an illustration of the generic observer/controller architecture is given. Three major components play a role, namely the *system under observation and control*

(SuOC), the *observer* and the *controller*. The SuOC is assumed to consist of a large collection of communicating active objects possessing certain common attributes. These objects are relatively simple and in sum they constitute the decentralized self-organizing system which needs to be controlled. It is assumed that the SuOC is able to run autonomously even if the observer and controller are not present. The observer collects and aggregates information on the micro level (about single objects in the SuOC) and on the macro level (global properties of the SuOC). The observation behavior itself is variable and is influenced by the so called *observation model*. The observation model selects the attributes of interest and the needed detectors, it also chooses appropriate analysis tools with regard to certain purposes given and selects appropriate prediction methods. The observation model can be selected by the controller and in this way the controller can direct the attention of the observer to certain observables of interest in the current context. The observer aggregates its observations into so called situation parameters that are reported to the controller component. Based on the comparison of the observed with the expected behavior the controller makes decisions about what actions are necessary to influence the SuOC in the best (known) way.

The observer/controller architecture establishes a control loop on top of the SuOC. Three different architectural variants are suggested: central (one observer/controller for the whole system), distributed (an observer/controller for each subsystem), and multi-level (one observer/controller for each subsystem as well as one (or more) for higher observer/controller levels). In particular, for larger and more complex systems it will be necessary to build hierarchically structured observer/controller systems instead of trying to manage the whole system with one observer/controller.

A drawback of an online trial-and-error process in general is that the system must be allowed to assess suboptimal system states in order to be able compare different control methods and their influence on the system. In running systems, especially if executing safety critical tasks, this is often unwanted. A solution to this problem is to extend the observer/controller architecture to a two-level adaptation and learning architecture by combining online adaption possibilities with a simulation based approach.

In several application of the observer/controller architecture it was shown that the resulting systems can show improved performance. For example, the Organic Traffic Controller approach presented in Branke et al. (2006); Prothmann et al. (2009, 2008) can reduce the average delay time at controlled intersections by extending traffic light controllers with an observer/controller architecture. Cakar et al. (2008) applies the generic observer/controller architecture to the control of a multi-agent simulation of an intersection without traffic lights in order to increase the traffic flow. Schöler and Müller-Schloer (2005) and recently Tomforde et al. (2009) applies the observer/controller architecture to the online adaption of parameters of network systems to dynamic environments.

To sum up, the engineering of self-organizing systems and the question how to utilize the resulting emergent behavior is a very active field of research in computer science. Different concepts have been developed in order to face the problem of finding appropriate local interactions between the components that lead to a desired global coherent behavior of a system.

2 Emergent Sorting and Clustering

One aim of Organic Computing is to utilize Emergence in technical systems. In this chapter we introduce and study two examples of such systems. Many decentralized local interactions between the entities of the systems lead to an increasing order on system level. This emergence of order is the intended purpose of the systems: sorting objects respectively grouping objects depending on their properties.

First we study so called Emergent Sorting Networks. These directed networks consist of router agents with fixed positions and buffer sites that the agents can use to store objects of different types. These objects are inserted randomly into the network. Moved by the agents that use simple local rules, the objects traverse the network. The aim of the whole system is to create a sorted outflow, that is, batches of objects of the same type, out of the network. We study different local routing rules on varying network topologies in terms of sorting performance and fairness.

In the second part of the chapter we investigate a new type of a decentralized clustering problem. The problem is to cluster packets that are sent around in a network. We propose an algorithm, called DPClust, for solving this problem. In this algorithm the clustering is achieved in a decentralized manner by the routers of the network using simple local rules to modify the packets and their cluster membership. We study the behavior of DPClust for different problem instances and for networks consisting of several subnetworks. We apply DPClust and two variants of DPClust to dynamic problems and study their performance.

2.1 Emergent Sorting in Networks of Router Agents

The following description of an industrial problem, arising in various industrial settings, was given in Brueckner (2000):

> *Given a segment of a transport system of arbitrary layout in discrete high-volume production [environments] composed of unidirectional line-buffers (e.g., conveyors) and multi-input multi-output sequential routing devices (e.g., rotation tables, lifts), and assuming that the workpieces sent through the segment are all of one product but may be differentiated on the basis of the value of one product parameter; how may the segment be controlled in a decentralized manner so that the outflow of workpieces occurs in batches of workpieces of the*

> *same product parameter value with the average batch size of the outflow being significantly higher than that of the inflow?*

This question becomes important in case the outflow of such a system is to be processed by machines that have to pay setup costs every time the considered product parameter changes. In order to minimize the setup cost over time, the system must minimize the probability of a parameter change at the outflow.

Brueckner (2000) states that the solution to such a sorting problem requires a new approach to control, based on self-organization rather than on a central controller. The reason is that the problem is highly dynamic: at any time new workpieces may enter the system while others leave it. Moreover, system parameters such as the volume of the inflow may vary strongly over time. It may not even be known how many different product variants have to be handled at a time.

The following distributed solution to the batching is proposed in Brueckner (2000): to each sequential routing device a so-called router agent is assigned. These agents act autonomously from each other. Agents can move a workpiece from an entry of its router to one of its exits. In its memory the agent stores for each exit the value of the product parameter of the last workpiece that has passed this exit. Having available multiple entries and multiple exits, a router agent must decide on which workpiece to move to which exit. These decisions are taken by a set of simple rules that reportedly result in a batching (sorting) behavior of the system. That is, from the local rules of the agents and their interaction with the traversing objects at a higher system level order observable through an increased batch size emerges.

In the following we will study several aspects of what we call Emergent Sorting Networks, an abstraction of the mentioned industrial system. First, we investigate and modify the set of simple local rules for the agents presented in Brueckner (2000). Second, we introduce and study an agent routing behavior that is based on pheromones, as used for example by ant colonies while foraging (see, e.g., Wyatt, 2003). The original proposal was limited to networks with square shape, as a third point, we thus study simpler networks that are composed of router agents organized in a line. We try to deepen the understanding of Emergent Sorting Networks by means of extensive experiments based on different routing rules, different network layouts, different number of object types, and different number of agents.

To the best of our knowledge, these systems of router agents have never been studied in great detail. They were first mentioned in the context of the ESPRIT LTR project MASCADA. In the PhD thesis (Brueckner, 2000) they served for motivating the work carried out in the context of the thesis. And in a poster paper presented in the proceedings

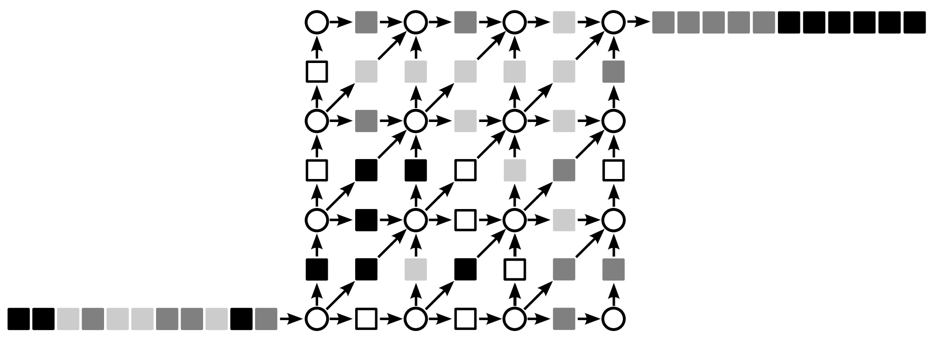


Figure 2.1: Square shaped sorting network; buffer positions are depicted as squares; white buffer positions are free; colored buffer positions indicate the types of objects stored; agents are shown as circles; input/output sequences of objects are shown as sequences of squares, in the same style as buffer positions

of the Genetic and Evolutionary Computation Conference (GECCO-2006) some limited experiments were presented (Tozier et al., 2006).

2.1.1 Sorting Networks of Router Agents

Abstracting from the industrial problem we study so called Emergent Sorting Networks. The basic components of Emergent Sorting Networks are *router agents*. Each router agent $a \in \mathcal{A}$ has an input and an output buffer with n, respectively m, positions. We denote the input buffer positions by $x_1, \ldots, x_n$, and the output buffer positions by $y_1, \ldots, y_m$. A buffer position can store exactly one object of k different types $t_1, \ldots, t_k$. Router agents are connected to form a network topology by associating output buffer positions with input buffer positions of other agents. This is done with respect to the following conditions: (1) All input/output buffer positions must be involved in connections. (2) A one-to-one relationship between associated output and input buffer positions must hold. (3) The connections must be such that the resulting network is acyclic. Note that the set $\mathcal{A}$ of agents contains two special agents: the so called *inflow agent* serves to feed the network with incoming objects and the *outflow agent* produces the output of the network.

Router agents can pick up objects from their input buffer positions and move them to a free output buffer position (if any). At each time step, in case the input position of the inflow agent is empty it is filled with an object of random type. Thereafter, in random order all agents apply their local routing rules. Within a time step every object can be moved at most once. The network starts empty.

In Figure 2.1 the network structure that was originally proposed in Brueckner (2000) is shown. Circles represent routing agents, buffer positions are depicted as squares and arrows indicate the possible transportation direction of objects. White buffer positions are empty, and a colored buffer position indicates the type of object it stores. The agent on the bottom left is the inflow agent and on the top right is the outflow agent. The input/output sequences of objects are shown as sequences of squares, in the same style as buffer positions. The Figure shows how a random sequence of objects is inserted into the system. The objects are moved by the agents and traverse the network. The sequence of object types at the outflow of the network is more sorted, i.e., batches of objects of the same type are build.

2.1.2 Investigated Variants of Agent Behaviors

In the following the different investigated agent behaviors, i.e., the considered variants of local routing rules are introduced in detail.

Original Behavior

The behavior of the agents used in the original system proposed in (Brueckner, 2000) is given in the following. In this approach every agent memorizes for each of its output buffer positions the type of the last object that was moved to this position (if any). If it is an agents turn to perform an action the agent first tries to make a "good move", i.e., the agent tries to move an object to a position for which it has memorized the type of the object to be moved. If this is not possible, with a certain probability it moves a random object to a random output buffer position. This probability depending on the fraction of occupied input buffer positions to all input buffer positions. This implies that if only few input buffer position are occupied with a high probability the agent does not move any object. For details see Algorithm 1.

Algorithm 1 Original Behavior of an Agent $a \in \mathcal{A}$

```
1: if it exists an unmoved object $o$ of type $t$ in the input buffer positions and a free
   output buffer position $y_j$ for which the agent has memorized the type $t$ then
2:     Move $o$ to $y_j$
3:     Memorize type of $o$ for $y_j$
4: else
5:     if exists a free output buffer position $y_j$ and an unmoved object $o$ in the input buffer
       positions then
6:         Let $r$ be the number of unmoved objects in input buffer positions of $a$
7:         Choose a random number $p \in [0, 1]$
8:         if $p < r/n$ then
9:             Move $o$ to $y_j$
10:            Memorize type of $o$ for $y_j$
11:        end if
12:    end if
13: end if
```

Pheromone Based Behavior

In the following we introduce a pheromone based variant of the agent behavior. For each agent $a \in A$ and for each object type t_i $(i = 1, \dots, k)$ we introduce a pheromone value $0 \leq \tau_i^a \leq 1$. All pheromone values are initially set to $1/k$. If it is an agents turn to perform an action it chooses an object based on the pheromone values for the different object types and puts it on a random empty output buffer position. The higher a pheromone value for a certain type, the higher the probability to choose objects of this type. The calculation of the probabilities works in the same way as used in the well known Ant Colony Optimization (ACO) metaheuristic (see, e.g., Dorigo and Stuetzle, 2004). The detailed formula is given in Line 5 of Algorithm 2. After an agent $a \in A$ has moved an object of type t_s, the pheromone values of agent a are updated (see Line 7 of Algorithm 2). The pheromone value belonging to type t_s is increased, thus the probability to move this type of objects again increases. All other pheromone values are decreased. The parameter β gives the learning rate, i.e., how strong the pheromones are updated. It has the similar functionality as the pheromone evaporation parameter of the ACO algorithm. For this parameter appropriate values depending on the used system parameters must be found.

Algorithm 2 Pheromone-Based Behavior of an Agent $a \in \mathcal{A}$

```
1: T is the set of types of the (within this time step unmoved) objects in the input buffer
   of $a$
2: Choose a random number $p \in [0, 1]$
3: **if** $p < |T|/n$ **then**
4:   **if** it exists at least one free output buffer position **then**
5:     Choose a type $t_s \in T$ according to the following probability distribution:
         $$\mathbf{p}(t_i) = \frac{\tau_i^a}{\sum_{t_l \in T} \tau_l^a} \quad \forall\, t_i \in T$$
6:     Move an unmoved object $o$ with type $t_s$ to a random free output buffer position
7:     Update pheromone values: $\tau_j^a := \tau_j^a + \beta(\mu_j - \tau_j^a), j = 1, \dots, k$, where $\mu_j = 1$ in
       case $j = s$ and $\mu_j = 0$ otherwise
8:   **end if**
9: **end if**
```

New Waiting Rule

In the agents' behaviors as given in Algorithm 1 and Algorithm 2 the agents wait depending on the number of occupied input buffer positions. We also test a variation concerning this original waiting rule. Using the so called new waiting rule an agent is only allowed to act if all its input buffer positions are occupied by objects, i.e., Line 8 of Algorithm 1 and Line 3 of Algorithm 2 are replaced by "**if** no input buffer position is empty **then**".

2.1.3 Experimental Evaluation

In order to study if networks need to be square-shaped for exhibiting a sorting behavior, we also consider networks that are simply composed of a line of agents, as shown in Figure 2.2.

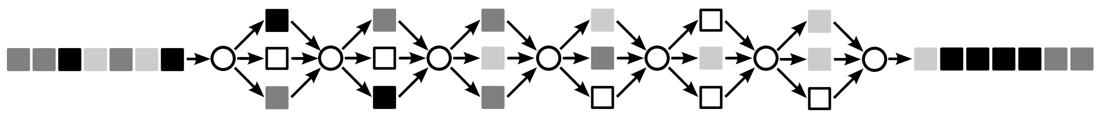


Figure 2.2: Line shaped sorting network

All presented results were obtained by simulation. We use the following notation for the different options outlined before. The notation XYZ consists of three letters, where:

- $X \in \{\mathbf{B}, \mathbf{P}\}$. Hereby, **B** denotes the original agent behavior (Algorithm 1), and **P** denotes the pheromone-based agent behavior (Algorithm 2).
- $Y \in \{\mathbf{o}, \mathbf{n}\}$. Letter **o** refers to the original waiting rule, i.e. the probability to act is proportional to the fraction of occupied input buffer positions, whereas **n** corresponds to the system using the new waiting rule, i.e., the agents only act when all their input buffer positions are occupied.
- $Z \in \{\mathbf{s}, \mathbf{l}\}$. This identifier refers to the network structure. Letter **s** indicates a square-shaped network, and letter **l** refers to a network in shape of a line.

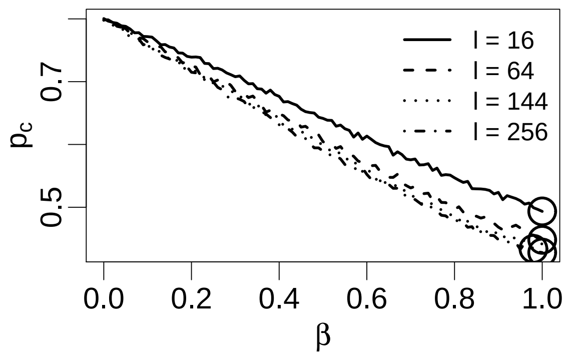
Measures of System Performance

We measure the performance of the system as the probability that at the outflow a change in the object type can be observed. The lower this probability, the longer the batches of objects of the same type and therefore the better the performance of the sorting network. This measure will henceforth be denoted by p_c. If not stated otherwise, for each parameter set 50 000 time steps were simulated.

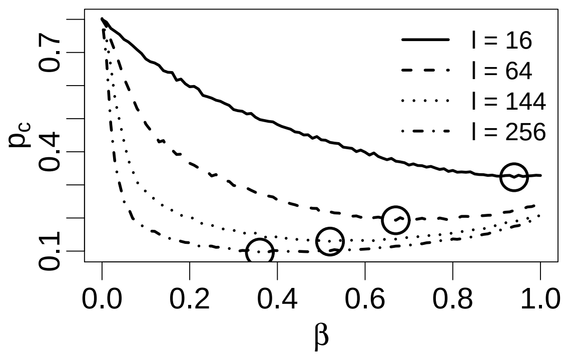
Additionally, the number of time steps that the objects spend in the systems (plus maximum and mean of these values) and how many objects leave the systems when simulating 1 000 000 time steps were measured.

Tuning

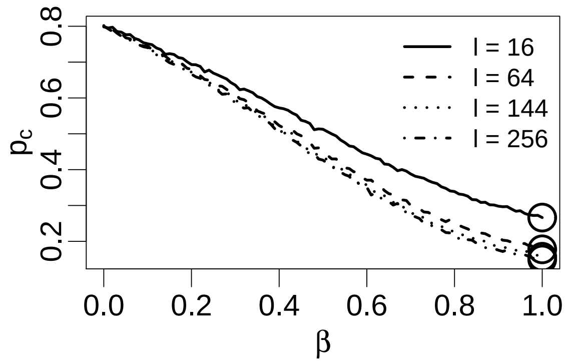
As mentioned previously, the pheromone-based agent behavior requires an appropriate setting of the parameter $\beta \in [0, 1]$. In order to find good values for this parameter the values $\beta \in \{0, 0.05, 0.1, \ldots, 1.0\}$ have been tested in different sorting networks. From the



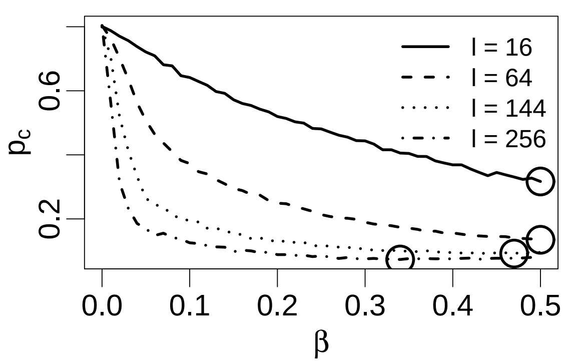
(a) **Pos**, **O**riginal waiting rule, **S**quare network



(b) **Pol**, **O**riginal waiting rule, **L**ine network



(c) **Pns**, **N**ew waiting rule, **S**quare network

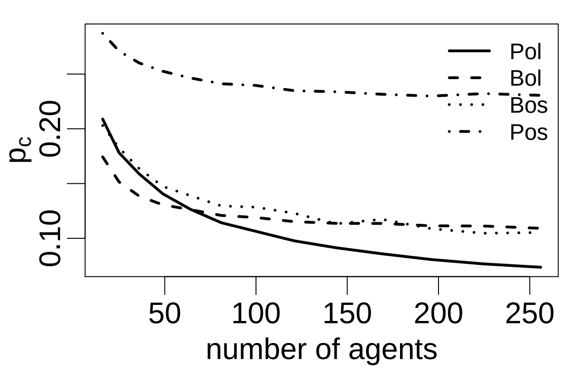


(d) **Pnl**, **N**ew waiting rule, **L**ine network

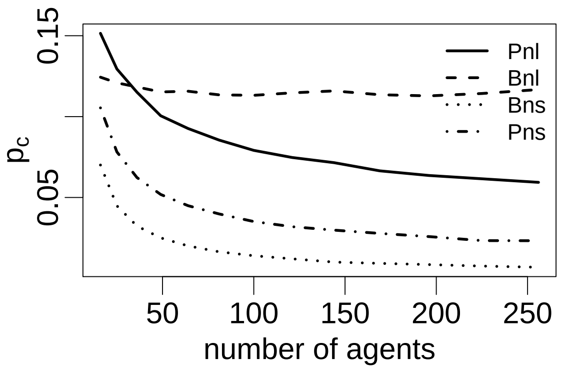
Figure 2.3: Tuning results for systems for the pheromone based systems **Po*** (top) and **Pn*** (bottom) for the square (left) and the line topology (right); 5 object types; circles indicate lowest values found

results for every system parameter combination we determine the value of parameter β that leads to the best performance p_c. These values were used in all subsequent experiments with pheromone based systems.

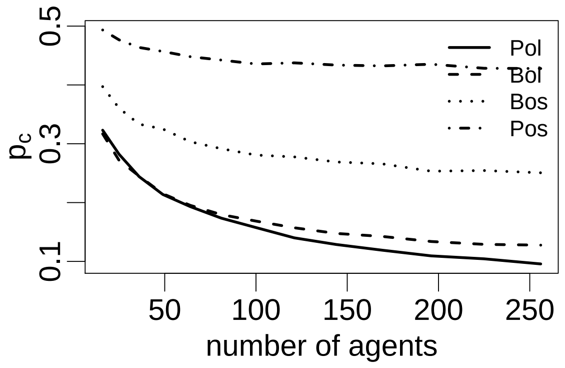
In Figure 2.3 tuning results of the pheromone based systems in terms of the measure p_c, i.e., the probability of a type change in the outflow, are presented. Each subfigure contains four performance curves, corresponding to four different network sizes: 16, 64, 144, and 256 agents. Circles in the plots indicate the lowest measured p_c for each network size. Following conclusions can be made: First, in square-shaped networks an agent should always try to repeat the action of the previous time step ($\beta = 1$). Second, in line-shaped networks the more agents used, the smaller the value of β should be. Third, the optimal value for β also depends on the different waiting rules and also slightly on the number of types (results not shown).



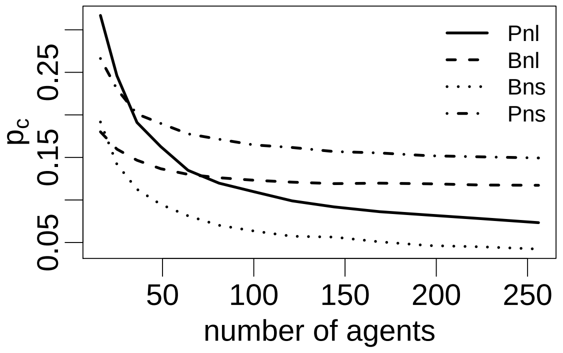
(a) original waiting rule; 3 object types



(b) new waiting rule; 3 object types



(c) original waiting rule; 5 object types



(d) new waiting rule; 5 object types

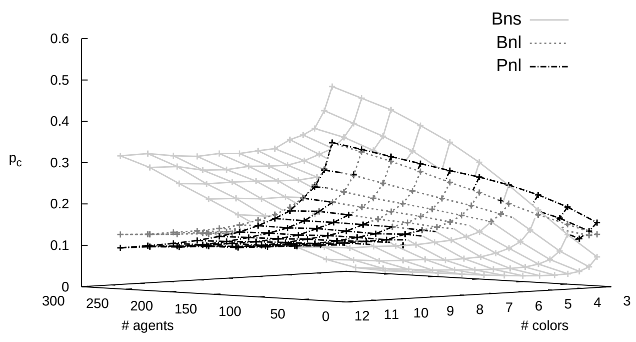
Figure 2.4: Performance p_c over the number of agents for systems ***o*** (left) and systems ***n*** (right) for 3 (top) and 5 (bottom) types.

2.1.4 Results

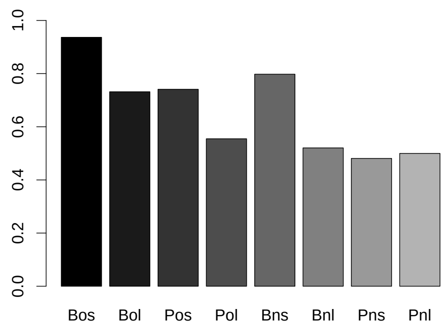
Figure 2.4 presents the probability of a type change p_c for the original system as well as for the proposed variants. Results are shown for different numbers of object types $\{3, 5\}$ dependent on the number of agents.

Concerning the original waiting rule (systems $*\mathbf{o}*$ in Figure 2.4(a) and (b)), one can observe that the sorting in the line-shaped networks is in general better than the sorting in the square-shaped networks. This trend becomes stronger the more object types are used. A second observation that can be made is that the pheromone-based systems greatly improve over the original systems when line-shaped networks with many agents are used. The opposite is the case for square-shaped networks.

Interestingly, the results concerning the new waiting rule of the agent behaviors (systems $*\mathbf{n}*$ in Figure 2.4(c) and (d)) look quite different. Here the original system in conjunction with a square-shaped network (**Bns**) works best. When three types of objects are in the systems, square-shaped networks outperform line-shaped ones. However, when the number



(a)



(b)

Figure 2.5: Performance results concerning higher numbers of types (a); throughput of all systems (b)

of types is increased, the performance of the pheromone-based system on a square-shaped network drops, but the performance of the line-shaped networks improves.

In order to study this effect in more detail, we performed experiments using up to 12 different object types. In the outcome the three systems **Bns**, **Bnl**, and **Pnl** showed the best performance. Their results are given in Figure 2.5(a). An interesting effect can be observed here. The results show that for a low number of object types the system **Bns** outperforms both line-shaped systems, but when there are more object types in the systems the number of agents becomes important: Using few agents system **Bnl** is best, whereas using many agents **Pnl** is the best performing system.

When comparing the performance of the systems with the original waiting rule (***o***) to the performance of systems using the changed waiting rule (***n***), there is a clear advantage of the modified systems. The best-performing system with the new waiting rule (**Bns**) performs always better than the best-performing system with original rule (**Pol**).

System	Average	Standard Deviation	Maximum
bos	257.5	354.7	16194
bol	241.7	16.1	341
pos	335.8	203.9	5600
pol	381.1	13.30	482
bns	302.9	301.1	11155
bnl	433.8	19.20	587
pns	527.6	310.2	6158
pnl	496.0	11.20	600

Table 2.1: Results concerning the time objects stay in the system.

In the second column of Table 2.1 the average time an object spends in the different systems is given. The third column contains the standard deviation of these times and the fourth column provides the maximal time an object spent in the system. In general, the pheromone-based systems are characterized by a longer "average time in system" but the "maximal time in system" is greatly reduced as compared to the original systems. The line-shaped networks show a short "maximal time in system" and are more fair since the times the different objects stay in the system has a lower variance. In addition, we can observe that the changed waiting rule (systems $*\mathbf{n}*$) leads to an increase in both average and maximal time that an object stays in the system.

In Figure 2.5(b) the throughput is given, i.e., the number of objects that traverse the system in 1 000 000 simulation steps. It can be seen that in systems using the original waiting behavior more objects leave the system, because the agents can act more often and do not need to wait. Pheromone based systems have a lower throughput than the original systems, because in pheromone based systems the waiting rule is always applied, whereas in the original system the waiting rule is only applied if no "good" move can be made (see Line 1 in Algorithm 1).

2.1.5 The Increase of Order in Emergent Sorting Networks

Based on local decisions of the router agents the proposed Emergent Sorting Networks sort random sequences of objects of different types. As a result an increase of order can be observed on system level. As stated in Section 1.2, Mnif and Müller-Schloer (2006) defines emergence as the formation of order from disorder based on self-organising processes. In the same work an approach to quantify emergence is proposed. This measure is based on Shannon's information theory, in particular on the information-theoretical entropy.

To calculate the proposed measure is done as follows. Given an enumerable attribute A of the system, the relative frequency of the occurrence of each possible attribute value i can be represented as probabilities p_i. The entropy of the system regarding attribute A is given by

$$H_A = -\sum p_i \log_2 p_i.$$

The unit of entropy is bit and gives the amount of information stored in the observed attribute. The quantitative emergence is then defined as "the increase of order due to self-organised processes between the elements of a system S in relation to a starting condition of maximal disorder" (Mnif and Müller-Schloer, 2006):

$$\text{Emergence } M_A = H_{max} - H_A - H_{view},$$

where H_A is the entropy value of the system as defined above, measured at a certain point of time, H_{max} serves as a reference value in order to normalise H_A and H_{view} is the amount of entropy that depends both upon the chosen attribute and the abstraction level. Observations made on a higher abstraction level result in a lower entropy value.

In the following we will neglect H_{view} and study how the choosen perspective influences the observed increase of order $H_{max} - H_A$ in Emergent Sorting Network systems. Consider an Emergent Sorting Network sorting objects of k types. As the observed attribute we choose the object types in the outflow of the network. To calculate the information-theoretical entropy of the outflow the relative frequencies of the object types are used. Because only a constant number of objects can actually be in the network, the relative frequencies of the different types are the same as in the inflow of the network. Thus, the entropy of the outflow is the same as the entropy of the inflow. No increase of order can be quantified observing the object types. As an example consider the inflow `rbgbgrrgb` (different observable object types are denoted by symbols `r`,`g`,`b`,…) and the outflow `rrrgggbbb`. The relative frequency of all symbols is 1/3 in both strings, which leads to an entropy $-3 \cdot \frac{1}{3} \log_2 \frac{1}{3} = 1.584963$ bits for both.

We now consider another attribute, i.e., in terminology of Mnif and Müller-Schloer (2006) we change the observation model. We observe the changes of the object types, i.e., we consider a “color blind” observer that can only determine that a type change occurred but not exactly which types were involved. That is, we consider the entropy of binary strings in which the symbol `1` denotes an observed change of the objects’ type in the outflow and the symbol `0` stands for no change. For example, the inflow sequence `rbgbgrrgb` translates to `11111011`, whereas the outflow `rrrgggbbb` is represented as `00100100`. For a given probability p of a type change (this is the also measure we used in our simulation experiments) the relative frequency of the symbol `1` (respectively `0`) is p (respectively 1-p). Thus, the increase of order can be calculated as:

$$H(p) = -p \log_2 p - (1-p) \log_2 (1-p).$$

When using k different object types, the probability that a type change occurs in the random inflow is $(k-1)/k$ and that two successive objects have the same type has the probability $1/k$ respectively. Therefore, the entropy of the inflow is $H_{max} = H((k-1)/k)$. Using this entropy as reference, for an (empirically measured) type change probability p in a system with k different object types we can calculate the emergence as

$$H((k-1)/k) - H(p).$$

As an example consider a system with $k = 10$ different object types and an observed type change probability $p = 0.7$. In the random input sequence the probability of a type change is $(k-1)/k = 0.9$. Thus, when the random input sequence traverses the Emergent Sorting Network the type change probability decrease from 0.9 to 0.7. This leads to an increase of order of

$$H_{max} - H_A = H((k-1)/k) - H(p) = 0.4689956 - 0.8812909 = -0.4122953.$$

Thus for a "type blind" observer the entropy in the considered attribute has increased, i.e., the order has decreased.

We change the observation model again. As the observed attribute we now consider the exact types of two successive objects. That is, the symbols representing possible attribute values are tupels of two types. For example, the sequence of objects `rbgbgrrgb` translates to `(rb)(bg)(gb)(bg)(gr)(rr)(rg)(gb)`, where `(rb)` is the symbol representing an observed type change from type `r` to type `b`. Hence, k different "even" symbols representing two consecutive objects of the same type exist, e.g., `(rr)` or `(gg)`, and $k \cdot (k-1)$ different "odd" symbols relating to type changes, e.g., `(rg)`, `(gr)` or `(rb)`, can be observed. Given a (measured) type change probability p, the probability to observe an even symbol is $(1-p)/k$ and the probability to observe a change from a specific type to another one (e.g., `(rb)`) is $p/(k(k-1))$. Thus, the entropy under this observation model of an outflow consisting of objects of k different types and an observed probability of a type change p can be calculated as

$$\begin{aligned} H(p,k) &= -k\frac{1-p}{k}\log_2\frac{1-p}{k} - k(k-1)\frac{p}{k(k-1)}\log_2\frac{p}{k(k-1)} \\ &= -(1-p)\log_2\frac{1-p}{k} - p\log_2\frac{p}{k(k-1)} \end{aligned}$$

Calculating the increase of order in the example system with $k = 10$ object types and an observed type change probability $p = 0.7$ now leads to $M = H(0.9, 10) - H(0.7, 10) = 1.928255$.

As have be shown, depending on the observers view on the Emergent Sorting Network systems no, a negative, or a positive increase of order can be calculated. Thus, this confirms the mentioned importance of the view of the observer and emphasises a careful application of emergence measures based on the information-theoretical entropy.

2.2 Decentralized Packet Clustering

In the following we will study a second system utilizing emergence. Again, based on many local decentralized decisions of the autonomous parts of the system, a desired global property emerges. This desired property is the grouping of objects in a way, that the similarity of objects within the same group is higher than between objects from different groups. That is, a so called clustering for the objects is generated. In the considered system the objects are packets in a network and the decentralized decisions are made by the routers of the network.

2.2.1 Distributed and Ant-Inspired Clustering

Clustering as the "identification of homogeneous groups of objects" (Arabie et al., 1996) is one of the core processes in data mining and important for many other applications in the sciences, as well as in commercial and economics areas. Usually clustering is performed as an unsupervised task for the classification of patterns (observations, data items, or feature vectors) into groups called clusters. Unsupervised means that the types and characteristics of the clusters are unknown in advance and have to be discovered in the clustering process. For a comprehensive overview of clustering methods and applications (see, e.g., Everitt et al., 2001; Gan et al., 2007; Jain et al., 1999).

Distributed clustering referes to algorithms and methods for parallelizing and distributing clustering algorithms and has been addressed for example in the Distributed Data Mining community, for a detailed survey the interested reader is referred to Kargupta and Sivakumar (2004). In recent years, with the evolution of large peer-to-peer networks distributed clustering became an important and intensively studied field (Datta et al., 2006a). Also in the field of sensor networks techniques are required for the distributed clustering of dynamic data streams (Beringer and Hüllermeier, 2006; Gaber et al., 2005; Hua et al., 2009; Lambertsen and Nishio, 2004).

k-Means

One of the most often used algorithms for clustering is called k-means. This iterative algorithm starts with a set of k initial data vectors, called center points. Within one iteration each object is assigned to its nearest (measured, e.g., with respect to the Euclidean distance) center point. All objects that are assigned to the same center point form a cluster. For each cluster its centroid is computed and these centroids form the new center points for the next iteration of the algorithm. The algorithm stops when some convergence criterion has been met, e.g., the center points have not changed or a maximal number of iterations has been done.

Several distributed or parallel versions of the k-means algorithm have been proposed in the literature. The aim of most of these algorithms is to provide a fast parallel or distributed implementation of k-means or one of its variants. Problems here are how to exchange the necessary information between the processors (see e.g. Datta et al., 2006b; Dhillon and Modha, 2000; Eisenhardt et al., 2003) or how to distribute the workload, for example, on shared-memory multi-core machines (Chu et al., 2007). Variants of k-means have been proposed to work on peer-to-peer systems (Datta et al., 2009) as well as for dynamic stream data mining (Shah et al., 2005).

Ant-Based Clustering

Several clustering algorithms have been proposed that are inspired by the behavior of ants. Ant based clustering and sorting has several different sources of inspiration. First and most famous is the clustering behavior of ants that relates to two types of behavior that can be observed in real ants. First is the formation of cemeteries (piles of corpses of dead nest mates) that can be found, for example, in the ant *Pheidole pallidulais* (Deneubourg et al., 1990; Theraulaz et al., 2002). The second, more sophisticated behavior is the spatial arrangement of items of different kinds according to their properties. This can be observed for example in nests of the ant *Leptothorax unifasciatus*, where larvae are arranged depending on their size.

In Deneubourg et al. (1990) a model was introduced to explain this clustering behavior. In this model agents (representing ants) move randomly on an array of cells. In this array initially randomly distributed items are located. The agents make probabilistic choices to pick up or drop items depending on the fraction of cells occupied by items in a defined neighborhood. Eventually this behavior leads to the emergence of clusters of items. In Chapter 3.3 this model will be introduced in more detail.

Although this clustering model was developed for use in collective robots, soon it was applied to data analysis, too. Lumer and Faieta (1994) proposes a basic ant-based data clustering algorithm closely related to the ant clustering model described in Deneubourg et al. (1990). Later several authors introduced modifications and extensions to this algorithm: speeding up the algorithm by moving directly to items (Monmarché et al., 1999), introducing pheromone values (Abraham and Ramos, 2003; Ramos and Merelo, 2002) or kernel functions (Peterson et al., 2008) to guide the ants to interesting regions, the adaptive setting of the algorithm parameters (Handl et al., 2006; Vizine et al., 2005), transportation of entire heaps of items (Kanade and Hall, 2003), introduction of a short term memory (Handl and Meyer, 2002; Peterson et al., 2008), communication between the agents (de Oca et al., 2005a), hybridization with other heuristics for example fuzzy c-means and k-means (Gu and Hall, 2006) or neural networks (de Oca et al., 2005b), and using fuzzy rules for dropping and picking up items (Kanade and Hall, 2003; Schockaert et al., 2004, 2007).

In Handl et al. (2003a,b) ant-based clustering is compared with k-means clustering, with a hierarchical agglomeration clustering method, and with one-dimensional self-organizing map clustering. It is shown that ant-based clustering performs competitively to these standard algorithms.

Another source of inspiration for ant based clustering algorithms is the self-assembling behavior of ants, i.e., the ability of ants to build live structures with their bodies. In Azzag et al. (2003) an algorithm called AntTree is proposed. In the algorithm every ant represents a data vector and is initially placed on the root of a tree. Based on their similarity with the ants already attached to the root, the ants move and attach themselves to the tree. The algorithm is used for example for texture segmentation (Channa et al., 2006).

A third possibility is to see clustering problems as optimization problems with the clustering quality as objective function. Runkler (2005) uses the Ant Colony Optimization metaheuristic to solve clustering problems, e.g., to cluster the lung cancer test data in the UCI Machine Learning Repository.

Labroche et al. (2002, 2003a,b) proposes an approach for ant inspired clustering that is inspired by the chemical recognition system of ants. By continuously exchanging chemical cues ants are able to discriminate between nestmates and intruders, and in this way they can create homogeneous groups of individuals sharing a similar odor. In the proposed algorithm the objects to be clustered are represented by artificial ants and clusters as ant nests. Ants (objects) can belong to nests (clusters). In the algorithm iteratively two random chosen ants meet and depending on the objects they represent and an adaptive threshold they determine whether they accept each other as being from the same nest. When two ants A and B meet different rules are applied: (1) If A and B are without a nest and accept each other, they build a new nest. (2) If A has no nest and B already belongs to a nest and A accepts B, then A joins in the nest of B. (3) If A and B accept each other and are already in the same nest they feel more comfortable in this nest (the comfortable feeling is an estimation on how good the ant fits into its cluster). (4) If A and B do not accept each other although they are in the same nest, the ant that feels less comfortable has to leave the nest. (5) If A and B accept each other and are from different nests, the ant from the smaller nest joins in the bigger one. It has been shown, that applying these rules iteratively the algorithm eventually finds a good clustering of artificial as well as real data sets.

An extension of AntClust called Visual AntClust uses two dimensional vectors as labels for a nest (Labroche et al., 2003b). The values of these labels are chosen so that nests with similar ants are placed nearby within the two dimensional space and are changed dynamically in the algorithm during meetings of the ants. It was shown in Handl et al. (2003a,b) that AntClust and Visual AntClust lead to competitive results when compared to k-means clustering for a fixed k.

2.2.2 Problem Formulation

In the following we study a new scenario for clustering that is relevant for distributed applications in networks. We assume that information packets for such an application are send around between the servers of the network. Additionally to the server nodes we assume router nodes in the network. In each server node an application process is running that uses the information in the packets that are send to it. In order that the application process can handle the packets appropriately we assume that the information packets have to be clustered according to a data vector that each packet contains. Additionally each packet contains its cluster identification number. Since the application is distributed over several application processes that run decentralized in the network and there is no central process that knows all packets and could do the clustering, we are interested in a *Decentralized Packet Clustering*.

In the following we will concentrate on the clustering problem and do not model the servers in our problem formulation. It depends solely on the application processes running on the servers, how the clustering information in the packets is used. Possible applications are manifold. For instance, in Section 4.5 of this thesis the Decentralized Packet Clustering is used for a network based approach to the problem of task allocation in so called Organic Support Systems. Janson et al. (2008) used the method for the decentralization of swarm intelligence algorithms that run on systems of connected, autonomous components.

2.2.3 The DPClust Algorithm

The *DPClust* algorithm is executed by the routers of the network and realizes the Decentralized Packet Clustering. The idea behind DPClust is that every information packet (in the following we call information packets simply packets) contains additionally to its data vector and its cluster number an estimation of the centroid of its actual cluster. While traversing the network, packets meet copies of other packets in the routers. If the traversing packet is assigned to the same cluster as the copied one, the data vector of the copied packet is used to update the packet's estimation of the centroid of the cluster. If, on the other hand, the copied packet is from a different cluster, for the traversing packet it has to be decided based on the two available centroid estimations if it is assigned to the cluster of the copy. Before the packet leaves the router its copy replaces the old copy. In this way packets do not need to wait for other packets to meet in the routers, the routers just copy and store the relevant information from the last packet.

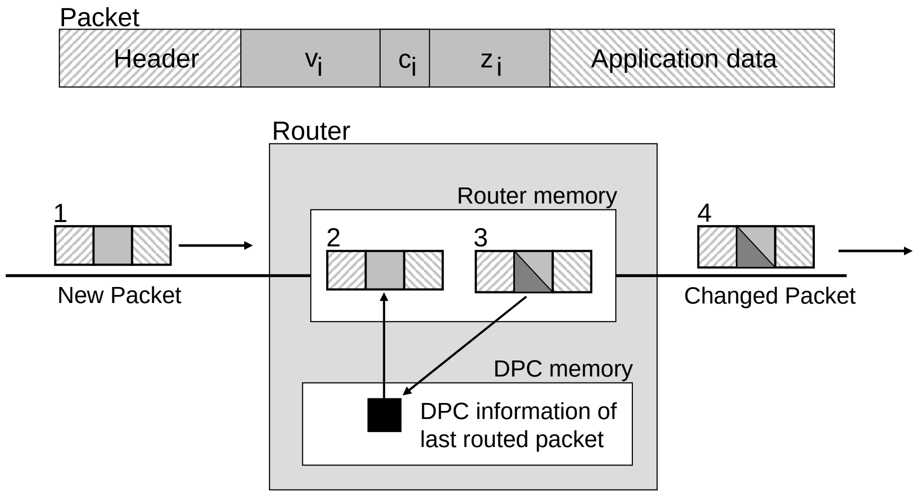


Figure 2.6: Scheme of DPClust in a router; structure of a packet (top); router with packet at different stages (bottom): 1) incoming new packet, 2) DPC information of the new packet is compared with the copy of the DPC information from the last packet, 3) DPC information of the packet is possibly changed and copied into the DPC memory of the router, 4) the packet leaves the router

DPClust has some similarities with the AntClust algorithm (see last Section). AntClust is also based on local meetings of artificial ants representing the objects that are to be clustered. When meeting, the ants make decisions about their nest (cluster) membership based on local information carried by the ants as well. The differences between DPClust and AntClust are the following: (1) DPClust uses centroid estimations, whereas in AntClust no centroids play a role. (2) In DPClust a packet is always associated to a cluster, whereas in AntClust there can be ants without a nest. (3) In DPClust packets only meet copies of other packets, thus only one of the meeting partners can be modified. (4) In DPClust there is no such thing as a “comfortable feeling”.

DPClust can be seen as a form of a distributed k-means algorithm since estimated centroids play a central role for determining the cluster of an information packet. The algorithm runs in the routers of the network but the computational effort of the routers is small. Also the memory requirements of the algorithm in the routers is small. Each packet does not store much additional information and the algorithm does not establish a control protocol that requires communication between the routers. There is no central control in a network using DPClust and the clustering emerges from the interactions of many packets within the routers.

More technically, consider a set of Packets $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ that are send around in the network. Each packet $P_i \in \mathcal{P}$ contains a data vector v_i. Based on this data vectors the packets have to be clustered. DPClust extends every packet P_i by a number

c_i, which denotes the actual cluster the packet is assigned to, and a vector z_i that is the packet's estimation of the centroid of this cluster. Together with the data vector this is called the DPC information of the packet. A packet might contain additional information, e.g., header information and application data but this is not relevant for the clustering algorithm. Thus, a packet P_i can be characterized by its DPC information, i.e., $P_i = (v_i, c_i, z_i)$ for $i \in [1 : n]$.

The cluster C_l is defined as the set of all packets with cluster number l, i.e. $C_l = \{P_i \mid c_i = l\}$. Thus the generated clustering $\mathcal{C} = \{C_1, \ldots, C_{\max\{c_i\}}\}$ is a partition of the set of all packets $\mathcal{P}$.

A router only stores a copy of the DPC information of the last packet that has passed the router. The main idea of DPClust is that the router compares the information of an arriving packet $P_i = (v_i, c_i, z_i)$ with the corresponding information $P = (v, c, z)$ that were copied from the predecessor packet. If both packets are assigned to the same cluster (i.e., they have the same cluster number $c_i = c$) the centroid estimation z_i of the new packet is updated by moving it into the direction of the data vector v of the stored packet. If, on the other hand, the cluster numbers of the packets are different, the router decides whether to reassign the new packet to the cluster of the copied one. This decision is made, when the distance between the data vector v_i and the centroid estimation z_i is larger than the distance to the estimated centroid z of the copied packet P. Formally and in more detail DPClust is given in Algorithm 3 (see also Figure 2.6).

2.2.4 Experiments

Test Networks

Each network $\mathcal{N}$ consists of a set of $r \geq 1$ subnetworks $N_1, N_2, \ldots, N_r$. Each subnetwork contains at least one router and each router is assigned to a subnetwork. Thus when $\mathcal{R} = \{R_1, R_2, \ldots, R_m\}$ with $m \geq r$ is the set of routers and $\mathcal{R}_i$ is the set of routers assigned to subnetwork N_i then $(\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_r)$ is a partition of $\mathcal{R}$. We assume that the routers within a subnetwork are fully connected. We study three types of topologies for the connection of the subnetworks, namely ring networks, fully connected networks, and star networks. A ring network N consists of a directed ring of subnetworks $N_1, N_2, \ldots, N_r$ so that $N_{(i+1) \bmod r}$ is the successor of N_i. In the fully connected network each subnetwork is directly connected to every other subnetwork. In the star network the subnetwork N_1 is connected to every other subnetwork and vice versa.

Each packet is assigned to a subnetwork. Let $f(i)$ be the index of the subnetwork packet P_i is assigned to. For our experiments we assume that all packets already exist in the network from the start and that all packets have an unlimited life time. In general our methods will also work when new packets arrive or packets are removed from the network

Algorithm 3 DPClust

Let $P = (v, c, z)$ be a copy of the data vector, the cluster number, and the estimate of the centroid of the last packet that was processed.

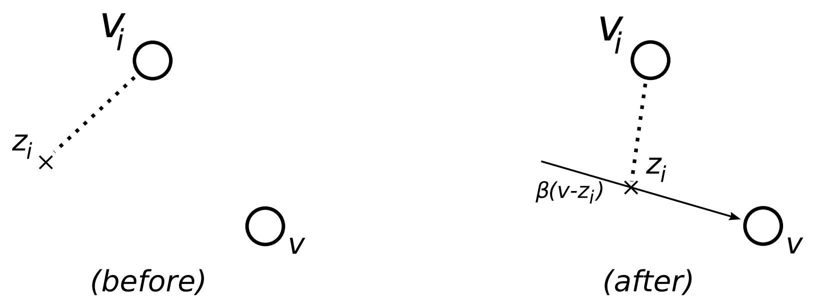
Let $P_i = (v_i, c_i, z_i)$ be a new arriving packet.

if P_i is in the same cluster as P, i.e., $c_i = c$

then update the estimate of the centroid of P_i by

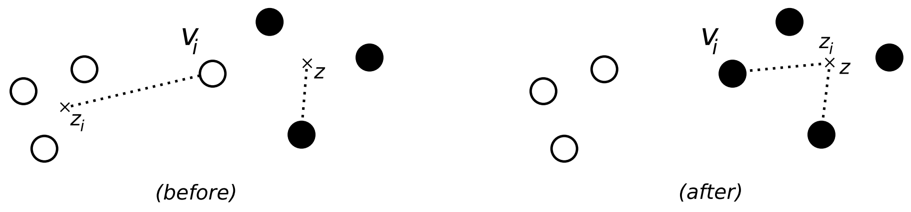
$$z_i := (1 - \beta) \cdot z_i + \beta \cdot v$$

where $0 < \beta \leq 1$ is a parameter that determines the relative influence of the other packets data vector and the old estimate z_i of packet P_i



else if the distance of v_i to the centroid z is smaller than to its own centroid z_i, i.e. $v_i - z_i > v_i - z$

then P_i is assigned to cluster of P, i.e., $c_i := c$, $z_i := z$



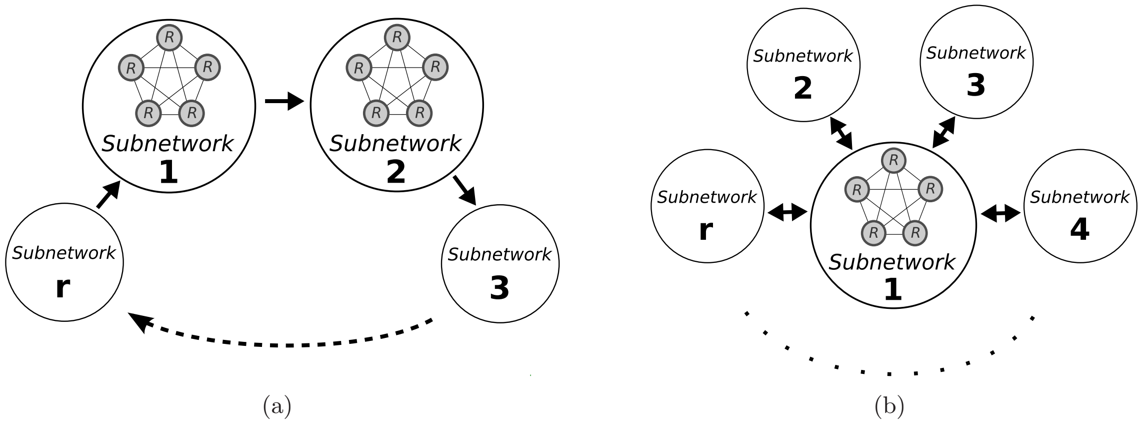


Figure 2.7: Schematic view of a ring network (a) and a star network (b); routers within a subnetwork are fully connected

(a similar situation is modelled later by connecting subnetworks that were not connected before) or if the data vector of the packets is changed over time.

Algorithm 4 Test Scenario for Ring Networks

```
Initialization

repeat
  i) Randomly choose a packet $P_i \in \mathcal{P}$ with uniform probability.
  ii) With probability $\alpha > 0$ set $f(i) := f(i)+1 \bmod r$, i.e., assign $P_i$ to the successor
      subnetwork of its actual subnetwork.
  iii) Randomly choose a router $R_j \in N_{f(i)}$ with uniform probability.
  iv) Apply DPClust in router $R_j$ to packet $P_i$.
until stopping criterion is met
```

An example of the implementation of a test scenario can be found in Algorithm 4. Basically, the algorithm describes how packets move in the ring network. Parameter $0 \leq \alpha \leq 1$ in the algorithm is called the exchange parameter and it determines the probability of packet exchanges between a subnetwork and its successor subnetwork. Note, that the algorithm can easily be modified to fit other network topologies. For fully connected networks Step ii) is: With probability α set $f(i) := f(j)$ with $j \neq i$ uniformly chosen from $1, 2, \ldots, i-1, i+1, \ldots r$, i.e., assign P_i to a random subnetwork. For the star network setting the exchange rate for the inner network N_1 to $\frac{\alpha \cdot r}{2}$ and for all other $r-1$ networks to $\frac{\alpha \cdot r}{2 \cdot (r-1)}$, ensures that the expected number of packets that change their subnetworks is the same as for a ring network with exchange rate α.

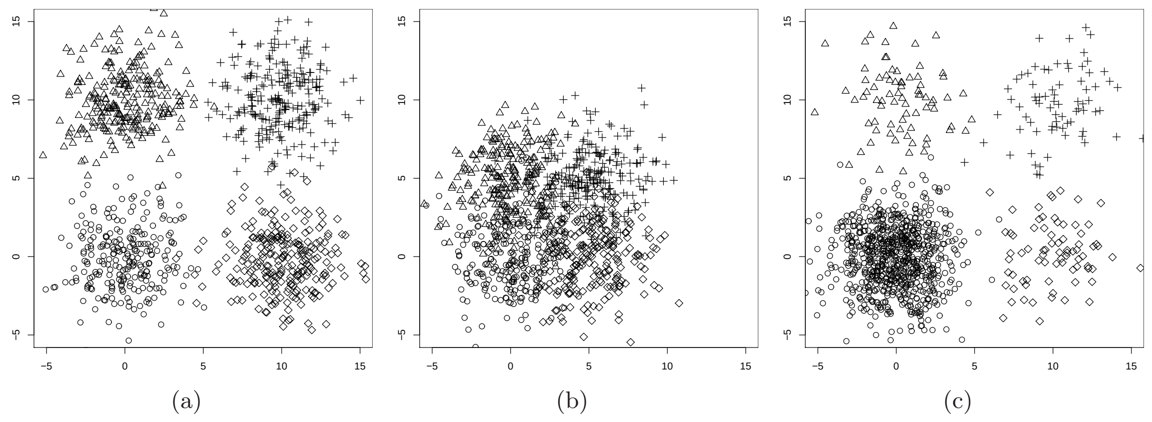


Figure 2.8: Example instances of the test datasets: *Square1* (a), *Square5* (b), *Size5* (c)

Initially each packet $P_i \in \mathcal{P}$ is assigned to a random subnetwork, i.e., $f(i)$ is chosen uniformly in $[1 : r]$, and to a random cluster, i.e., c_i is chosen uniformly from $[1 : k]$, where k is parameter of the algorithm and gives the initial number of clusters. The estimate of the centroid is set to the data vector of a random packet, i.e., $z_i := v_h$ where h is chosen uniformly from $[1 : n]$.

Problem Instances

To test DPClust we used the same type of problem instances as have been studied in several other papers on ant-based clustering (e.g., Handl et al. (2003b); Matake et al. (2007)). The problem instances determine the distribution of the data vectors of the packets. There are two types of instances, both consisting of two-dimensional data vectors from four classes. One data set is defined for investigating the influence of class overlaps and the other data set for investigating the influence of different class sizes.

For the first type of instances called *Square* each of the four data classes contains 250 data vectors. The data vectors are generated by a two-dimensional normal distribution with standard deviation 2. The centers of the normal distributions of the four classes are arranged in a square. The test data sets *Square1* to *Square7* differ by the distance between the class centers, which is $10, 9, \ldots, 4$ respectively. The second type of instances called *Sizes* is similar to *Square1* problem, but the number of data vectors of the classes differs. For problems *Sizes1* to *Sizes5* the ratio between the size of the three small classes (which are of equal size) and the size of the large class is $2, 4, \ldots, 10$ respectively. Examples of test instances of type *Square1*, *Square5* and *Size5* are depicted in Figure 2.8.

F-Measure

For the evaluation of our method we apply the F-Measure (Rijsbergen, 1979). This measure uses the real partition of the packets into classes for calculating how well the algorithm grouped the packets into clusters.

Let s_i be the number of packets that belong to class i. Let n_j be the number of packets in cluster j and let n_{ij} be the number of packets which belong to class i and are assigned to cluster j ($j \leq k$), where k is the number of clusters generated by the clustering algorithm. For each class i and cluster j the so called precision is defined as $p_{ij} = n_{ij}/n_j$ and the so called recall as $r_{ij} = n_{ij}/s_i$. The F-Measure for a clustering with respect to the given partitioning into classes is defined as

$$\text{F-Measure} = \sum_i \frac{s_i}{n} \max_{j \leq k} \left\{ \frac{2 \cdot p_{ij} \cdot r_{ij}}{p_{ij} + r_{ij}} \right\}.$$

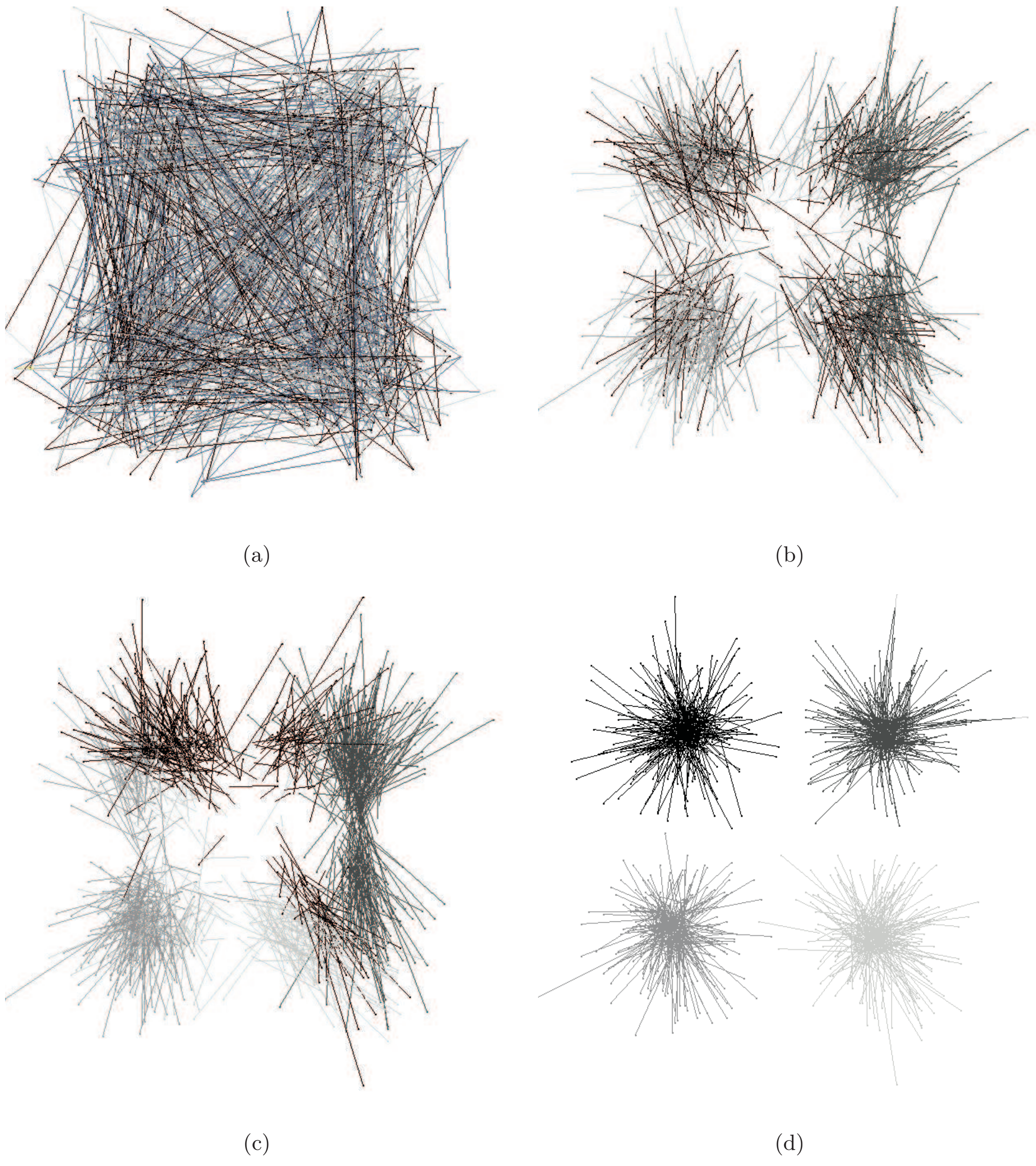
The higher the F-Measure, the better a given clustering and a perfect clustering has a F-Measure of 1.

Simulation Parameters

If not stated otherwise, for parameter β the value 0.1 was used and the standard test instance was *Square1*. All results are averaged over 50 runs. For DPClust and for the k-means algorithm the parameter $k = 4$ was used. In the following a step of an algorithm means as many iterations of the test scenario were done as packets were present in the network (for most experiments 1000 packets were used).

2.2.5 Results

To illustrate the behavior of DPClust four snap-shots from a run of DPClust in a network with only one router on an instance of *Square1* are depicted in Figure 2.9. In the figure each packet P_i is depicted by an arrow that connects its data vector v_i with the actual estimation z_i of the centroid of its cluster. The cluster number c_i is indicated by the grey value of the arrow. Figure 2.9(a) shows the random situation at the start of the run. As can be seen in Figure 2.9(b) after 40 time steps the distances of the packets data vectors to the estimated cluster centroids became smaller. Well formed clusters have not been found in this state of the simulation. It can be observed that the centroid estimates do not approximate the real centroids well, as very different centroids with the same grey value occur can be seen. This is not surprising because at this stage of a run the packets are often changing their clusters. In the later stage depicted in Figure 2.9(c) the quality of the clustering has increased and most of the clusters contain only packets from one or two



(a) (b) (c) (d)

Figure 2.9: Behavior of DPClust on *Square1* at steps 0 (a), 40 (b), 80 (c), and 120 (d); for each packet P_i an arrow connects the data vector v_i with the estimation z_i of the centroid of the cluster; the grey value of an arrow indicates the cluster number

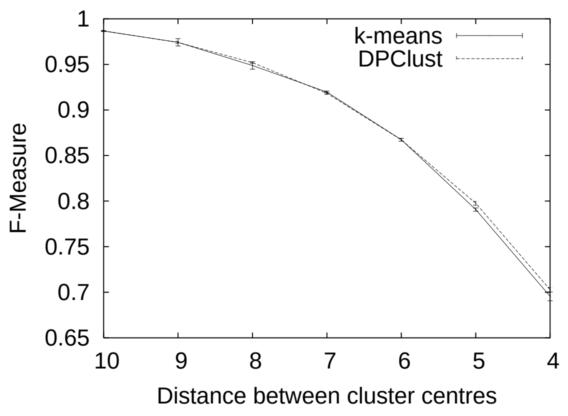
classes. Finally well formed cluster that correspond to the real classes are found (Figure 2.9(d)).

In the following DPClust is compared to the k-means algorithm. The experimental setup is the same as used before, i.e., a network with one router. It is shown later that the results of DPClust are very stable with respect to a changing number of routers. The results show that both algorithms perform equally well on the *Square* instances (see Figure 2.10(a)). Note that vertical bars in result graphs show the standard deviation. It is obvious that an increasing overlap between the classes leads to decreasing F-Measure values for the clustering of both algorithms. On the *Size* instances both algorithms have difficulties for large size differences between the clusters (see Figure 2.10(b)). But k-means performs better than DPClust for size differences that are larger than 2. It has to be mentioned here that the initialization is an important factor for DPClust. As stated before, data vectors of random packets are used for the initial centroids of both algorithms. Thus, most of these centroids point to the large class after initialization. This is a difficult situation for both algorithms, but it seems k-means can deal better with it.

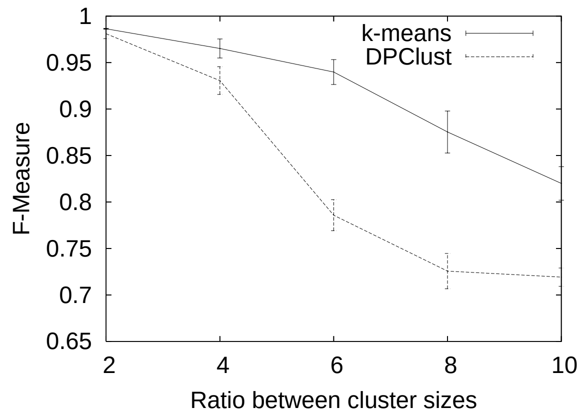
The influence of parameter β is shown in Figure 2.10(c). Recall, that β determines how strong the centroid estimation of a packet traversing a router is influenced by the data vector of the copied DPC information that are stored in the router. It can be seen in the figure that the higher the value of β, the faster the F-Measure of the system converges to the maximum. Figure 2.10(d) shows the average deviation of the estimated centroids of the packets from the true centroids of the clusters. Clearly, large values of β lead to high deviations of the estimated centroids to the true centroids. In the following we use $\beta = 0.1$ to ensure that DPClust converges in a reasonable time and the difference between the estimated and the real centroids are not too large.

Figure 2.11 shows the influence of the number of routers on the clustering behavior of DPClust. It is surprising that this influence on the quality of clustering is so small. This indicates that DPClust will work successfully in large networks with many routers working in parallel. Note, that the reason for the shifted curve of 8000 routers is that it takes some time until all routers have received at least one packet.

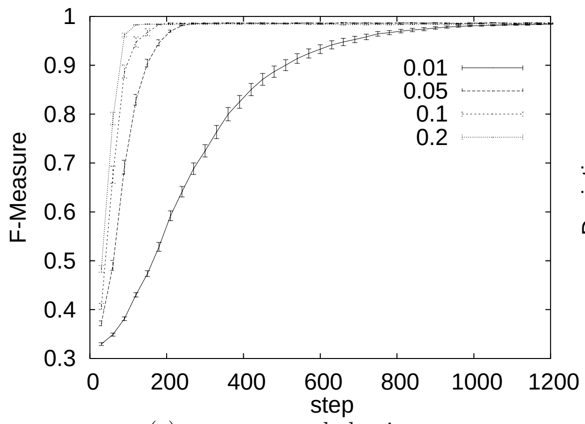
For packet clustering in networks it is interesting to investigate networks of loosely connected subnetworks. Figure 2.12(a)-(d) shows the results for DPClust on ring networks consisting of different numbers of subnetworks for varying exchange probabilities α. The results show that the algorithm has difficulties to find a good (global) clustering when the packet exchange rate between the subnetworks is very small (e.g., $\alpha \leq 0.002$ for 4 subnetworks). The more subnetworks in the ring, the smaller the F-Measure. It converges to approximately 0.65 (0.55, 0.43) for 2 (4, 16) subnetworks when using $\alpha = 0.0005$. For 32 subnetworks the F-Measure is still improving after 10000 evaluations. The reason for this behavior is the following. Soon after the first time steps in every subnetwork a good



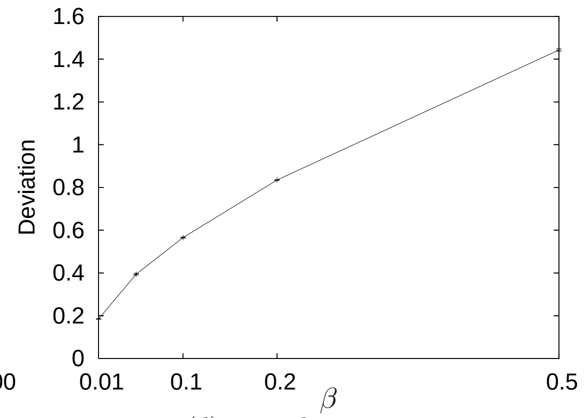
(a) *Square1* to *Square7*



(b) *Size1* to *Size5*



(c) convergence behavior



(d) mean deviation

Figure 2.10: Performance of DPClust and k-means on *Square* (a) and *Size* (b) problem instances; DPClust: convergence behavior (c) and mean deviation between estimated centroids and true centroids (d) for $\beta \in \{0.01, 0.05, 0.1, 0.2, 0.5\}$

clustering is established, i.e., most packets from the same class have the same cluster number and most packets from different classes differ in the cluster number. The cluster numbers which are assigned to the same class in the different subnetworks are the same only by chance, typically they will differ. Hence, when a packet changes its subnetwork very likely the corresponding cluster in the new subnetwork has a different number. The good message is that DPClust finds a consistent numbering of the clusters when the packet exchange rate between the subnetworks is reasonable high (for $\alpha = 0.016$ the final large value of the F-Measure is reached after about 400 steps for up to 8 subnetworks). Not surprisingly, the larger the parameter α, that is, the larger the number of exchanged packets, the faster the system finds a consistent numbering. On the other hand the larger the number of subnetworks, the longer this takes.

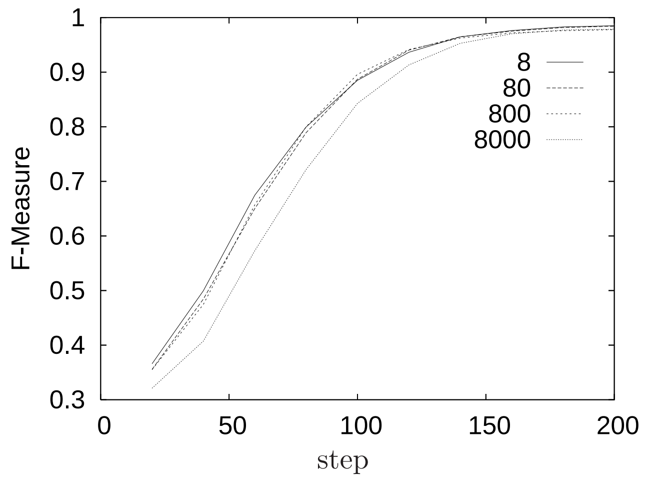
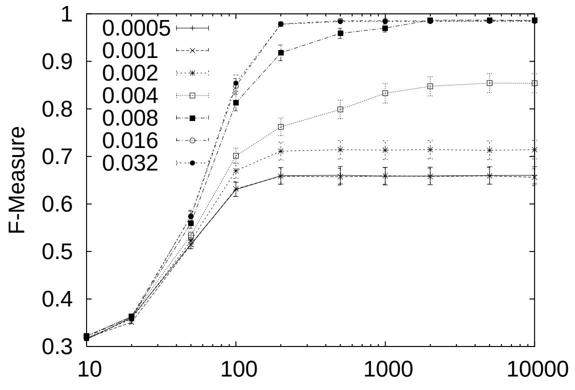


Figure 2.11: Influence of the number of routers (8, 80, 800 or 8000 routers)

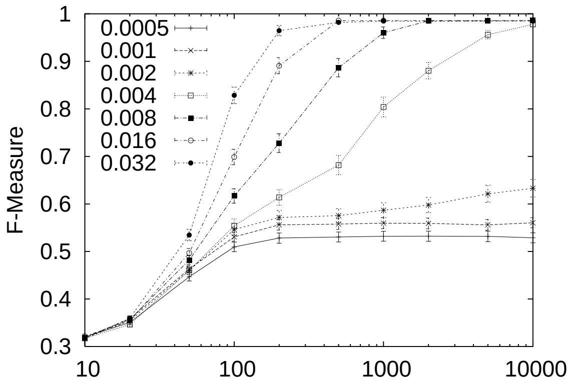
In the experiments corresponding to Figure 2.12(a-e) the total number of packets was 1000 for all tests. Thus the networks with a large number of subnetworks had less packets in each subnetwork. In order to study the influence of the number of packets we experimented with an increased number of packets (2000) in a ring network with 8 subnetworks. The results are shown in Figure 2.12(f). While the general appearance of the curves is very similar compared to the results of 1000 packets (see Figure 2.12(e)), it can be observed that it takes slightly longer for the larger number of packets and smaller exchange rates to reach the same F-Measure values.

Results regarding the influence of the connection topology between the subnetworks are given in Figure 2.13. Shown is the performance of DPClust on a fully connected and on a star network, both consisting 8 subnetworks. Comparing the depicted curves to the ones of the ring network with 8 subnetworks (Figure 2.12(e)), it can be seen that the F-Measure in the fully connected network increases faster and in the star network slower than in the ring network. The reason is that it is more unlikely in fully connected subnetworks that an inconsistent numbering in the subnetworks persists for long. On the other hand, in the star network it seems to be more difficult to establish the same numbering in all subnetworks. If two outer subnetworks differ in the numbers associated with the same class, in order to establish a consistent numbering the center subnetwork has to “decide” for one of both alternatives. This is a hard task in the presence of the relatively high packet exchange in the center subnetwork.

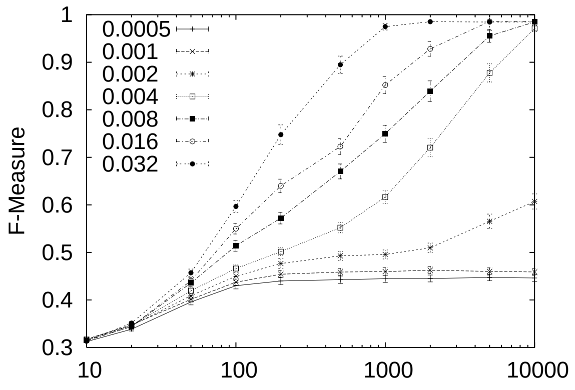
In the following the behavior of DPClust in case of dynamic network exchange rates is studied. Especially situations are considered in which formerly disconnected subnetworks become connected. In the experiments DPClust was run for 400 steps in each of 4 sub-



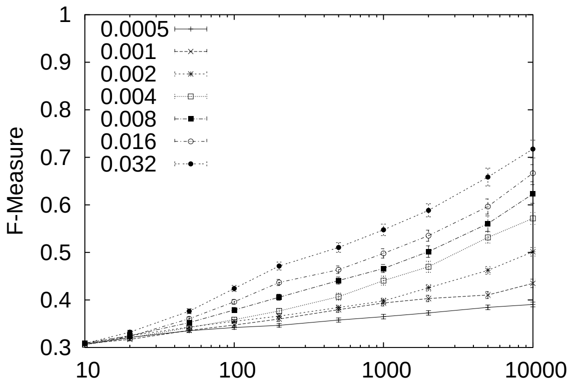
(a) 2 subnetworks



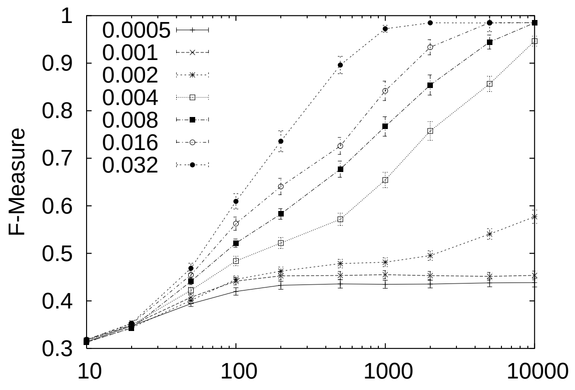
(b) 4 subnetworks



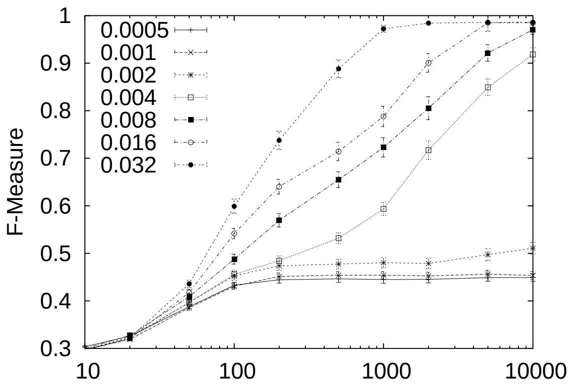
(c) 16 subnetworks



(d) 32 subnetworks



(e) 8 subnetworks



(f) 8 subnetworks, 2000 packets

Figure 2.12: Performance over time of DPClust in ring networks with 2 (a), 4 (b), 16 (c), 32 (d), and 8 (e-f) subnetworks using a total number of 1000 packets (respectively 2000 for (f)); packet exchange parameter $\alpha \in \{0.0005, \ldots, 0.032\}$; 1 router in each subnetwork

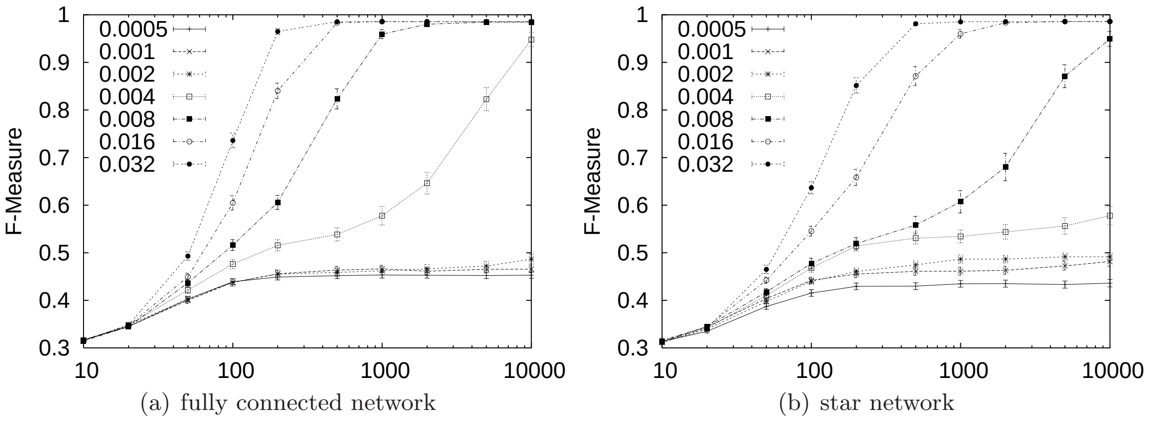
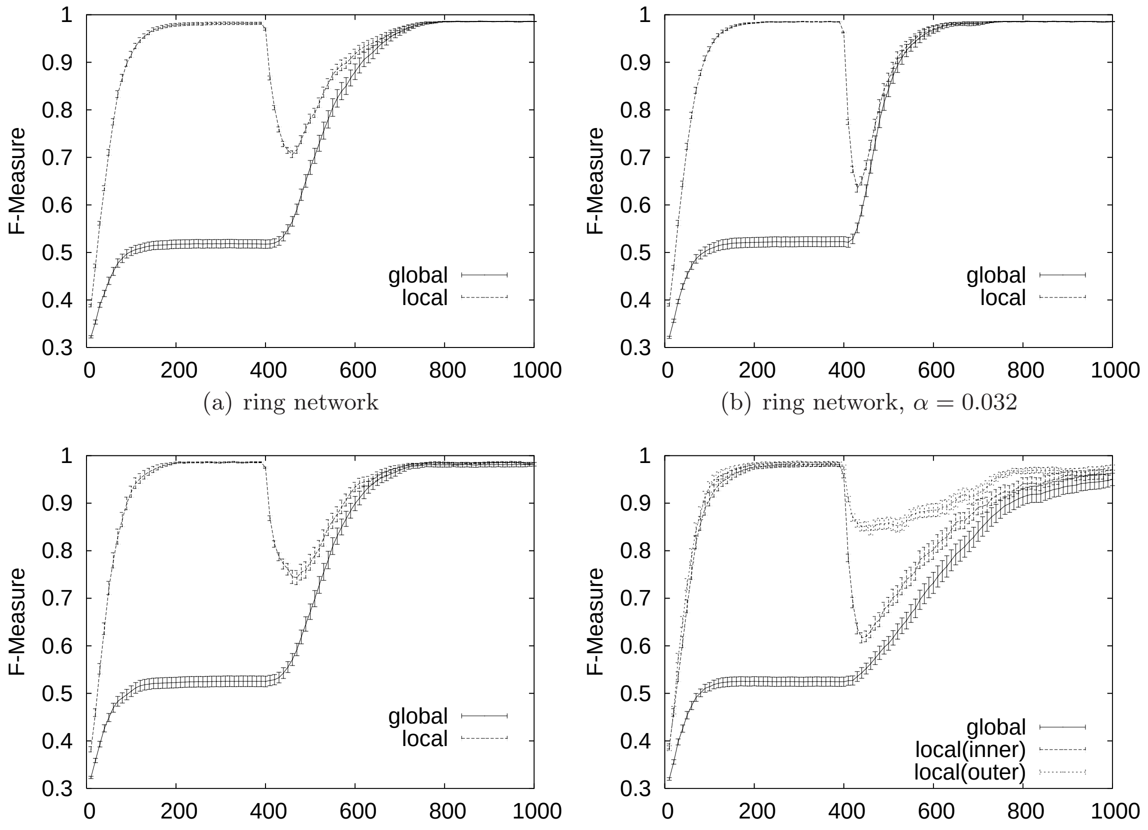


Figure 2.13: Performance of DPClust in a fully connected network (a) and a star network (b) with 8 subnetworks over time; packet exchange parameter $\alpha \in \{0.0005, \ldots, 0.032\}$, 1 router in each subnetwork

networks separately (α was set to zero). Thereafter the subnetworks were connected by setting $\alpha = 0.016$ (respectively $\alpha = 0.032$). Additionally to the global F-Measure (over all 1000 packets) also the local F-Measures in the subnetworks, i.e., with respect to the packets that are actually in the subnetworks, is calculated. In Figure 2.14(a) and (b) the global and the mean of the local F-Measures over time are depicted for a ring network with 4 subnetworks. The figure shows that the average local F-Measure has the maximal possible value 1 before the subnetworks are connected. This shows that at this point a good clustering in the subnetworks is established. The global F-Measure is small because of the differing numbering of the clusters in the subnetworks. After the subnetworks are connected at time step 400 the average local F-Measure decreases because packets from other subnetworks enter and an overall consistent numbering has to be found again. It is encouraging how fast this happens and the local F-Measure increases to the old value. The influence of the packet exchange rate between the connected subnetworks can be seen by comparing Figures 2.14(a) and (b). A higher exchange parameter $\alpha = 0.032$ leads to a stronger decrease of the local F-Measures but also to a faster convergence to the maximal F-Measure after the connection of the subnetworks.

Figure 2.14(c-d) shows the same scenario but using star and fully connected networks (the subnetworks were connected with $\alpha = 0.016$ as well). The curves for the fully connected network are quite similar to the corresponding curves for the ring network. Again, the fully connected network can converge a little faster than a ring network. However for the star network the global F-Measure increases slower after opening the connection. This is in accordance with the observations that have been made in the static scenarios. It is interesting that there is a clear difference between the inner subnetwork and the outer



(a) ring network

(b) ring network, $\alpha = 0.032$

(c) fully connected network

(d) star network

Figure 2.14: Average local and global F-Measure in a ring (a-b), a fully connected (c) and in a star network (d) with 4 subnetworks: no packet exchange ($\alpha = 0.0$) was done for the first 400 steps, then α was set to 0.016 (0.032 for (b)); for the star network the local F-measure is shown separately for the inner subnetwork and an outer subnetwork

subnetworks of the star network. In the inner subnetwork the local F-Measure decreases stronger than the local F-Measures in the ring or fully connected networks, whereas the decrease is much less for the outer subnetworks. This is because the inner network receives more packets from other subnetworks than the outers do, as the exchange rate is $(\alpha \cdot r)/2 = 0.032$ for the inner and $(\alpha \cdot r)/(2r-2) = 0.012$ for each outer subnetwork.

The F-Measures over time for a single typical run of DPClust in a star network with 4 subnetworks are depicted in Figure 2.15. It is evident from the figure that the local F-Measure in the inner subnetwork decreases much stronger after opening the connection than the local F-Measures in the outer subnetworks. As stated before, this is because the inner subnetwork receives three times more packets from outside than each outer subnetwork. Moreover it can be seen that in one of the outer subnetworks the local F-measure remains significantly higher. The reason is that for this run all three outer

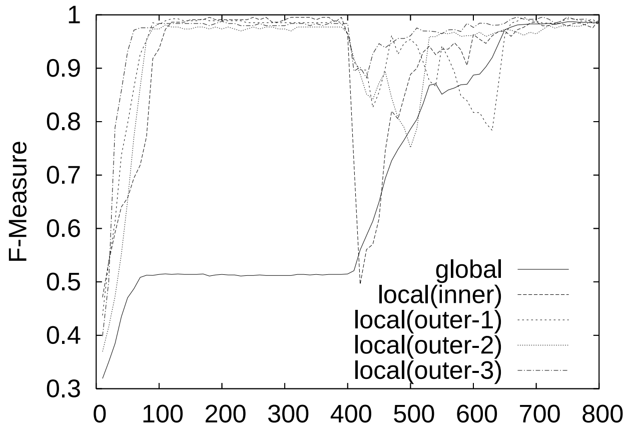


Figure 2.15: Single run of DPClust in a star network with 4 subnetworks: shown are the local F-Measures for the inner subnetwork and the three outer subnetworks: no packet exchange ($\alpha = 0.0$) was done for the first 400 steps, then α was set to 0.016

subnetworks evolved different cluster numberings. The subnetwork with the higher local F-measure was the "winner" in this run with respect to the renumbering of the local clustering, i.e., its numbering was finally adopted by the other subnetworks. Clearly it can also happen that two or three of the subnetworks use the same numbering (or partially equal numberings). In this case it can typically be observed that only for one or none of the outer subnetworks the local F-Measure decreases strongly after the connection.

2.2.6 Dynamic Problem Instances

In this subsection we study dynamic clustering instances, i.e., we assume the packets data vectors v_i are changed over time (e.g., by the application processes). It will be shown, that DPClust does not perform very well on dynamic problem instances. Therefore, additionally to DPClust we introduce two modified variants, called d-DPClust$_{cz}$ and d-DPClust$_{zc}$. These variants do store the centroid estimations in the routers instead of in the packets.

d-DPClust$_{cz}$

For algorithm d-DPClust$_{cz}$ each packet $P_i = (v_i, c_i)$ consists of a data vector v_i and a cluster number c_i. Each router r stores a vector of estimated centroids $\mathcal{Z}_r = (z_r^1, \ldots, z_r^{|\mathcal{C}|})$. For a packet P_i that arrives at router r the cluster number c_i is determined by using the distances of its data vector v_i to the estimated centroids z_r^j, $j = 1, \ldots, |\mathcal{C}|$ that are stored in the router. A packet is assigned to the cluster for which this distance is minimal, i.e., $c_i = \text{argmin}_j ||v_i - z_r^j||$. Thereafter the router's centroid estimation for cluster c_i is modified according to $z_r^{c_i} = (1-\beta) \cdot z_r^{c_i} + \beta \cdot v_i$. The parameter β has the same use in d-DPClust$_{cz}$ as in DPClust.

Note, that in algorithm d-DPClust$_{cz}$ the centroid estimations of two routers r_1 and r_2 may have a different order (in the sense that $z_{r_1}^i$ corresponds to the centroid estimation $z_{r_2}^j$ with $i \neq j$). Therefore, the routers $r = 2, \ldots, |R|$ iteratively reorder their centroids after every $e \geq 1$ time steps. The reordering is done according to a permutation π for which $\sum_{i=1}^{|\mathcal{C}|} ||z_r^{\pi(i)} - z_{r-1}^i||$ gets minimal. To be able to do this exchange step the routers must communicate directly to exchange the needed data.

d-DPClust$_{zc}$

The d-DPClust$_{zc}$ algorithm works similar to d-DPClust$_{cz}$. The difference is that in d-DPClust$_{zc}$ the router centroid estimation is modified *before* the packet's cluster is determined. That is, first the modification of the centroid estimation is applied to $z_r^{c_i}$. Afterwards the cluster number for P_i is determined similarly to d-DPClust$_{cz}$, i.e., the cluster is determined by $c_i = \text{argmin}_j ||v_i - z_r^j||$. A router exchange step as in d-DPClust$_{cz}$ is not needed.

2.2.7 Experiments

In the following the dynamic problem instances and three new cluster validity measures are introduced. The reason for using other measures is that in the investigated problem instances a large overlap of data vectors from different classes can happen. In case of an overlap the data points of the classes are indistinguishable for any algorithm. Therefore, it makes no sense to use clustering measures like the F-Measure which take the real partition of the packets into classes into account. The following introduced measures do not need any knowledge about the classes of the packets.

Dynamic Problem Instances

The first problem instance called T_1 is a dynamic version of the *Square1* data set (see Section 2.2.4). At the begin of a simulation an instance of *Square1* is generated. Recall, the generated data vectors are from four classes with the four center points $(0,0)$, $(0,10)$, $(10,0)$, and $(10,10)$ and 250 data vectors in each class. For every class $j \in \{1,2,3,4\}$ an uniformly distributed random moving direction $\Delta v_j = (\Delta v_j^1, \Delta v_j^2) \in [-1,1]^2$ is chosen. After every time step of the simulation all data vectors v_i are moved according to $v_i = v_i + \Delta v_{c(i)} \cdot v$ where $c(i) \in \{1,2,3,4\}$ denotes the class of packet P_i and parameter v is used to adjust the strength of the dynamics, as it relates to the moving velocity of the classes. If the center point of a class would leave the predefined cluster area $A = [-10,20]^2$ in a dimension k, then the sign of Δv_j^k is flipped, i.e., the moving direction of the class is

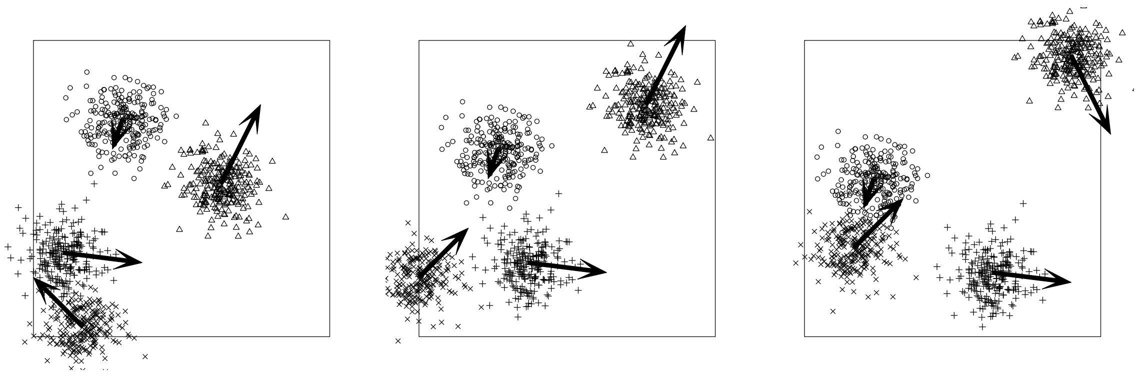


Figure 2.16: Problem instance T_1 dynamic case ($v = 0.1$); 100 time steps between the figures; arrows indicate direction and velocity of the classes

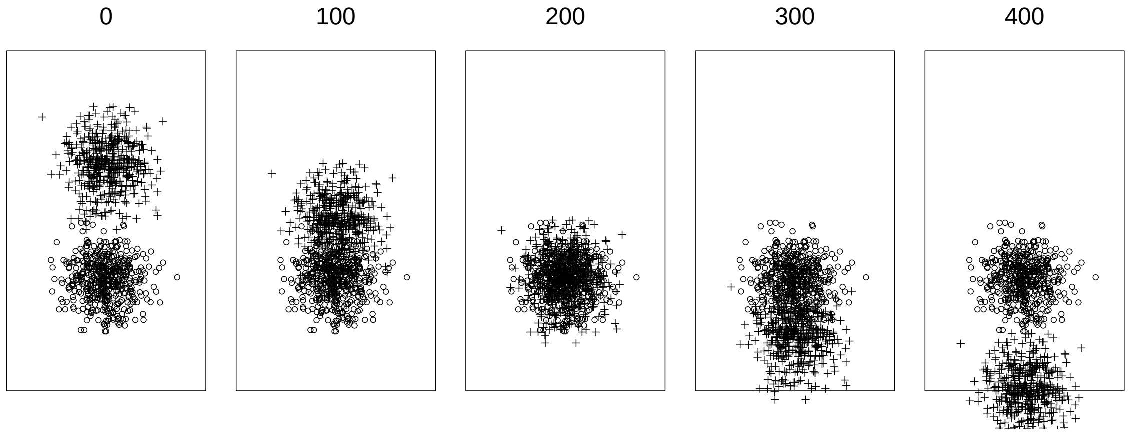


Figure 2.17: Problem instance T_2 at time steps $0, 100, \ldots, 400$; velocity parameter $v = 0.05$; the framed area gives the border for reflecting the moving class

reflected at the border of the area. In Figure 2.16 an example for the problem instance of type T_1 is given for different time steps.

Problem instance T_2 consists of two classes with center points $(0, 0)$, $(0, 10)$ and 500 data vectors in each class. The class with center point $(0, 0)$ does not move (i.e., $\Delta v_1 = (0, 0)$), and the second class initially moves to the bottom along the vertical axis ($\Delta v_2 = (0, -1)$). The cluster area for instance T_2 is $A = [-10, 10] \times [-10, 20]$.

Silhouette Coefficient

Recall, a clustering $\mathcal{C} = \{C_1, C_2, \ldots\}$ is a partition of all packets $\mathcal{P}$. Since the data vectors v_i are parts of the packets P_i the clustering $\mathcal{C}$ implies a partition of the set of data vectors as well. To keep things simple, in the following we directly use the clustering $\mathcal{C}$ as a clustering of the data vectors v_i, that is, we write $v_i \in C_j$ in case $P_i = (v_i, c_i) \in C_j$.

In Kaufman and Rousseeuw (1990) the so called *Silhouette Coefficient* is defined as follows. Let $d(v, C)$ be the distance of the data vector v to the geometric centroid of the data vectors $v_i \in C$. Let $\gamma_1(v, \mathcal{C})$ be the number of the cluster which is the nearest cluster to data vector v, i.e., $\gamma_1(v, \mathcal{C}) := \arg\min_j d(v, C_j)$ and let $\gamma_2(v, \mathcal{C})$ be the cluster which is second nearest cluster to data vector v, i.e., $\gamma_2(v, \mathcal{C}) := \gamma_1(v, \mathcal{C} \setminus C_{\gamma_1(v,\mathcal{C})})$. The Silhouette Coefficient s_i for data vector v_i is defined as the normalized difference:

$$s_i := \frac{d(v_i, C_{\gamma_2(v_i,\mathcal{C})}) - d(v_i, C_{\gamma_1(v_i,\mathcal{C})})}{\max\{d(v_i, C_{\gamma_1(v_i,\mathcal{C})}), d(v_i, C_{\gamma_2(v_i,\mathcal{C})})\}}$$

The Silhouette Coefficient SC is defined as the average value over all s_i:

$$SC := \frac{1}{n}\sum_{i=1}^{n} s_i.$$

Empirical studies show that $SI > 0.7$ indicates an excellent separation between the clusters, a value between 0.5 and 0.7 indicates a clear assignment of data points to clusters, values between 0.25 and 0.5 indicate that there are many data points that cannot be clearly assigned, and $SI < 0.25$ indicates that it is practically impossible to find significant cluster centers (Kaufman and Rousseeuw, 1990). For dynamic test instances $SC_\varnothing$ denotes the average Silhouette Coefficient SC over all measured time steps.

Dunn Index

The *Dunn index* measures the minimal ratio between cluster diameter and inter-cluster distance for a given clustering $\mathcal{C}$. Let $d(C_i, C_j)$ be the average distance of all pairs of elements in C_i and C_j, and let $diam(C)$ be the maximal distance between two elements of cluster C. Then the Dunn index DI can be computed as

$$DI = \frac{\min_{\{C_i, C_j \in \mathcal{C}\}} d(C_i, C_j)}{\max_{\{C \in \mathcal{C}\}} diam(C)}.$$

A low Dunn index indicates a fuzzy clustering, whereas a value close to 1 indicates a near-crisp clustering. The Dunn index tries to identify well separated and compact clusters. $DI_\varnothing$ denotes the average DI value over all measured time steps.

Sum of Squares

Let $\hat{v}_l$ be the geometric centroid of cluster C_l. The *Sum of Squares* criterion is defined as

$$SS = \frac{1}{|\mathcal{C}|}\sum_{l=1\ldots|\mathcal{C}|}\left(\sum_{v_i \in C_l}\frac{||v_i - \hat{v}_l||^2}{|C_l|}\right)$$

and measures the compactness of a clustering. The smaller the value SS, the more compact the clustering. $SS_\varnothing$ denotes the average Sum of Squares SS over all measured time steps. In contrast to $SC_\varnothing$ and $DI_\varnothing$, which have to be maximized, the average Sum of Squares $SS_\varnothing$ has to be minimized for a good clustering.

2.2.8 Results

The experimental results of the clustering algorithms k-means, DPClust, d-DPClust$_{cz}$ and d-DPClust$_{zc}$ on static and dynamic problem instances are presented in the following. If not stated otherwise parameter $\beta = 0.1$ is used for all algorithms and $e = 10$ for d-DPClust$_{cz}$. The number of iterations is 20 000 per test run. All tests using dynamic data sets were started with the same initial random seed for the different algorithms, to ensure that the problem instances are the same in every simulation step. As for DPClust the initial centroid estimations of the routers in d-DPClust$_{cz}$ and d-DPClust$_{zc}$ are the data vectors from randomly chosen packets. Results are averaged over 50 runs.

As a reference again the k-means algorithm is chosen. To solve the dynamic problem instances in every simulation step k-means is performed until it is converged. The centroids that are finally found in a simulation step are used to initialize the centroid estimations of the subsequent k-means run. Note again, that k-means is a centralized algorithm with global knowledge of the data vectors.

Static Problem Instances

To evaluate the new algorithms and measures we first present results for static problem instances. In Figure 2.18 the Silhouette Coefficient SC (calculated at the end of the simulations) of the four algorithms on problem instances $Size_s$, $s \in \{2, 4, 6, 8, 10\}$ is shown for a network with a single router. Note, that d-DPClust$_{cz}$ outperforms k-means. For large values of k the initial center points are mostly chosen from large class of data vectors. Thus, k-means starts very likely with a partition of the large class into several smaller clusters and combines some of the smaller classes into one cluster. In contrast to k-means, algorithm d-DPClust$_{cz}$ has the ability to escape from this situation.

In Figure 2.19 the Silhouette Coefficient over time for the algorithms DPClust, d-DPClust$_{cz}$, and d-DPClust$_{zc}$ in networks with 1, 10, 100, and 1000 routers is given. Although d-DPClust$_{cz}$ leads to very good results on the static problem instances, its convergence speed gets worse if the ratio between number of routers and packets gets too large. Hence, d-DPClust$_{cz}$ applicability in dynamic situations may be bad for a large number of routers. It should be noted that the reordering step of the routers can be time consuming if the number of clusters gets too large.

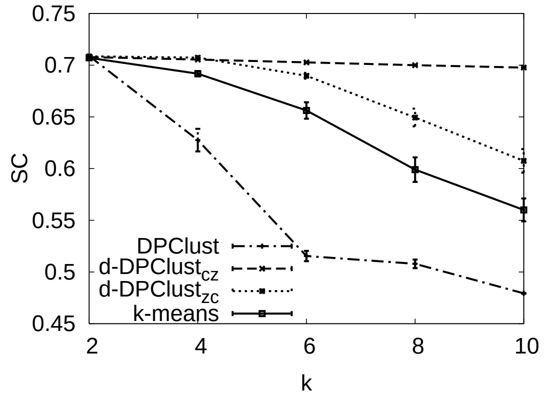


Figure 2.18: Silhouette coefficient for k-means, DPClust, d-DPClust$_{cz}$, and d-DPClust$_{zc}$ on $Size_s$, $s \in \{2, 4, 6, 8, 10\}$; one router

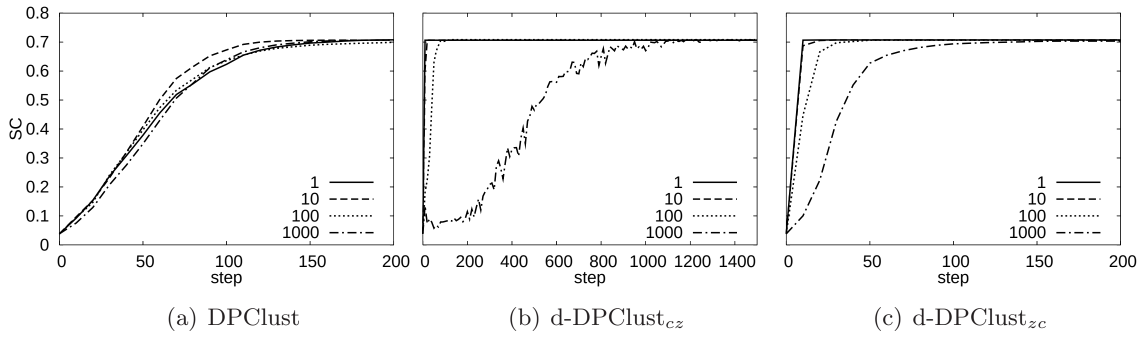
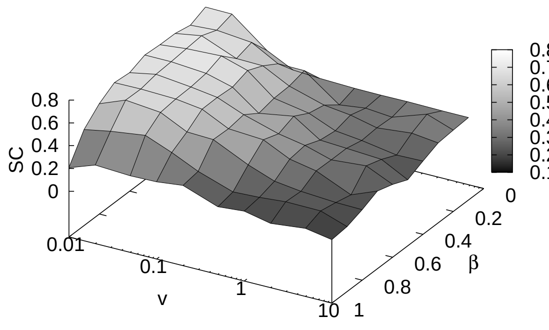


Figure 2.19: Convergence Speed on problem instance $Size_1$; different lines correspond to different number of routers; shown is the silhouette coefficient SC at different time steps

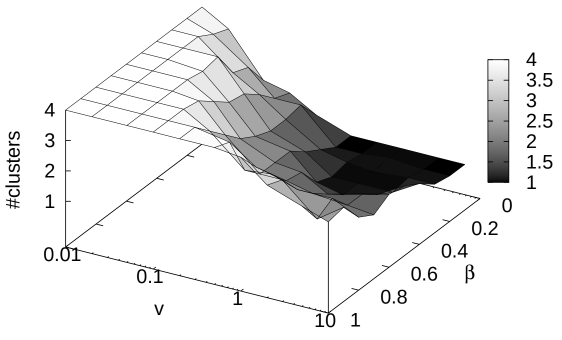
Dynamic Problem Instances

In Figure 2.20 the results of DPClust on the dynamic instance T_1 for different class velocities v and varying parameter β are given. The Silhouette Coefficient shown in Figure 2.20 attests that DPClust performs poorly on this problem instance for class velocities $v \gtrapprox 0.2$. Moreover, due to the moving data vectors it can happen that clusters get lost, i.e., that there are no more packets which belong to a certain cluster. In algorithm DPClust, in contrast to the other algorithms, this is definitely irreversible. Whereas to a small extent this effect can be reduced by adapting β, the loss of clusters sometimes even happens for very low dynamics $v \approx 0.01$. For high dynamics often only one cluster survives (see Figure 2.20(b)). Therefore we exclude DPClust from further investigations on dynamic instances.

In the following we study the performance of d-DPClust$_{cz}$ and d-DPClust$_{zc}$ on instances T_1 and T_2. In Figure 2.21 the values of the average Silhouette Coefficient $SC_\varnothing$, the average Dunn index $DI_\varnothing$, and the average Sum of Squares $SS_\varnothing$ are depicted for the algorithms d-



(a) Silhouette Coefficient



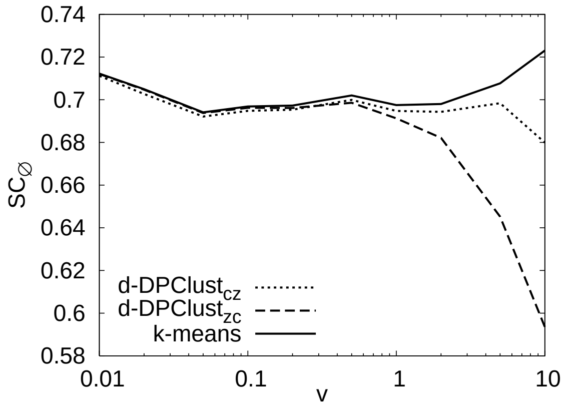
(b) Average number of clusters

Figure 2.20: Silhouette Coefficient (a) and the average number of clusters (b) after 20 000 steps for DPClust on problem instance T_1

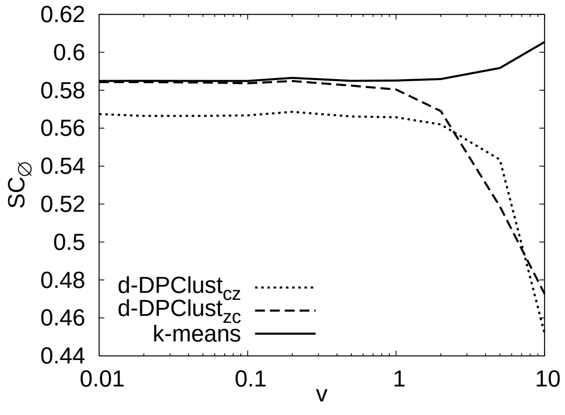
DPClust$_{cz}$, d-DPClust$_{zc}$, and (as reference) for k-means. The extent of dynamic changes varied with $v \in \{0.01, 0.02, 0.05, 0.1, \ldots, 10\}$. Note, that $v \gtrsim 5$ is a very high cluster velocity. With respect to all three validity measures both variants of d-DPClust show a very good clustering behavior for $v \lesssim 1$ on both problem instances. They perform only slightly worse than the centralized k-means algorithm. For highly dynamic situations ($v \gtrsim 1$) the performance of d-DPClust$_{cz}$ is better compared to d-DPClust$_{zc}$.

However on problem instance T_2 d-DPClust$_{cz}$ performs slightly worse than d-DPClust$_{zc}$ for $v < 1$. The reason for this is illustrated in Figure 2.22. This figure depicts the Silhouette Coefficient in the first 400 steps. In this time frame one class crosses the other class completely (see Figure 2.17). As can be observed in the situation of a strong overlap of the classes d-DPClust$_{zc}$ can maintain a good clustering as the SC is never worse than 0.45. Algorithm d-DPClust$_{cz}$ on the other hand has quite some trouble and the SC fluctuates strongly, but stabilizes again after the classes leave each other alone again. Note, the chance for such a strong overlap in instance T_1 is much smaller than in instance T_2, which is the reason why d-DPClust$_{cz}$ performs better on T_1.

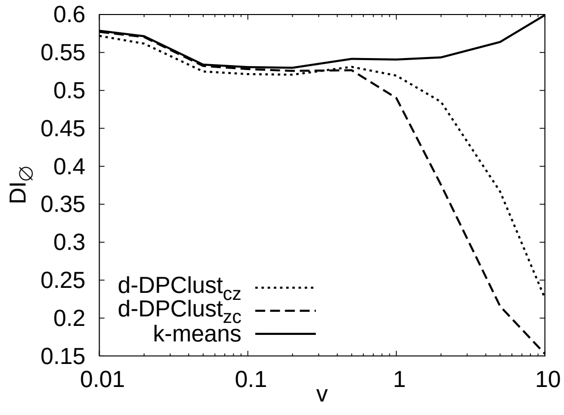
Int the following the d-DPClust variants are investigated on T_1 in networks with more than one router. In Figure 2.23 the performance in terms of $SC_{\varnothing}$ for d-DPClust$_{cz}$ using 10, 100, and 1000 routers is given for varying class velocities v. The result depicted in Figure 2.23(a) emphasize that in networks with 10 routers strong dynamics lead to a bad performance of d-DPClust$_{cz}$. How strong the cluster velocity v influences the performance depends on the frequency of router exchange steps: the less frequent, the worse the results. Regarding the results for higher number of routers two observations can be stated. First, when using many routers the influence of the frequency of router exchange steps becomes less. That is, even when aligning the numbering of the clusters every time step, this does not increase the performance compared to less frequent exchange steps. Second, the



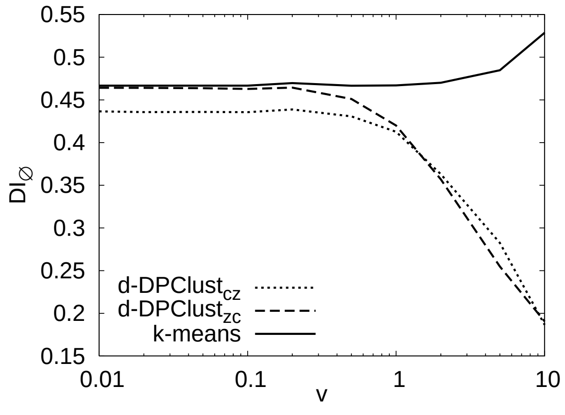
(a) average Silhouette Coefficient on T_1



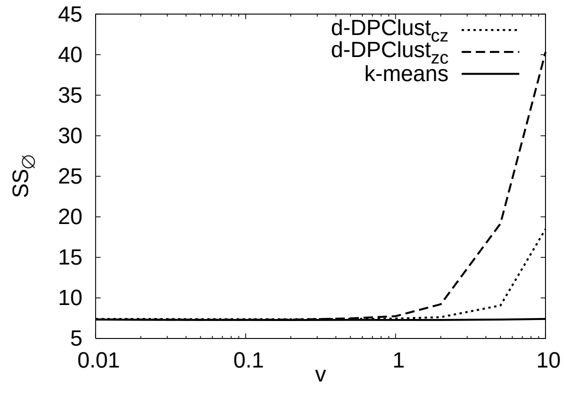
(b) average Silhouette Coefficient on T_2



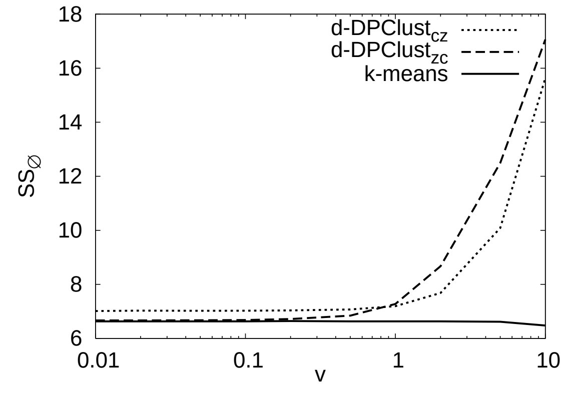
(c) average Dunn index on T_1



(d) average Dunn index on T_2



(e) average Sum of Squares on T_1



(f) average Sum of Squares on T_2

Figure 2.21: Performance of d-DPClust$_{zc}$, d-DPClust$_{cz}$, and k-means: Given are average Silhouette Coefficient $SC_{\varnothing}$ (a-b), average Dunn index $DI_{\varnothing}$ (c-d) and average Sum of Squares $SS_{\varnothing}$ (e-f) for problem instance T_1 (left) and T_2 (right); dynamic change $v \in \{0.01, 0.02, 0.05, 0.1, \ldots, 10\}$

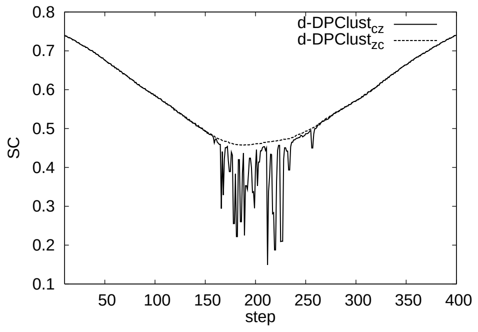


Figure 2.22: Silhouette coefficient for d-DPClust$_{cz}$ and d-DPClust$_{zc}$ given over time steps $t = 10, \dots, 400$ when one class crosses the other class in problem instance T_2; v=0.05

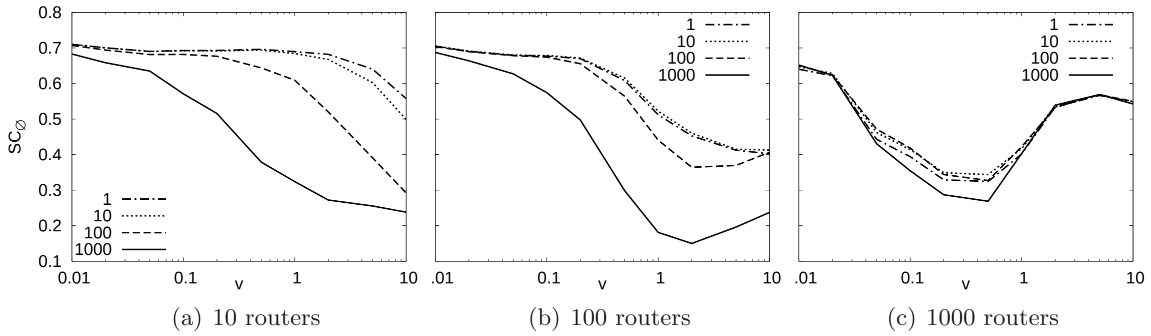


Figure 2.23: Performance on problem instance T_1 with respect to $SC_\varnothing$ of d-DPClust$_{cz}$ with different number of routers each for different values of parameter $e \in \{1, 10, 100, 1000\}$

performance of d-DPClust$_{cz}$ becomes better again for highly dynamic situations. The increasing values of $SC_\varnothing$ for 1000 routers with $v \gtrsim 2$ can be explained in the following way. Figure 2.24(a) shows the average movement of all estimated cluster centroids of all routers until simulation step T, defined as

$$\frac{1}{T \cdot |\mathcal{C}| \cdot |\mathcal{R}|} \cdot \sum_{t=1}^{T} \sum_{R_i \in \mathcal{R}, C_j \in \mathcal{C}} ||\hat{v}_{ij}^t - \hat{v}_{ij}^{t-1}||,$$

where $\hat{v}_{ij}^t$ is the estimated cluster centroid of cluster C_j in router R_i at time step t. It can be observed that for highly dynamic situations the estimated cluster centroids in networks with few routers do change very strong. Whereas, in networks with many routers there is nearly centroid movement can be stated, i.e., the cluster centroids remain almost the same. Hence, applied on many routers the algorithm does not follow the moving classes. Nevertheless, data vectors close to a non-moving cluster centroid are assigned to

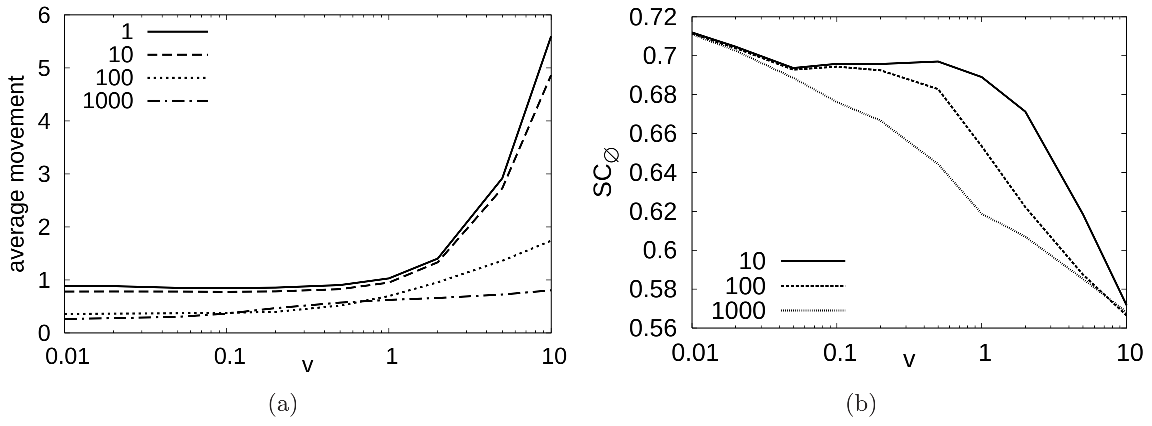


Figure 2.24: d-DPClust$_{cz}$: average movement of the estimated cluster centroids of the routers for different values of v and different number of routers (1, 10, 100, 1000) (a); d-DPClust$_{zc}$: Silhouette Coefficient $SC_{\varnothing}$ for 10, 100, and 1000 routers (b)

the cluster. Although there is no tracking behavior, this leads to an increasing value for $SC_{\varnothing}$.

The results for d-DPClust$_{zc}$ given in Figure 2.24(b) emphasize that for very low and very high dynamics d-DPClust$_{zc}$ shows the same performance regardless of the number of routers in the network. But for intermediate values ($v \approx 1$) systems with less routers perform better. The results of d-DPClust$_{zc}$ are very promising since in all cases, even with strong dynamics like $v = 10$, it holds $SC_{\varnothing} > 0.55$.

2.3 Summary

In this chapter we investigated two typical systems showing emergent behavior. First we presented a study of emergent sorting effects exhibited by a certain type of networks of router agents. In addition to the original proposal of such networks, we examined variants and extensions, including a pheromone-based agent behavior. The experimental results show that the sorting performance strongly depends on the shape and the size of the network, the number different object types, and the agent behavior.

Furthermore, we dealt with the problem of clustering a set of packets that are send around in a network of routers. We proposed an algorithm called DPClust which can be executed by the routers without direct information transfer and with minimal use of computational and routing resources. It was shown that DPClust has similar performance as k-means on some standard benchmark problems while it is worse on others. However, our main focus was to investigate whether DPClust is robust and successful for networks of different topologies. The results are promising and show that DPClust performs well, even in loosely connected networks. Additionally we investigated DPClust and two introduced variants on dynamic problem instances. It was shown that DPClust performs poorly in dynamic situations since it can happen that the algorithm looses clusters. On the other hand, the new DPClust variants called d-DPClust$_{cz}$ and d-DPClust$_{zc}$ can cope in general well with dynamic problems. While d-DPClust$_{cz}$ mostly has a better average performance, d-DPClust$_{zc}$ can better handle situations with large numbers of routers.

In both investigated systems, the Emergent Sorting Networks and the Decentralized Packet Clustering, large populations of interacting elements without central control and hence based only on local rules generate macroscopic behaviors not existent on the element-levels. As we have shown these emergent behaviors scale well with the number of entities (agents, packets, routers), a typical property of self-organizing systems. For the DPClust algorithms we have seen that the emergent behavior is robust in dynamic situations, like in case of a sudden connection of former unconnected networks or the dynamic change of the data vectors. Both investigated systems are examples of the technical utilization of emergence, as the emergence of order, showing as sorted respectively clustered objects, is the main purpose of the systems.

3 Controlled Emergence

Emergence in complex technical systems is an ambivalent property. On the one hand, as seen in the last chapter, emergent effects can be an intended goal of a system design. Principles of emergent behavior of natural systems have been successfully applied in many cases to increase the capabilities of technical systems or to design algorithms with improved performance. Most researchers have considered mainly these positive aspects of emergent behavior. On the other hand recently concerns came up that self-organized computing systems which consist of many autonomous components might show an emergent behavior that is neither wanted nor has it been intended or foreseen to occur when the systems were designed. In this a new approach for controlling emergent effects called *swarm controlled emergence* is introduced and its application on a nature inspired test system is shown. In the second part of the chapter another way to control emergent behavior is investigated. Namely the possibility to control a system of ant-like moving agents by changing the environment instead of changing the behavior of the agents.

3.1 Negative Emergence

In self-organizing technical systems emergent behaviors can occur that has not been intended and that has negative consequences for the system. Such negative emergent behavior with unwanted effects can be observed in everyday life and somehow we must deal with them. The following examples show that this is often not a trivial task.

The network of neurons in the human brain forms a dynamic system that shows nonlinear, complex and chaotic interactions and activities. Within an epileptic seizure these neuronal networks change from their normally complex, chaotic activity to a synchronized state in which all the neurons are doing the same thing at the same time (Ohayon et al., 2004). One way to treat epilepsy is to implant a medical device, called brain pacemaker, into the brain to send electrical signals into the tissue. Even if the exact mechanism of action of this deep brain stimulation is not known, it helps to control or even prevent the emergence of the abnormal, synchronized firing of neurons.

In 1850 more than 700 French soldiers marched lock-step over the rope bridge of Angers. 226 soldiers died after the bridge began to vibrate and collapsed. This tragedy is an example of a resonance catastrophe, a situation where a building is destroyed by vibrations.

The vibrations emerge through the accumulated energy in the system due to periodic stimulation in the eigenfrequency of the system. Today, beside the use of vibration absorbers, it is not allowed to march lock-step over a bridge in order to avoid such a catastrophe again.

In technical systems that are created as self-organizing collections of self-interested agents, as the result of many locally reasonable agent decisions, highly dysfunctional dynamics, called social pathologies, can emerge (Jensen and Lesser, 2002). Social pathologies occur when improvements in the local performance of the agents do not improve the system performance. This can lead to problems like inefficient resource allocation (Hardin, 1968), suboptimal collective decision processes (Klein et al., 2003) and several more. Klein et al. (2005) describes an emergent oscillation effect in the use of a resource in peer-to-peer systems, induced by a delayed view of the peers on the resource's queue. To handle this problem, the authors suggest the spreading of misinformation, which was found to dampen oscillations and improve system performance.

From an engineers points of view, to implant electrodes into the brain without exactly knowing the mechanism of action, to simply prohibit marching lock-step over bridges and to spread misinformation in order to improve a system might sound like strange problem solutions. But these examples show that to cope with the question how (negative) emergent behavior can be reduced or prevented, or more general, how to "control" self-organization and emergence, new kinds of thinking may be required.

An obvious question to ask is, what makes it hard to apply standard control mechanisms to complex self-organizing technical systems ? There are multiple reasons for that. First, the parts of self-organizing technical systems must exhibit a sufficient degree of freedom to be able to self-organize and to generate, if intended, useful emergent properties on a higher system level. This means in fact, that all possible system states can not be foreseen in advance, which is usually required for standard approaches for designing robust systems. Second, due to the distributed character and the complex, often non-linear, relations between the parts of the systems, it is hard or even impossible to find single points to impose reasonable control over the whole system.

Unwanted emergent behavior in technical systems, also called *negative emergence* (Mnif and Müller-Schloer, 2006; Müller-Schloer and Sick, 2006) or *emergent misbehavior* (Mogul, 2006), generates the need for new approaches to deal with it. Especially when designing systems that solve safety-critical tasks, methods must be developed which leave sufficient degrees of freedom for self-organization while keeping control over resulting emergent effects to avoid negative emergence.

A way to deal with unforeseen system states is to create systems which rely on feedback loops. A feedback loop can guide the system by comparing the actual state of the system with given high-level objectives/goals. In Section 1.3 we introduced self-adaptive systems, the autonomic element, and the observer/controller architecture as possible design methodologies for such systems. Two examples for preventing negative emergence using feedback loops implemented by using the observer/controller architecture are given in the following.

Preventing Bunching in Lift Systems

Using certain (simple) lift group control systems, under heavy traffic load conditions a phenomenon called *bunching* can be observed. When the bunching effect occurs the lifts tend to synchronise and serve the floors in form of a wave. The lifts behave like a huge, single lift with the capacity equal to the sum of the individual lifts. Bunching itself cannot be expected beforehand nor its occurrence be predicted from the system description. Since it affects the performance of the lift system negatively it is considered as an example for negative emergence (Mnif and Müller-Schloer, 2006).

In Ribock et al. (2008) the generic observer/controller architecture is applied to a lift group traffic control system to evaluate its applicability for preventing bunching. The SuOC (system under observation and control) is formed by the lifts and the passengers waiting at the floors. The only observed parameters are the lift positions and the travelling directions of the lifts. For controlling the lifts two simple methods were implemented that modify the lift's view on the environment and thus affect the local behavior of the lifts. It was shown that a bunching effect can be prevented autonomously by such a system.

Preventing Cannibalistic Behavior in Chicken Farms

If chickens perceive a wounded chicken they chase this chicken and pick on it. The chasing and picking of a wounded chicken attracts more chicken and a deadly crowd of chicken builds. Since eventually this leads to the death of the wounded chicken, the emergence of such spatial, moving clusters of chicken is sure an unwanted negative emergent effect.

In order to observe, classify, and control this behavior automatically in Mnif et al. (2007) the observer/controller paradigm is studied in a simulation, which reproduces the collective cannibalistic behavior of chickens. An entropy-based measurement method taken from Mnif and Müller-Schloer (2006) was used to observe the spatiotemporal chicken patterns. To reach the final goal of maximising the lifetime of the simulated chickens the controller part of the system disperses chicken swarms or even prevents their formation by emitting noise which frightens the chicken.

3.1.1 Drawbacks of Feedback Loops

As shown, the use of feedback loops like promoted in the observer/controller architecture, is one way to control emergence in technical systems. Different grades of distributions can be assumed for these feedback loops. In the most centralized case one global loop controls the system and tries to reach given goals. Here the controlling part has a global view on the system and can measure the global system states, if needed. In large distributed systems such a global feedback loop is not always applicable, since sometimes it is not possible to collect all needed information in a reasonable time or the amount of this data is to large. In such cases a decentralized distributed organization of feedback loops, each controlling only subsystems, is needed (Cakar et al., 2007). The distributed local feedback loops only have a local view on the system and their direct control interventions only affect specific subsystems. This is an important point, since the realisation of a global goal for the whole system must emerge through the realisation of the local goals of the introduced local feedback loops. At the end, to design such decentralized local feedback loops can again lead to the problem of how to engineer emergent effects.

Beside the problem of feedback loops in large decentralized systems, there are situations where its even not possible to add a feedback loop to a system. If a system is already in use and the necessity for control did not show up in its design phase, it can be too costly to modify the system. For example "a clear case of emergent misbehavior" (Mogul, 2006) is the so called ethernet capture effect (Ramakrishnan and Yang, 1994). Ethernet hardware has been in significant use for serveral years, but this problem has not been seen until the hardware was fast enough to fully exploit the timing allowed in the ethernet specification. To exchange all the ethernet hardware is definitely no option and so other ways of dealing with the problem have to be found.

In general the options to control a self-organizing system without changing its existing parts are limited, since the only way to do so is by adding something. In case the system is scalable in the number of components, a way to impose control is to add new components to the system. These components have to interact with the system in the same way the other components of the system already do, but they can be designed to use a different behavior. Results of this may be that the emergent effect disappears or is changed and also new emergent effects may occur on system level. A second option is to leave the agents as they are and to add something that changes their environment. We will discuss both ideas in the following Sections.

3.2 Swarm Controlled Emergence

In some natural and technical systems it can be observed that a small fraction of individuals can influence a whole group and in this way have an effect on the emergent behavior on system level. For instance consider the model of collective movement behavior of animal groups introduced in Section 1.3. In this model the global emergent movement results from the local rules of the individuals. Recently developed models of moving animal groups divide the population into naive and "informed" individuals (Couzin et al., 2005). Whereas naive individuals follow the classical collective motion rules, members of the informed sub-population update their orientations according to a weighted average of the "normal" social rules and a fixed "preferred" direction, shared by all the informed individuals. These models can explain how a small fraction of informed individuals can guide the whole group into a desired direction, as for example found in bees moving to a new nest side (Janson et al., 2005). Also the introduction of artificial individuals can influence the moving directions and aggregation behavior of real animal swarms, for example shown in experiments with fish (Sumpter et al., 2008) or cockroaches (Caprari et al., 2004). A small fraction of (artificial) individuals, using a slightly different behavior, is able to influence the collective emergent movement of the animal group in a significant way.

A subset of the individuals is thus able to control emergent effects on system level. From the systems point of view there is no difference between these controlling individuals and the usual ones, since both interact in the same way with the rest of the system. We stipulate this principle as a general method for controlling emergent effects in technical systems and call this approach *swarm controlled emergence*. Swarm controlled emergence means to introduce so called *control agents* or *control components* into a system in order to control emergent effects on system level. The new system consisting of usual and control components will show new properties, i.e., the emergent effect which is intended to be controlled can disappear or change. Even new effects can occur. In this way "to control emergence" is an emergent effect itself, based on the interaction of a swarm of control components/agents with the system.

A schematic comparison of a feedback loop based control approach and the swarm controlled emergence approach is given in Figure 3.1. The circles and their connections represent the self-organizing components of the system with their local relationships and interactions. On a higher system level emergence occurs. In the feedback loop controlled system depicted in Figure 3.1 (a), local and global measures of the system state are observed and based on these information and (e.g. learned) knowledge the system is influenced to control the emergent effect. To impose control the components of the systems are often designed to have some kind of control interfaces.

Comparing the swarm controlled system depicted in Figure 3.1(b) there is no special kind of control effectors needed. The system is controlled through the introduction of control components (depicted as circles with a "C") that infer in the same way with other components as the usual components do. A prerequisite for applying swarm control is that the system must be scalable in terms of components, i.e., there must be the possibility to introduce and integrate new components into the system or at least there must be the possibility to exchange some of the components against control components. The gener-

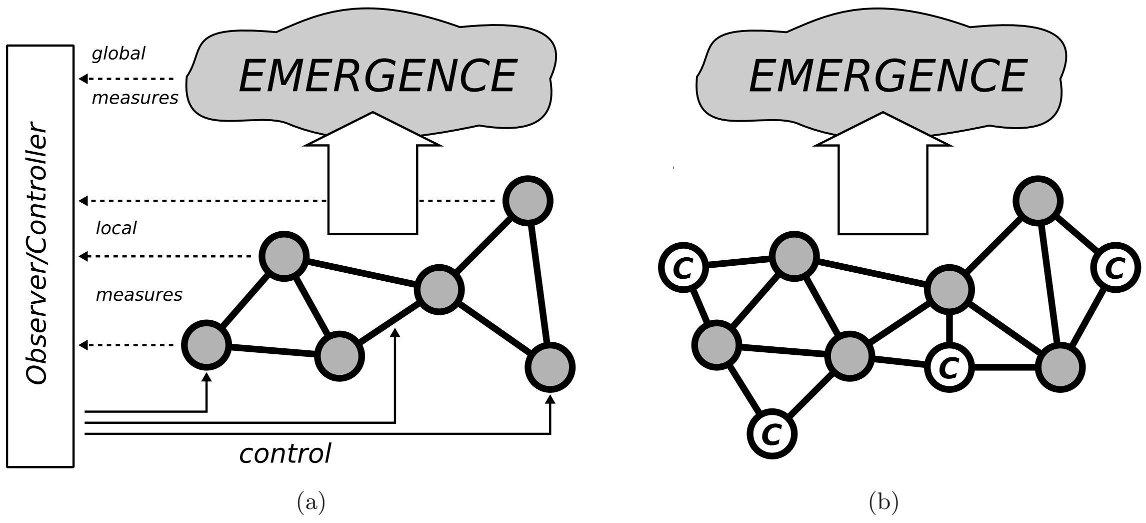


Figure 3.1: (a) Controlling emergence through the introduction of a feedback loop following the centralized observer/controller approach; (b) controlling emergence using the swarm controlled emergence approach through the introduction of control components

ality of the method should not be overestimated and the applicability must be considered from system to system. There are cases where it makes not much sense to use a swarm controlled approach. Preventing the mentioned cannibalistic behavior in chicken farms by introducing robotic or genetically modified control chickens or controlling the discussed bunching effect of lifts by installing additional lifts or by employing control passengers could work, but it probably makes not much sense. Swarm controlled emergence is an additional possibility for controlling emergence beside others.

To design the behavior of the control components leads the problem already discussed in Section 1.3, i.e., the problem of how to engineer emergent effects in general. Beside the use of "design patterns for emergent effects" for example taken from nature self-organizing systems, it was suggested that a manual or automatic trial-and-error process can help to find appropriate local rules. This means in case of designing control components experiments must be made by applying test implementations of the control components to the real systems or by testing them via simulations. If possible, this search process for good control components and depending parameters can be (at least partially) done automatically, for example by using evolutionary algorithms or by simply sampling the possible parameter spaces. Often it will be necessary to engineer and test the control components manually. This can make an excessive search for appropriate behaviors and parameters necessary.

The introduction of a second swarm of components with different behavior into a system is not only useful in terms of controlling emergent effects, but it can also lead to a deeper understanding of the emergent effects in the considered system. For example, it can show how robust an emergent effect is, that is, how many “misbehaving” components the system can handle. As an example consider the network protocols relaying on the emergent self-synchronisation found in fireflys which we introduced in Section 1.3. A potential attacker might try to disturb or control the system by infiltrating it with adversarial nodes that do not follow the usual firefly algorithm. From the point of view of the attacker the swarm controlled emergence approach is used to influence an existing system. To make the network protocols more robust and to find ways to prevent the possibility of such attacks they need to be investigated in advance.

3.3 Applying the Swarm Controlled Emergence Approach to Ant Based Clustering

In the following we will apply the swarm controlled emergence approach to a well known model of emergent behavior in social insects. The model, which was already briefly introduced in Section 2.2.1, explains the clustering behavior of ants and was proposed in Deneubourg et al. (1990). Recall, in this model agents (representing ants) move randomly on an array of cells with randomly distributed items. The agents, also called *clustering agents*, make probabilistic choices to pick or drop items depending on the fraction of cells occupied by items in their neighborhood. Eventually this behavior leads to the emergence of clusters of items. The formation of clusters, as a higher system level property, is an emergent effect in the model (Handl et al., 2003b). This model is chosen to test the swarm controlled emergence approach, because it is well known and has been studied in many variants and applications. For example, Samaey et al. (2008) used the model to illustrate their approach of an equation-free macroscopic analysis of self-organizing emergent systems.

The task we are dealing with in the following, is to control or even prevent the emergent clustering effect in systems implementing this model. We assume the size of the array, the number of randomly distributed items and the number the clustering agents and their probability functions for picking and dropping, to be given and fixed. The task of controlling/preventing the clustering must be realized without modifying these given parts in any way. We apply the swarm controlled emergence approach by introducing different numbers and types of control agents. Since the given systems are assumed to be fixed, these control agents are restricted to interact with the systems like usual clustering agents do. They can pick up, carry, and drop items and they have the same perception of the system as clustering agents, i.e., they can measure the fraction of occupied cells in their neighborhood. We further assume their internal functioning to be very similar to the one of the clustering agents. That is, the behavior of the control agents is realized by the use of modified probability distributions for picking and dropping items. The same as clustering agents, they move randomly in the array and they have no memory or any other sophisticated behavior. These relatively restricted assumptions keep the investigations of this proof of concept of swarm controlled emergence manageable and simple. When applying swarm controlled emergence to control emergent effects in technical systems, the realizable implementations of agents in general determine the possible behaviors of control agents.

We investigate systems that contain clustering agents in combination with additionally introduced control agents. The outcomes and dynamics of these systems is compared to systems without control agents. The systems are simulated over a number of time steps. The final item distributions are visually observed and several measures are applied in order to quantify the influence of the different parameter settings on the simulations outcome.

Not much work has been done on the topic of ant-based clustering systems using agents with different behavior at the same time yet. In Lumer and Faieta (1994) the agents have different velocities, what can be seen as having different behavior. In Handl et al. (2006) the sorting agents change their behavior over time. There is a phase a different neighborhood function is used. This causes the agents to not do the normal sorting, instead they spread out the items. Such a phase was shown to improve the overall sorting results.

A work using different agents at the same time is given in Magg and Boekhorst (2006); Magg and te Boekhorst (2007). These papers consider a two dimensional array with movable items. Two types of agents, called Dozers and Grabbers, act within this array by moving items and dropping pheromone. Dozers keep areas free of particles by pushing them to the next wall or pile, whereas Grabbers carry away items and try to drop them in free areas with few pheromone. The resulting distribution patterns for different ratios of these two types of agents and different pheromone dropping variants are analysed visually and by using an entropy measure. In contrast to our work the used movement, picking, and dropping rules differ strongly from the one used by Deneubourg et al. (1990). Also we do not incorporate a pheromone which affects the behavior of the agents.

3.3.1 A Model of the Pile Formation in Ants

The basic model used in our study was proposed in Deneubourg et al. (1990). This simple agents based model was developed to explain the clustering behavior in ants, i.e., the pile formation of indistinguishable items (brood items or dead nestmates). A behavior that has been found in real ants colonies and in other social insects as well. In the model several items are distributed in a two-dimensional toridial array of cells (at most one item per cell). Agents are distributed within the array and in one time step each agent moves randomly to one of the four directly neighbored array cells. Thereafter, if the agent does not carry an item and there is an item located at its new position, the agent picks up the item with a certain probability. If the agent already carries an item and the new position of the agent is empty, the agent drops its item with a certain probability. The probabilities for picking and dropping items depend on the items within the neighborhood of the agent. Formally, the probabilities p_{pick}^{clust} for an unladen cluster agent to pick up an item and $p_{drop}^{clust}(f)$ for a laden cluster agents to drop its item are given by:

$$p_{pick}^{clust}(f) = \left(\frac{k^+}{k^+ + f}\right)^2 \qquad \text{and} \qquad p_{drop}^{clust}(f) = \left(\frac{f}{k^- + f}\right)^2,$$

where f is the neighborhood function and $k^+ > 0$, $k^- > 0$ are threshold parameters. Different methods for defining the neighborhood function f have been proposed. One method is to count how many items were encountered by the agent within a given time window and define f as the fraction of time steps where the agent moved across cells that were occupied by an item. Another way to determine f is to calculate the fraction of cells that are occupied with item in the von Neumann neighborhood of the agent. In our study we use the latter definition.

3.3.2 Anti-Clustering

The aim is to construct control agents which behave similar to the standard clustering agents but can, when added to an existing clustering system, reduce (or prevent) the clustering effect. We call these control agents anti-clustering agents, or $\mathcal{AC}$-agents. As mentioned before the different types of anti-clustering agents, which will be introduced in the following, behave the same way as the clustering agents. The cluster agents and $\mathcal{AC}$-agents only differ in the probability distributions for dropping and picking up items.

Reverse $\mathcal{AC}$-agents

For reverse $\mathcal{AC}$-agents the probabilities that a clustering agent picks up an item or drops an item in a certain situation are swapped. Such an agent will drop (respectively pick up) an item with the same probability as a cluster agent would pick up (respectively drop) an item in the same neighborhood f:

$$p_{pick}^{rev}(f) = p_{drop}^{clust}(f) = \left(\frac{f}{r^+ + f}\right)^2$$

$$p_{drop}^{rev}(f) = p_{pick}^{clust}(f) = \left(\frac{r^-}{r^- + f}\right)^2$$

where r^+ and r^- are the threshold values used for reverse agents. In most experiments both parameters $r = r^+ = r^-$ are set to the same value.

As an extreme case of the reverse $\mathcal{AC}$-agent behavior we introduce the so called strict reverse $\mathcal{AC}$-agents, or just strict $\mathcal{AC}$-agents. They pick up items with probability 1 if there is any other item in their neighborhood and drop items only if there is no item in the neighborhood. Such a behavior is denoted by $r = 0$, since its the same as the reverse $\mathcal{AC}$-agents behavior for the threshold parameters $r^+ = r^-$ close to zero.

An alternative possibility to revert the behavior of the clustering agents is to reverse the probability of picking and dropping so that $p_{pick}^{rev}(f) = 1 - p_{pick}^{clust}(f)$ and $p_{drop}^{rev}(f) = 1 - p_{drop}^{clust}(f)$. But since these functions lead to very similar probability distributions as the ones that are used for the definition of the reverse $\mathcal{AC}$-agents further investigations of the corresponding $\mathcal{AC}$-agents are omitted.

Inverted neighborhood $\mathcal{AC}$-agents

Inverted neighborhood $\mathcal{AC}$-agents have the same behavioral rules as cluster agents, but they have an inverted perception of the neighborhood, i.e., on every neighbored cell (but not on the cell they are currently placed) they see an item when the cell is empty and otherwise they see no item. Thus, these agents drop items with higher probability when less item are in their neighborhood. Formally, the probability to pick up an item $p_{pick}^{inv}(f)$ and to drop an item $p_{drop}^{inv}(f)$ are

$$p_{pick}^{inv}(f) = p_{pick}^{clust}(1 - f) = \left(\frac{i^+}{i^+ + 1 - f}\right)^2$$

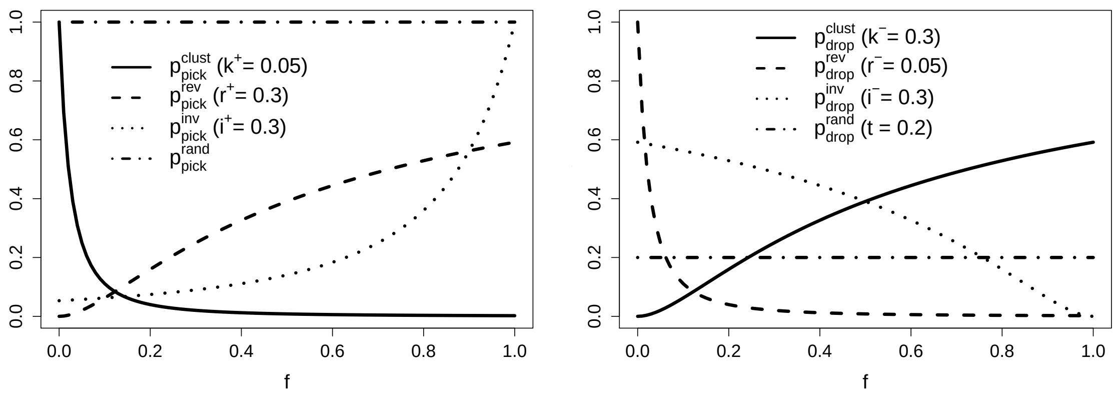
$$p_{drop}^{inv}(f) = p_{drop}^{clust}(1 - f) = \left(\frac{1 - f}{i^- + 1 - f}\right)^2,$$

where i^+ and i^- are the threshold values used for inverted neighborhood $\mathcal{AC}$-agents.

Random $\mathcal{AC}$-agents

Introducing sufficient randomness in the behavior of the agents in the sense that items are transported to random cells can obviously hinder a strong clustering. Random $\mathcal{AC}$-agents always pick up items when they enter an occupied cell. If such an agent carries an item, it drops it on an empty cell with a fixed probability $t > 0$. Formally, for random $\mathcal{AC}$-agents the probability to pick up an item $p_{pick}^{rand}(f)$ and to drop an item $p_{drop}^{rand}(f)$ are

$$p_{pick}^{rand}(f) = 1 \qquad p_{drop}^{rand}(f) = t.$$



(a) (b)

Figure 3.2: Probabilities to pick up (a) and to drop (b) an item for the different types of agents (cluster agents, reverse $\mathcal{AC}$-agents, inverse neighborhood $\mathcal{AC}$-agents and random $\mathcal{AC}$-agents); f denotes the fraction of occupied cells in the neighborhood

In Figure 3.2 examples of the probability functions to pick up or drop items for the different agents are given.

3.3.3 Clustering Measures

One central aspect for our study is to measure how the characteristics of emerging clusterings changes for different systems of agents. In order to make the results not dependent on a single way to quantify the results several measures for the degree of clustering are used.

Number of Clusters N_d

The number of clusters N_d is the number of different connected regions of cells where each cell has more than d items in its neighborhood. In other words, the measure counts the number of high density regions. If not stated otherwise the considered density level is $d = 75$ and the used neighborhood is the same as used by the agents (by default a von Neumann neighborhood with radius 10).

Spatial Entropy E_s

Gutowitz (1995) and Bonabeau et al. (1999) suggest to measure the spatial entropy to track the dynamics of ant based clustering. This measure can be used to classify spatial distributions of items according to their cluster validity on different spatial scales. To calculate the spatial entropy the (two-dimensional) cell array A is partitioned into so called s-patches, i.e., subarrays of size $s \times s$. Let p_I be the fraction of cells in a s-patch I that are occupied by an item. Then the spatial entropy E_s at scale s is defined as

$$E_s = - \sum_{I \in \{s\text{−patches}\}} p_I \log p_I$$

Hierarchical Social Entropy S

Balch (2000) proposes the so called hierarchical social entropy measure which is defined as described in the following. Let $\mathcal{R} = \{r_1, \ldots, r_N\}$ be the items for which the measure is to be calculated and for each pair of items in $\mathcal{R}$ let $d(r_i, r_j)$ be a dissimilarity measure.

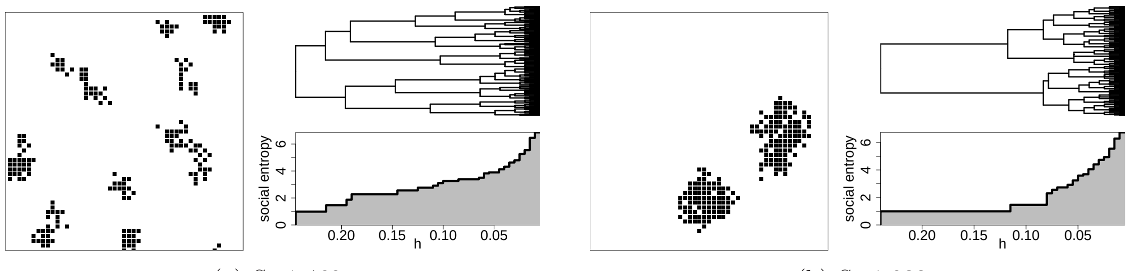
Based on the dissimilarity measure a hierarchical clustering can be calculated as follows. Initially each item is assigned to its own cluster. Then iteratively the two most similar clusters are merged, until there is one single cluster left. We choose the complete linkage method for calculating the dissimilarity between two clusters, which is defined as the maximal dissimilarity between two arbitrary items of these clusters (for more details on hierarchical clustering see, e.g., Day and Edelsbrunner, 1984).

The hierarchical clustering leads to a dendrogram which visualizes the agglomeration process in a binary tree, where the leaves of the tree represent the items. Two nodes are siblings if their corresponding clusters are agglomerated during the hierarchical clustering. A cut through the dendrogram at level $h \geq 0$ defines a clustering $\mathcal{C}(h) = \{C_1, \ldots, C_{M(h)}\}$, where $M(h)$ is the number of clusters at h. For every cluster $C \in \mathcal{C}(h)$ the maximum dissimilarity between all pairs of items $r_i, r_j \in C$ is smaller or equal than h, i.e., $d(r_i, r_j) \leq h$ and the dissimilarity between all pairs of clusters $C_1, C_2 \in \mathcal{C}(h)$ is larger than h.

The hierarchical social entropy of a set of items $\mathcal{R}$ is defined as

$$\mathcal{S}(\mathcal{R}) = \int_0^\infty H(\mathcal{R}, h) dh,$$

where $H(\mathcal{R}, h) = - \sum_{i=1}^{M(h)} p_i \log_2(p_i)$ is the simple social entropy of $\mathcal{R}$ at level h and $p_i = \frac{|C_i|}{|R|}$ is the proportion of items in the i-th cluster $C_i \in \mathcal{C}(h)$. Note, that the hierarchical social entropy is invariant in relation to the scale of the dissimilarity measure. Balch (2000) uses the measure to calculate the diversity of a set of robots and to distinguish between fine grained and coarse grained clustering situations. In Figure 3.3 two clustering situations, the resulting dendrograms, and the values of the social entropy at different levels are depicted.



(a) S=1.469 (b) S=1.032

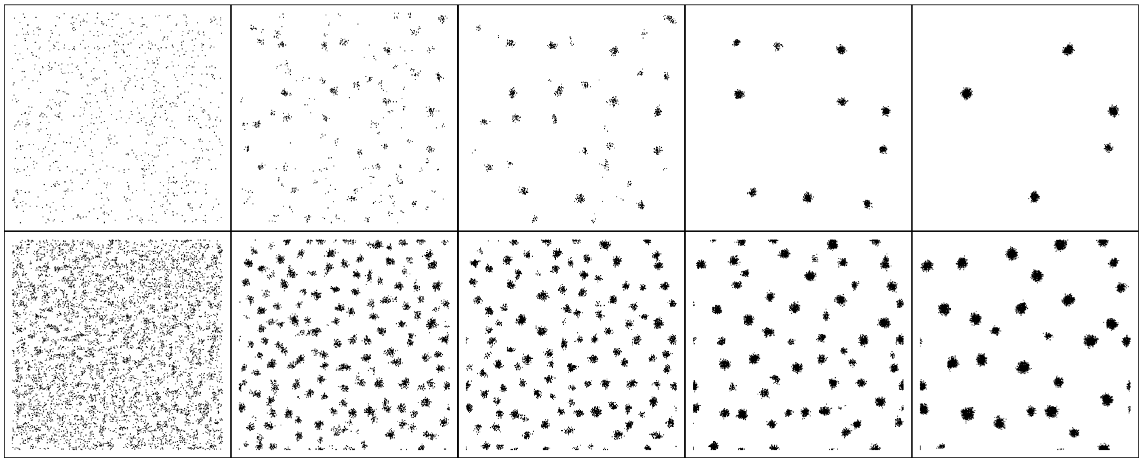
Figure 3.3: Hierarchical social entropy; shown are two clustering situations on a 50×50 array with corresponding dendrogram, social entropy values at different taxonomic levels and hierarchical social entropy value

3.3.4 Experiments and Results

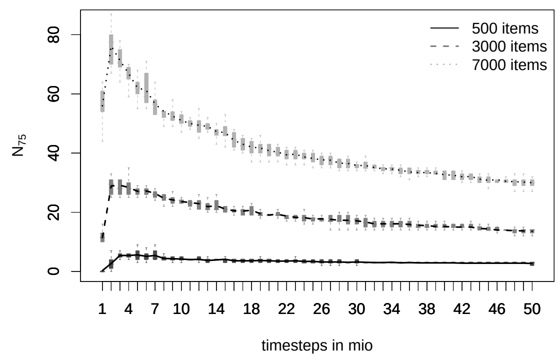
For all experiments a two-dimensional array of cells with size 500×500 is used. In the initial state the items and agents are distributed randomly within this array. The number of cluster agents is set to 50 and different numbers of $\mathcal{AC}$-agents are added to the system. The neighborhood of an agent is defined as the von Neumann neighborhood with radius 10, i.e., all cells for which $d_x + d_y \leq 10$ holds are in this neighborhood, where d_x and d_y are the absolute distances of the considered cell to the cell of the agent in the two dimensions. The threshold parameters for the clustering agents are chosen as $k^+ = 0.05$ and $k^- = 0.3$, since this leads to a good clustering performance for the used item densities. If not stated otherwise, the results are given after 50 million simulation steps.

No $\mathcal{AC}$-Agents

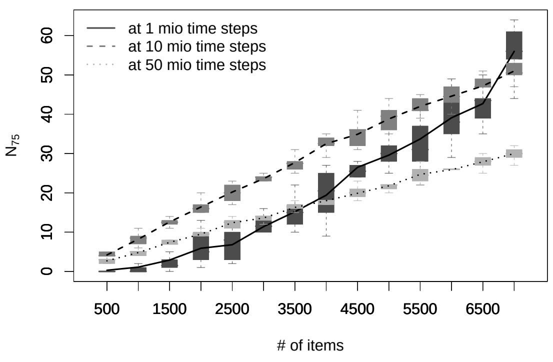
For reasons of comparison the clustering behavior of a system without $\mathcal{AC}$-agents is studied first. Figure 3.4(a) shows the clustering behavior of such a system after different time steps for 1000 and 7000 items. It can be seen that there is a strong clustering with a decreasing number of clusters over time in both cases. At the same time step in systems with 7000 items more clusters can be observed than in systems with only 1000 items. Figure 3.4(b) shows this effect quantitatively using the N_{75} measure. During the first few million simulation steps an increasing number of clusters emerge. After a maximum is reached the number of clusters slowly decreases. The reason is that clusters disappear and the items of these vanishing clusters are inserted into other clusters. These results coincide with the findings of Theraulaz et al. (2002). The influence of the number of items on the clustering measures N_{75}, E_{25}, and $\mathcal{S}$ at different time steps is shown in Figures 3.4(c), 3.4(d) and 3.4(e). For all three measures at a given simulation time in systems with more items the values of the clustering measure are higher. Both entropy measures decrease



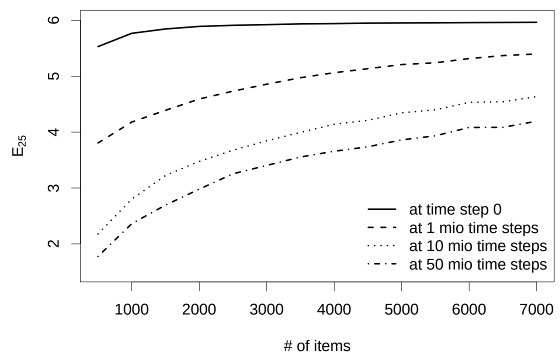
(a)



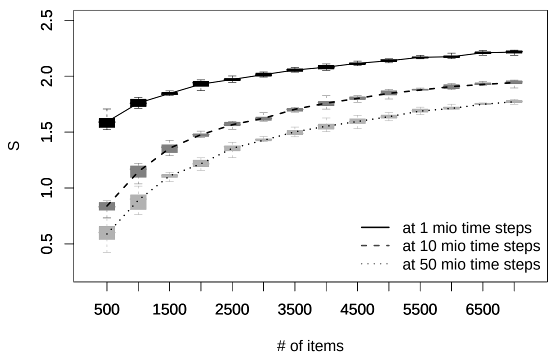
(b)



(c)

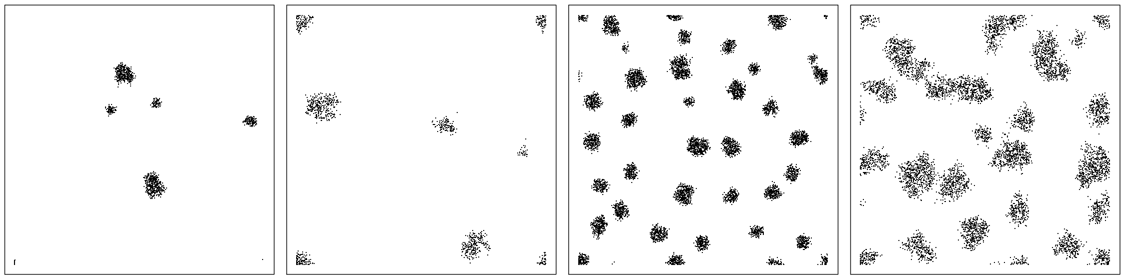


(d)



(e)

Figure 3.4: System with only clustering agents; (a) distributions of 1000 items (top row) respectively 7000 items (bottom row) at simulation steps 100 000, 1 000 000, 2 000 000, 10 000 000 and 50 000 000 (from left to right); (b) number of clusters N_{75} for different number of items over time; (c) ((d), (e)) number of clusters N_{75} (respectively, spatial entropy E_{25}, hierarchical social entropy $\mathcal{S}$) at different time steps for different number of items



(a) 1000 items, 100 inv. $\mathcal{AC}$-agents (b) 1000 items, 1000 inv. $\mathcal{AC}$-agents (c) 7000 items, 100 inv. $\mathcal{AC}$-agents (d) 7000 items, 1000 inv. $\mathcal{AC}$-agents

Figure 3.5: Example distributions of different number of items for different numbers of inverted neighborhood $\mathcal{AC}$-agents

with an increasing number of simulation steps because the emerging clustering increases the order in the system. The comparison of the values of the two entropy measures is difficult. Even at time step zero, when there is a "perfect" disorder in the systems, the values differ.

Inverted Neighborhood $\mathcal{AC}$-Agents

The influence of inverted neighborhood $\mathcal{AC}$-agents on the clustering is shown in Figure 3.5. After experimenting with different values of the parameter i^+, a value of $i^+ = 0.3$ has been used for the simulations because for this value the inverted neighborhood $\mathcal{AC}$-agents had the strongest influence.

It can be seen that 100 inverted neighborhood $\mathcal{AC}$-agents have a very small influence on the clustering and even 1000 inverted neighborhood $\mathcal{AC}$-agents can not hinder a clustering. The only effect is that the clusters become more diffuse, i.e., more cells within the clusters area are not occupied by items. For the inverted neighborhood $\mathcal{AC}$-agents regardless of the chosen value for parameter i^+, the probability to drop an item in a neighborhood with $f \approx 0.2$ is nearly the same as for $f = 0$. This is the reason why the inverted neighborhood $\mathcal{AC}$-agents are so weak and the inverted neighborhood $\mathcal{AC}$-agents are no good choice for preventing a clustering.

Random $\mathcal{AC}$-Agents

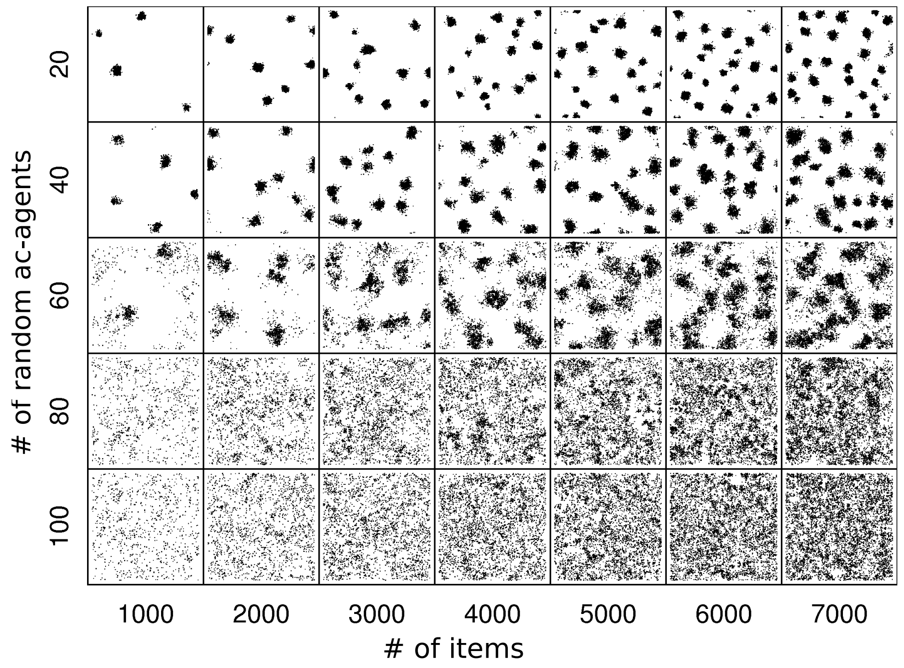
Figure 3.6(a) shows sample distributions of the items at simulation step 50 000 000 for different numbers of random $\mathcal{AC}$-agents with parameter $t = 0.1$. It can be seen the more random agents are introduced into the system the less clustering occurs. When the number of random $\mathcal{AC}$-Agents is the same as the number of clustering agents the clusters become diffuse. When the system has about twice as much random $\mathcal{AC}$-Agents as clustering agents no larger clusters of items occur. These results hold independent of the number of items that have been tested. Hence, the random $\mathcal{AC}$-Agents are possible candidates to be used as anti-clustering agents.

In Figure 3.6(b) the influence of parameter t which denotes the probability that a random $\mathcal{AC}$-agent drops an item is depicted. The figure shows that the strength of clustering strongly depends on that parameter t. If the value of t is high it is likely that the random agents drop a picked up item very fast again. Thus, the items are not transported very far and therefore the random $\mathcal{AC}$-agents have only a small influence on the system. If the parameter t is small the random $\mathcal{AC}$-agents are likely to carry an item for a long time without dropping it. In this case their influence on the clustering is not very strong, too. Intermediate values of t ($0.05 \geq t \geq 0.1$) used by larger numbers (≥ 60) of random $\mathcal{AC}$-agents can prevent a strong clustering. To quantify the dependency of the system of parameter t the spatial entropy measure E_{25} is shown in Figure 3.7 for different numbers of random $\mathcal{AC}$-agents and different values of t. The figure shows that the t values where the entropy is highest (and the clustering is weakest) are smaller for a larger number of random $\mathcal{AC}$-agents. Thus, the larger the number of random $\mathcal{AC}$-agents, the shorter the time they should carry the items.

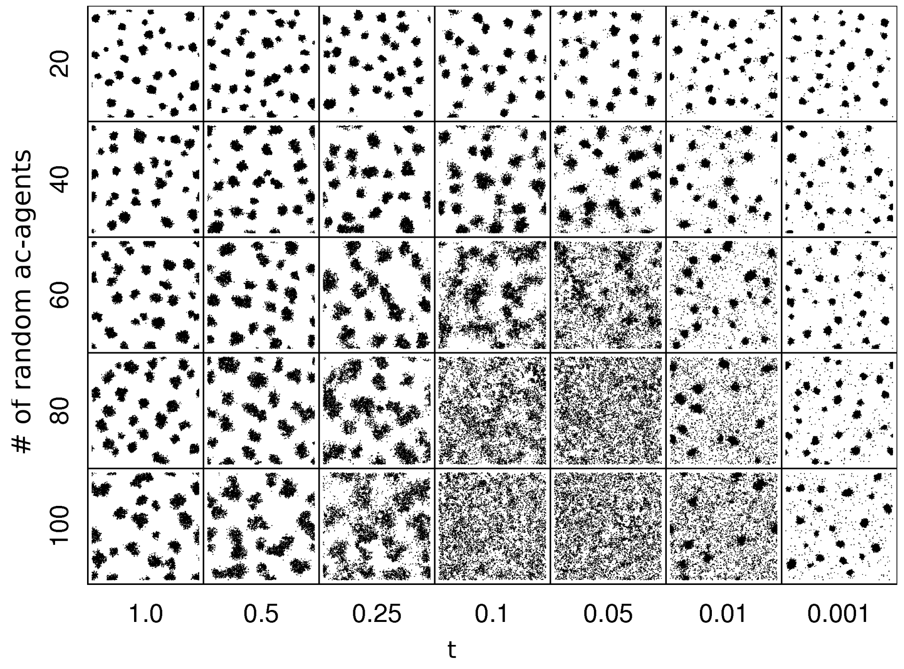
Reverse $\mathcal{AC}$-Agents

Figure 3.9 shows sample item distributions for a system with 50 reverse $\mathcal{AC}$-agents for different values of parameter r and different numbers of items. Several effects can be observed. First, reverse $\mathcal{AC}$-agents with the same parameters as the clustering agents (second row) can not hinder a clustering, although they make the clusters more diffuse. Second, for a small number of items (≤ 2000) the lower the r parameter, the more diffuse the patterns become. Third, for a large number of items (≥ 3000) the clusters disappear for medium values of r but for small values of r clusters occur. The values of r that lead to the occurrence of clusters depends on the number of items. For example, for $r = 0.0025$ in the system with 7000 items about 10 large clusters are clearly visible. Whereas no such clusters occur for a system with 5000 items and the same r value (for $r = 0.001$ clusters are visible also for the case of 5000 items).

The fact that clusters occur for a low r value and a high number of items can be explained in the following way. The lower the r value, the smaller is the probability of dropping an item if there are other items within the neighborhood. For the strict reverse behavior ($r = 0$, last row), the probability of dropping an item is nearly zero if there is any item in the neighborhood. In situations where almost every cell in the array has a neighboring item and r has a low value (for example, for 3000 items and parameter $r = 0.001$) the probability of dropping an item is very small in most cells. Therefore, the



(a)



(b)

Figure 3.6: (a) Example distributions of different number of items using varying numbers of random $\mathcal{AC}$-agents with parameter $t = 0.1$; (b) Example distributions of 7000 items using different numbers of random $\mathcal{AC}$-agents and varying parameter t

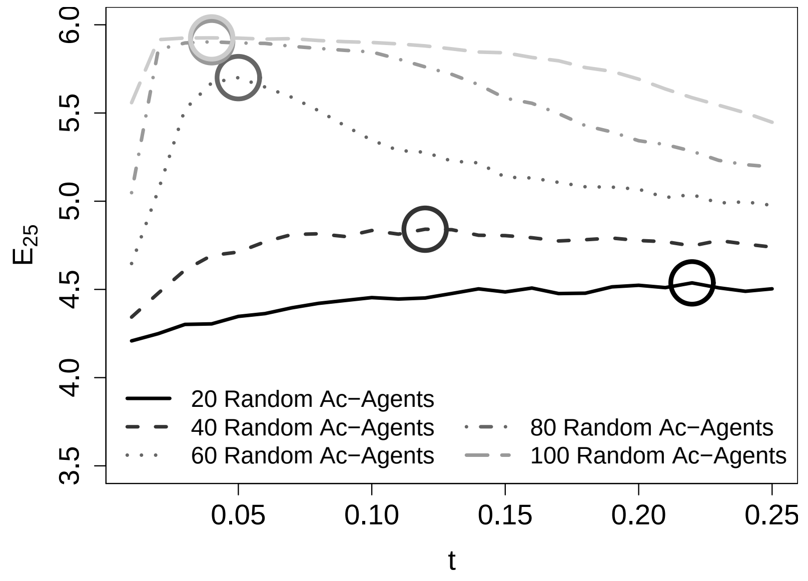


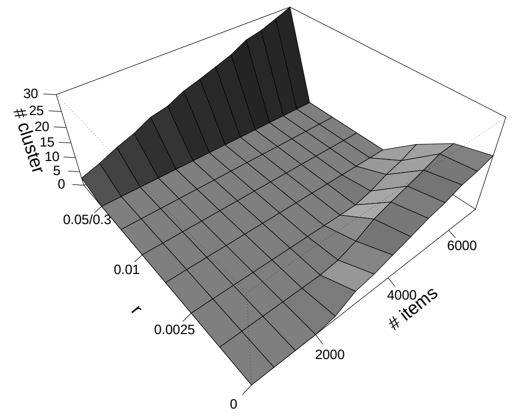
Figure 3.7: Spatial Entropy E_{25} using different numbers of random $\mathcal{AC}$-agents for varying parameter t; maximum values are marked by circles

$\mathcal{AC}$-agents spend all their time carrying items in search for places to drop them, whereas the clustering agents can build clusters from the remaining items.

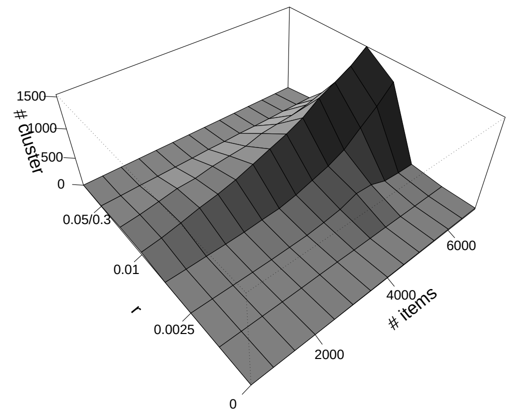
To investigate the system in more detail we consider the clustering measure N_c. Figure 3.8 depicts for different parameters r and different numbers of items the values of N_{75} 3.8(a) and N_{10} 3.8(b). Measure N_{75} shows that only for high or low values of r in combination with a high number of items clusters of high density occur. This fits with the observations that can be made from Figure 3.9 and shows that for medium values of r the reverse $\mathcal{AC}$-Agents work well. Measure N_{10} shows that for medium values of r and larger number of items many (1500) small clusters of low density 3.8(a) are build.

The spatial entropy E_{25} and the hierarchical social entropy S depending on r are depicted in Figure 3.10(a) and 3.10(b). Since low entropy values signalize a strong clustering, high values are deserved in terms of anti-clustering. It can be observed that the entropy values depend not only on the chosen parameter r but also on the number of items. For example, when there are 3000 items in the system a value of $r = 0.005$ leads to the highest spatial and hierarchical social entropy, i.e., the strongest anti-clustering effect. On the other hand for 7000 items a value of approximately $r = 0.05$ is best.

In summary, it can be noted that a number of reverse $\mathcal{AC}$-agents that is similar to the number of clustering agents is able to prevent a strong clustering when the parameter values are chosen adequately. To confirm this statement, simulations were done with a system that was initialized with a clustered situation. In this case the reverse $\mathcal{AC}$-agents destroy the initial clustering successfully. Hence, the reverse $\mathcal{AC}$-agents can be classified as



(a)



(b)

Figure 3.8: System with 50 reverse $\mathcal{AC}$-agents; number of clusters with a certain density for different parameters r and different number of items; (a) results for N_{75}; (b) results for N_{10}

efficient anti-clustering agents because they "win" against the same number of clustering agents.

To study the effect of the number of reverse $\mathcal{AC}$-agents in more detail, sample distributions of different numbers of items for systems with 35 reverse $\mathcal{AC}$-agents using different values of r are given in Figure 3.11. Comparing these distributions with those for an equal number of clustering agents and $\mathcal{AC}$-agents (Figure 3.9) it can be noticed that for $r \geq 0.01$ the outcome looks quite similar. Only for lower values of r and higher number of items the distributions of the items differ strongly. For instance, a clear difference can be observed comparing 35 reverse $\mathcal{AC}$-agents with 59 reverse $\mathcal{AC}$-agents in case of $r = 0.001$ and 4000 items. In the former occur empty areas between the clusters, whereas in the latter these areas are densely filled with items (Figure 3.9 and 3.11).

Comparing the values of the entropy measures for 35 reverse $\mathcal{AC}$-agents (Figure 3.12(a) and 3.12(b)) with the ones for 50 reverse $\mathcal{AC}$-agents (Figure 3.10(a) and (Figure 3.10(b)) it can be observed that for 1000 items there is nearly no difference. For 3000 and 7000 items using values of $r \geq 0.01$ also no difference can be seen. The entropy measures for 7000 items and values of $r < 0.01$ are slightly smaller for 35 reverse $\mathcal{AC}$-agents than for 50 reverse $\mathcal{AC}$-agents and there is a stronger shift near $r = 0.005$. But the main difference can be observed for systems with 3000 items and values of $r < 0.01$. Here the entropy measures are much smaller when using only 35 reverse $\mathcal{AC}$-agents. For example, the spatial entropy E_{25} is 6.3 for 50 strict $\mathcal{AC}$-agents (this means reverse $\mathcal{AC}$-agents with $r = 0$) but only 4.7 when using 35 of these anti-clustering agents. This indicates a strong clustering at these values. The value $r = 0.005$ leads to the strongest anti-clustering effect when

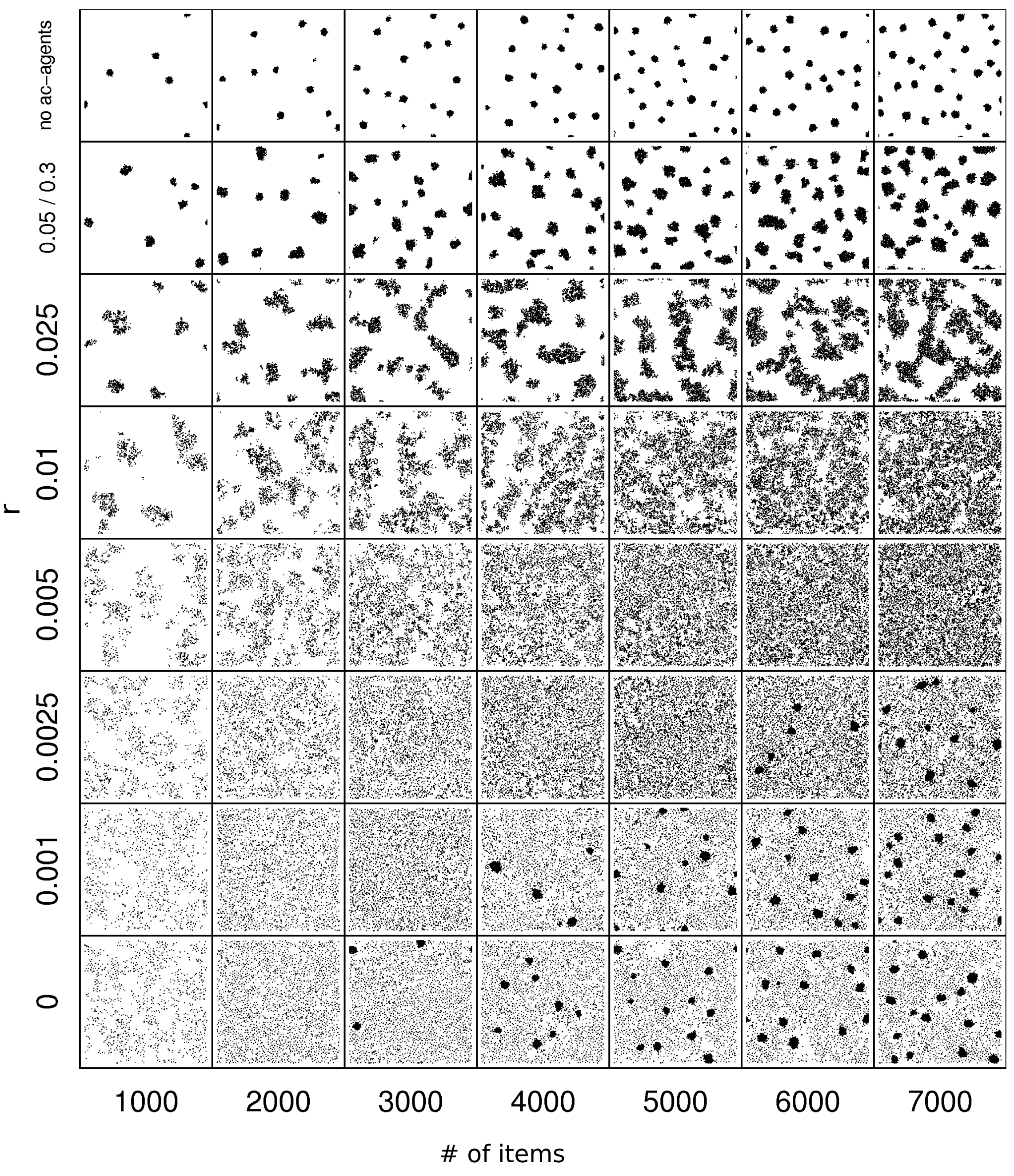
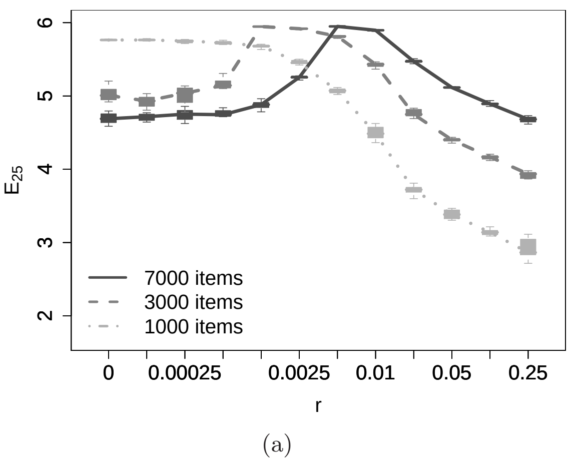


Figure 3.9: System with 50 reverse $\mathcal{AC}$-agents; example distributions of items for different number of items and different values of parameter r; second row $r^+ = 0.05$ and $r^- = 0.3$ (same values as for clustering ants)



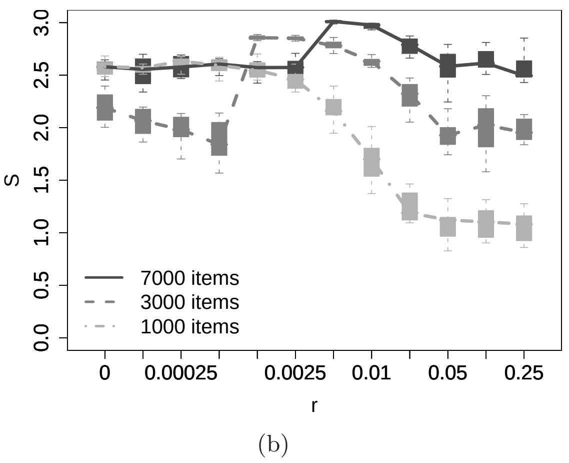


Figure 3.10: (a) Spatial entropy E_{25} and (b) social hierarchical entropy S for 50 reverse $\mathcal{AC}$-agents using different number of items for varying parameter r

using the same number of reverse $\mathcal{AC}$-agents as clustering agents. But when using only 35 $\mathcal{AC}$-agents these value for r leads to the worst results.

The emergence of empty areas between the clusters that can be observed for 35 reverse $\mathcal{AC}$-agents, small values of r and high number of items, can be explained as follows. At the beginning of the simulation the same happens as in a system with more (for example 50) reverse $\mathcal{AC}$-agents. A part of the items is distributed equally by the reverse $\mathcal{AC}$-agents, whereas from the other items the cluster agents form clusters. In situations where the neighborhood function is low the clustering agents are not very "strong" since the probabilities of picking an item and dropping it are nearly the same (compare $f \approx 0.1$ in Figure 3.2). Therefore, the clustering agents act nearly randomly in such a neighborhood. On the other hand in regions with a high density (within or next to already existing clusters) the clustering agents become "stronger". Thus, next to clusters the clustering agents are nearly as strong as the reverse $\mathcal{AC}$-agents. Since they outnumber the reverse $\mathcal{AC}$-agents they can move all items of the area next to the clusters.

An important observation at this point is the fact that there are parameter settings (for example $r = 0.001$ and 3000 items) for which the number of clusters after a certain simulation time is even less than in a system without reverse $\mathcal{AC}$-agents. This is because at the beginning only a part of the items are clustered by the clustering agents and the remaining items are equally distributed by the reverse $\mathcal{AC}$-agents. This leads to a smaller number of clusters, i.e., less "crystallization points" in the rest of the simulation. Therefore, also at the end the number of clusters is smaller.

This effect is interesting from an anti-clustering point of view, because it has to be eliminated as far as possible. It is also interesting in terms of ant based clustering algorithms because it shows that the incorporation of a second type of agents could help improve the

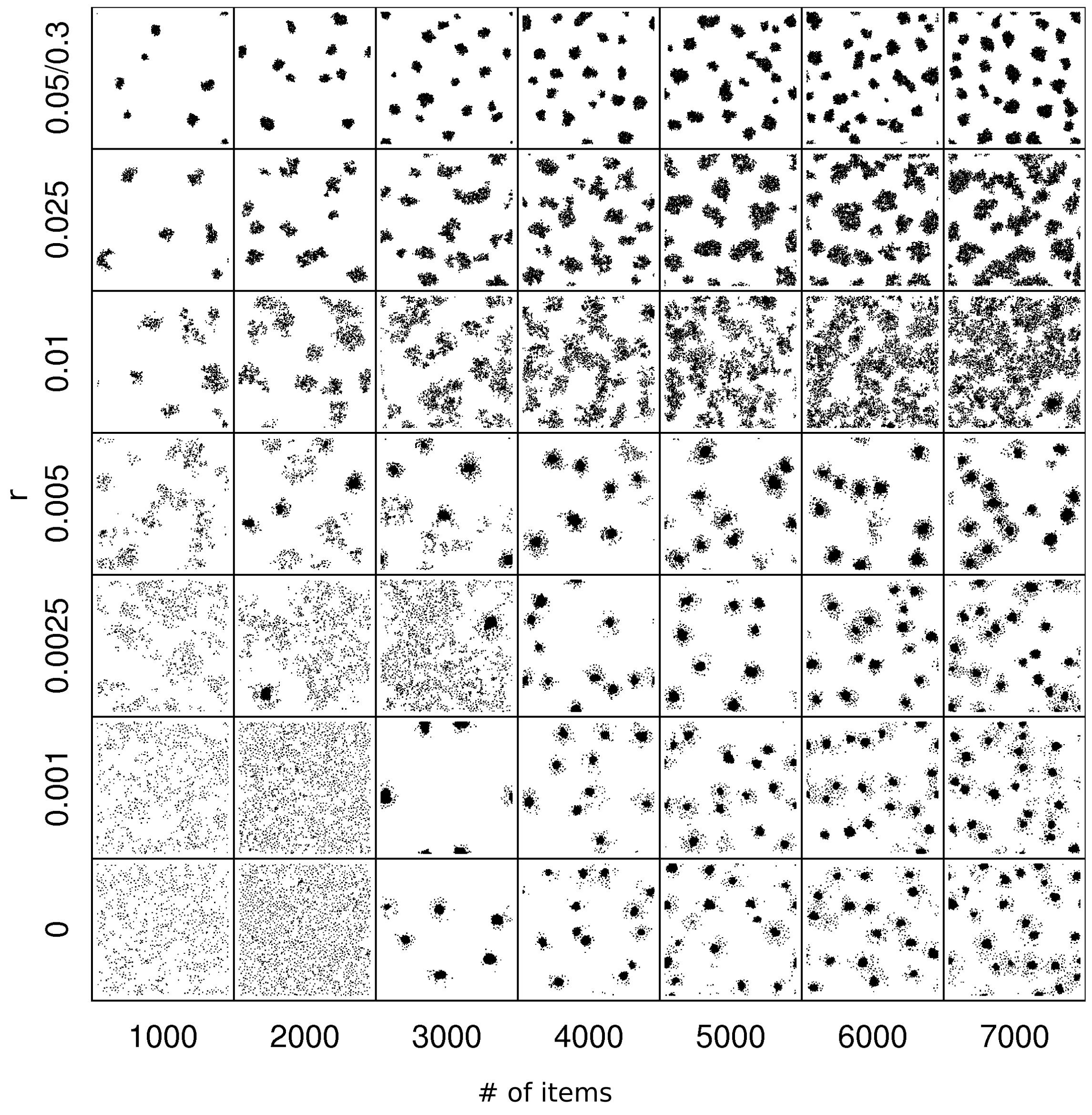


Figure 3.11: Example distributions of items after $50 * 10^6$ time steps for different number of items and different parameter r^+ and r^- used for the 35 reverse $\mathcal{AC}$-agents

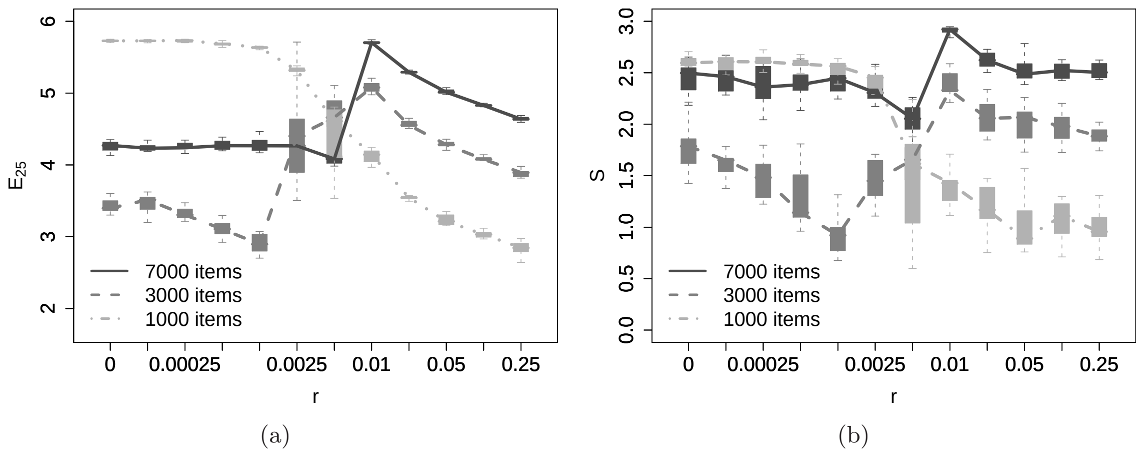
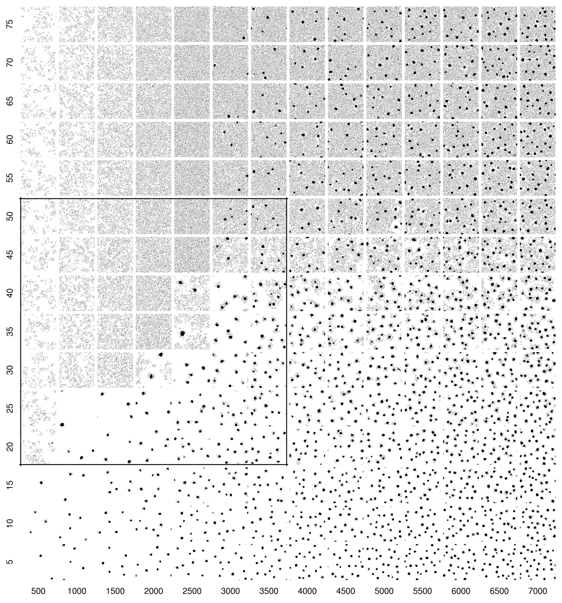


Figure 3.12: System with 35 reverse $\mathcal{AC}$-agents for different number of items and different values of r; (a) spatial entropy E_{25}; (b) social hierarchical entropy S

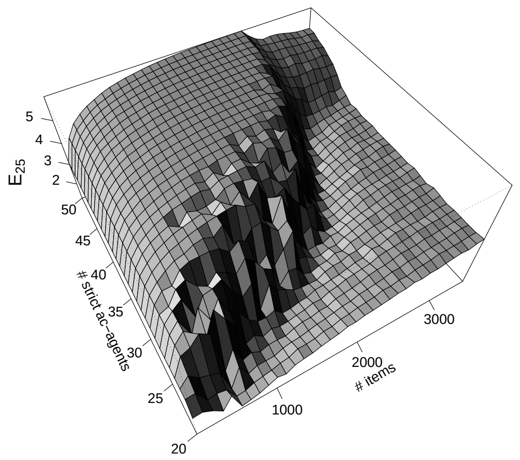
clustering process. Because of this the effect is studied in more detail in the following. To keep things simple, the following investigations concentrate on reverse $\mathcal{AC}$-agents with parameter value $r = 0$, i.e., strict $\mathcal{AC}$-agents.

First the influence of the number of strict $\mathcal{AC}$-agents has on the system is studied. In Figure 3.13(a) sample item distributions for varying numbers of strict $\mathcal{AC}$-agents and items are depicted. For a fixed number of items the distributions are very similar for any number of 50 or more strict $\mathcal{AC}$-agents. The analogous statement holds also for 20 or less $\mathcal{AC}$-agents. If all runs with more than 3500 items are compared it can be seen that there is no observable difference. The remaining cases are more interesting, i.e., simulations with 20 to 50 strict $\mathcal{AC}$-agents and at most 3500 item (framed part of the figure). In these simulations the resulting distribution strongly depends on the exact parameter values. In Figure 3.13(b) the spatial entropy for the interesting region is given. Three regions can be distinguished here. In the first region the anti-clustering works well, i.e., the spatial entropy is high. The second region has a very low spatial entropy, because of a strong clustering. And the third region is a small region in the upper right corner of the graph, where the number of items is ≥ 3000 and the number of strict $\mathcal{AC}$-agents is ≥ 40. In this region the spatial entropy has medium values because the relating simulations lead to some clusters within equally distributed items. These findings suggest that the strength of the anti-clustering agents depends strongly on the chosen parameters. This shows that the anti-clustering problem is not a trivial one.

The evolution of the spatial entropy E_{25} of a system with 2500 items over time is given in Figures 3.14(a) and 3.14(b). It can be observed that at the end a medium number (e.g., 30) of strict $\mathcal{AC}$-agents leads to the lowest E_{25}. Figure 3.14(b) shows that spatial entropy values for systems with 30 strict $\mathcal{AC}$-agents become smaller than for systems with 50 strict



(a)



(b)

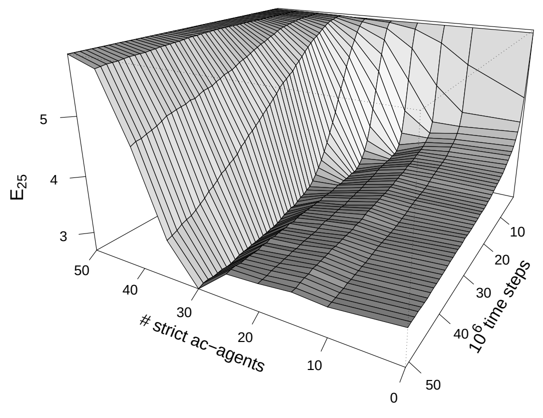
Figure 3.13: (a) System with different number of strict $\mathcal{AC}$-agents; distribution of items for different numbers of items; (b) spatial entropy E_{25} for parameter values that correspond to the framed area (a)

$\mathcal{AC}$-agents after about 16 million simulation steps. Thus, the introduction of a medium number of strict $\mathcal{AC}$-agents can improve the clustering in terms of spatial entropy.

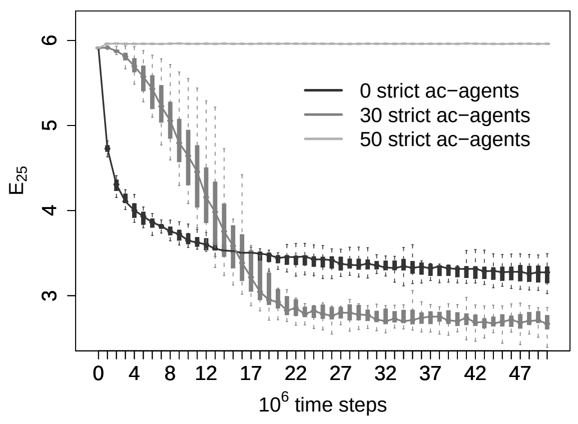
To make sure this interesting effect is not the result of a larger total number of agents (clustering agents plus $\mathcal{AC}$-agents) two systems with a total number of 80 agents are compared, i.e., one system with 80 clustering agents and one with 50 clustering agents and 30 strict $\mathcal{AC}$-agents. The results are shown in 3.14(c). Although the spatial entropy E_{25} for 80 clustering agents shows lower values than when using only 50 clustering agents, the values are still worse than for systems with a mixture of 50 clustering and 30 strict $\mathcal{AC}$-agents. The same holds for the number of clusters N_{75} and the social hierarchical entropy S - every measure suggests a significant better clustering for the agent mixture. This shows that a mixture of different types of agents can improve the clustering compared to a system of clustering agents.

3.4 Congestion Control in Ant Like Moving Agent Systems

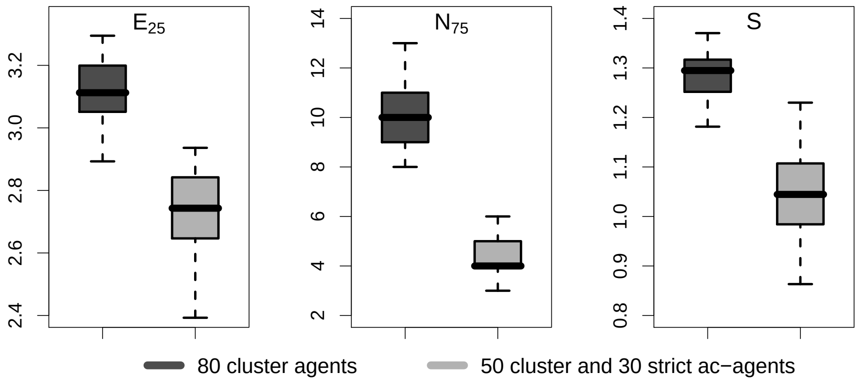
In the last section we introduced the swarm controlled emergence approach and investigated how to control the emergent pattern formation of items in a cell array that was originated by the interaction of the items with randomly moving agents. No direct interaction between these agents took place. In the following we shift our attention to systems



(a)



(b)



(c)

Figure 3.14: Spatial entropy E_{25} over time for different number of strict $\mathcal{AC}$-agents (a) and (b); measures E_{25}, N_{75}, and $\mathcal{S}$ for systems using 80 cluster agents compared to systems with 50 cluster agents and 30 strict $\mathcal{AC}$-agents (c)

of moving agents that interact directly. Direct interaction means that the agents are not allowed to move over each other ,i.e., agents are obstacles for other agents.

Different models for the movement of the ant *Leptothorax unifasciatus* within a nest have been introduced in Sendova-Franks and Lent (2002). It was shown that small differences in the movement behavior can lead to spatial sorting of the ants (i.e., on average over time ants with different behavior can be found in different areas of the nest), whereas the degree of the sorting depends on the particular movement model. In the model with the strongest sorting there is an attraction point in the nest center, which establishes a centripetal force on the ants. In natural ant nests this can be a CO_2 gradient which is assumed to point to the center of a brood chamber (Cox and Blanchard (2000); Nicolas and Sillans (1989)). In Scheidler et al. (2006); Scheidler (2005) this model has been investigated and modified slightly to avoid an unnatural blocking effect in the nest centre.

In Scheidler et al. (2006) we adapted the movement models to fit the requirements of systems of moving artificial agents influenced by multiple attraction points. Hereby, we assumed the agents have to visit one of several service stations from time to time (e.g., to recharge their batteries or to drop items they have collected). It was shown that emergent patterns in the distribution of the agents can occur even when only slight behavioral differences between the agents exist. These patterns are determined by the relative size of the influence area of the service stations. These results can find application in swarm robotic systems which exhibit different service stations for the robots. It was also shown, that using the ant inspired movement models an unwanted congestion can emerge at the service stations if there is a larger number of agents in the system.

In the following we investigate different methods for reducing and controlling this (negative) emergent congestion effects. Like the swarm controlled emergence approach the considered control methods do not need to use any global information or additional sensory data. Two of the methods modify the environment of the agents and leave the internal functioning of the agents unchanged, whereas the third studied method does only a slight change to the behavior of the agents.

3.4.1 Agent Model

Similar as in Sendova-Franks and Lent (2002) the shape of an agent is modeled as a disc with radius ρ. The center of the disc (x_i, y_i) represents the position of agent i . Each agent has an actual direction of movement α_i, which is measured as the angle relative to the lower border of the rectangular simulation area. The point at position $(x_i + \rho \cos \alpha_i,\ y_i + \rho \sin \alpha_i)$ models the center of the agents head. From the center of the head every agent can sense obstacles within a range of distance σ, called sensing range (see Figure 3.15).

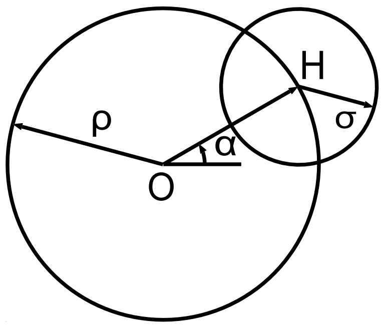


Figure 3.15: Agents are modeled as discs; ρ - radius; O - center of the body; α - direction of movement; H - center of the head; σ - sensing range

Agent i collides with agent j if agent j is within the sensing range of agent i, i.e., when the distance between the center of the head of agent i and the center of the body of agent j is smaller than $\sigma + \rho$. Similarly, an agent collides with the nest wall when the euclidean distance between the center of its head and the wall is less than the sensing range. For our experiments the body of an agent has radius $\rho = 0.01$, the sensing range is $\sigma = 0.006$.

The movement model used corresponds to the *repulsive ant model* from Scheidler et al. (2006). This model is a mixture of the *centripetal ant model* and the *avoiding ant model* from Sendova-Franks and Lent (2002), and was introduced to overcome the problem that agents get stuck near the focal point. Each agent i has a parameter $0 \leq \mu_i \leq 1$ that influences its moving behavior, modelling the behavioral differences between the agents. In Sendova-Franks and Lent (2002) and Scheidler et al. (2006) fixed values of μ_i were used for the experiments.

Here we investigate a movement model where the parameter μ_i can varies over time and models the actual state of an agent. The higher μ_i, the faster the agent can move. The value of μ_i is increased in case agent i visits a certain service area. The motivation behind this is that the agent gets new power or becomes unladen at the service station. During free movement of an agent its value μ_i decreases, modelling the use of power or the influence of the weight of collected items.

Movement when unobstructed

If there is no obstacle (wall or other agent) within its sensing range an agent moves and turns at each time step. The agent moves distance ν_i in direction α_i, i.e., $x_i \leftarrow x_i + \nu_i \cos \alpha_i$ and $y_i \leftarrow y_i + \nu_i \sin \alpha_i$. The values ν_i representing the velocities of the agents dependent on the internal parameters μ_i of the agents as follows: $\nu_i = (1 - \mu_i)\nu_s + \mu_i \nu_f$ where the parameters ν_s and ν_f, $0 < \nu_s < \nu_f < 1$ denote the slowest and the fastest velocity. They are set to $\nu_s = 0.0006$ and $\nu_f = 0.006$ in our experiments.

The agent changes its movement direction by $\alpha_i = \alpha_i + \theta_i$, where θ_i is the turning angle. For the calculation of this turning angle the clinotaxis model from Grünbaum (1998) is used: $\theta_i \leftarrow p_u(1 - \mu_i)\chi + p_b \mu_i \tau \cdot (1 - \cos(\phi_i))/2$ where $\chi = 15°$, $\tau = 30°$ are constants and the values of p_u and p_b, determining the direction of turning, are randomly chosen from $\{-1, 1\}$. The parameter ϕ_i denotes the angle between the actual moving direction α_i and the vector towards the service point. The larger this value is the stronger the agent will turn.

Agents with large value μ_i will be less affected by their ϕ_i as agent with small μ_i (see Fig. 3.17(b)). Therefore, for agents with small value μ_i the attraction to the service point is stronger than for agents with large value μ_i.

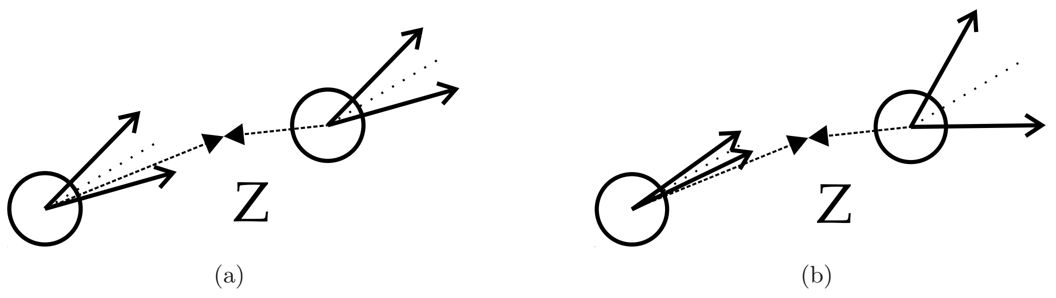


Figure 3.16: Effect of different values of the parameter μ_i on the turning behavior when unobstructed; Z is the service point; (a) for large μ_i there is only a slight difference between moving from or to the service point; (b) for small μ_i the turning angle becomes significantly smaller the smaller the angle between actual moving direction and the vector to the service point is

Movement when obstructed

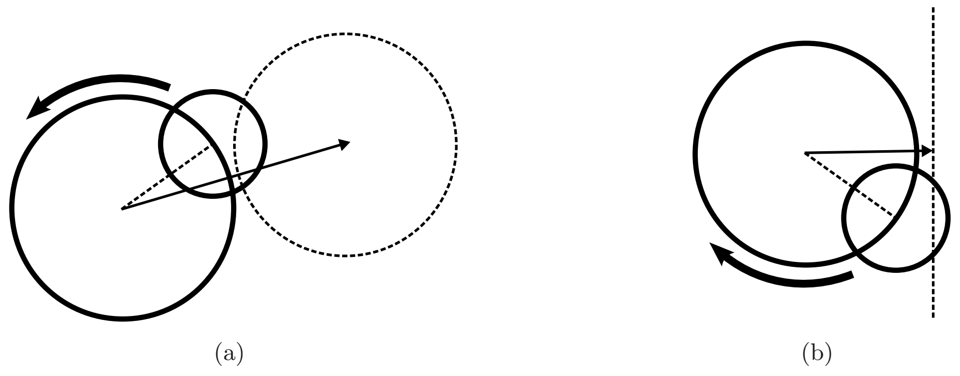


Figure 3.17: Turning behavior when colliding with an agents (a) or a wall (b)

If a wall or another agent is within the sensing range of an agent, it will not move, but only make a turn. It avoids the obstacle explicitly by turning into the same direction until it can move again. To determine the turning direction assume that agent i collides with agent j. The sign of the scalar product between the vector that is perpendicular to the vector of the moving direction of agent i and the vector from the center of agent i to the center of agent j determines the direction of turning: $\theta_i \leftarrow \text{sign}((-\sin\alpha_i, \cos\alpha_i) \cdot (x_j - x_i, y_j - y_i))U(0, \Theta_i)$, $\Theta_i = 60°$ is a constant. A collision with the wall is handled analogously.

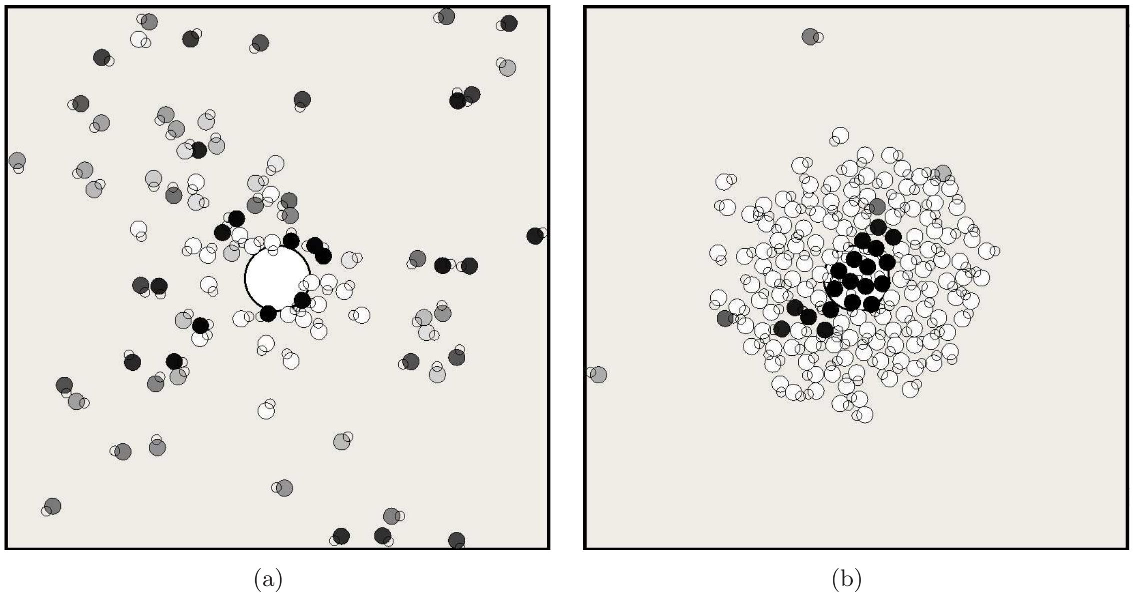


Figure 3.18: Distribution of the agents for different number of agents after 2000 time steps; (a) 90 agents; (b) 150 agents; the smaller the value μ_i of an agent the brighter is its color; the service area is the white circle in the middle

3.4.2 Emergent Congestion

The experiments took place in a quadratic area with side length 1. At the start of a simulation run the positions of the agents are distributed randomly with uniform distribution over this area. The values of the internal parameters μ_i are chosen randomly with uniform distribution between 0 and 1. In the center of the field there is a circular service area with radius 0.04 representing a service area. If an agents position (i.e., the center of its body) is within the service area its internal parameter μ_i is set to 1. If agent i moves (e.g., the agent is unobstructed) the value of μ_i is decreased by a fixed value 0.001 until $\mu_i = 0$. Observe that the smaller the value of μ_i is the slower moves the agent and also the higher is the attraction force to the service area.

Figure 3.18 shows the distribution of the agents after 2000 time steps for different number of agents. As can be observed depending on the system size there can occur a congestion situation. A system with 90 agents works without strong congestion at the service station. Agents with small value μ_i (bright color) tend to be close to the service station. Agents with large value μ_i are nearly randomly distributed over the whole field. This is different for a system with 150 agents. Here nearly all agents can be found close to the service station. Directly at the service station agents with large value μ_i are located. They cannot move away because the way is blocked by the agents with small value μ_i that try to move into the service area. As shown later, for this system the agents cannot do much useful work (if that means that the agents should ideally move over the whole field). Altogether, the observed congestion is an unwanted effect of the system that depends on colony size.

3.4.3 Congestion Control

To resolve a possible congestion of the agents at the service point we introduce three different congestion control methods. The goal of these methods is to resolve the congestion either by leaving the behavior of the agents unchanged or by changing the behavior of the agents only slightly but without need for introducing any new type of sensory information or global knowledge. The first two control methods C_{P} and C_{W} do not change the agent behavior and the third method C_{D} changes only the sensing range of the agents.

Control Method C_{P}

Control method C_{P} introduces two parallel walls next to the service station that form a pipe. The idea of this method is that agents that have visited the service station and have a high value μ_i might be able to move away from the service station through the pipe whereas only few of the agents that have a small value μ_i might use the pipe to move to the service station.

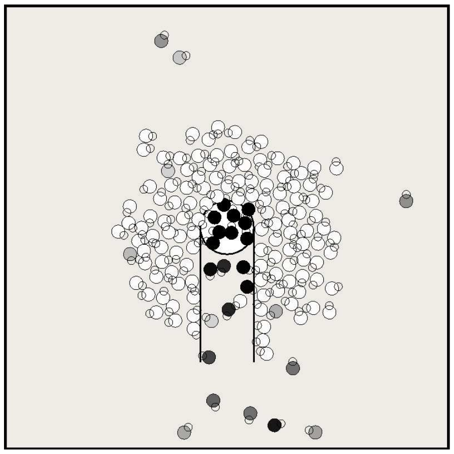


Figure 3.19: Distribution of agents for a system with 150 agents after 2000 time steps using congestion method C_{P}

An example of a pipe can be seen in Figure 3.19 where the distribution of the agents for a system with 150 agents after 2000 time steps using the congestion methods C_{P} is given. It can be seen that there is much less congestion by slow agents with small value μ_i within the pipe for method C_{P} than outside of the pipe next to the service area. It can also be seen that the agents with high value μ_i can move through the pipe.

Control Method C_{W}

Control method C_{W} is to introduce two additional walls on two sides of the service station. Each wall has a small opening in the middle that is next to the service station. The idea of this method is that slow agents with small value of μ_i might be forced to wait behind a wall and therefore do not block the service station. Hence, the agents that have visited the service station can move away from it. An example for this control method C_{W} can be seen in the middle of Figure 3.20.

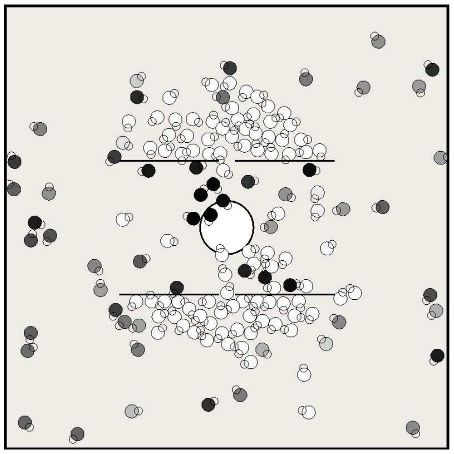


Figure 3.20: Distribution of agents for a system with 150 agents after 2000 time steps using congestion method C_{W}

For method C_{W} it can be seen that the congestion around the service area is much less compared to the system without congestion control. Agents with high value μ_i can be found in different parts of the field and not only next to the service area, as it was the case when no congestion control is used.

Control Method C_{D}

The third control method C_{D} changes the behavior of the agents slightly. Here the sensing range σ_i of agent i depends on the internal parameter μ_i. The sensing range is calculated as follows: $\sigma_i = 2\rho - 1.4\mu_i\rho$. The idea behind this method is that agents with a small value of μ_i that move to the service station have a larger sensing range and therefore leave some space when they are next to other agents. This space can be used by the agents that have visited the service station and therefore have a large value μ_i to move away from the service station.

For method C_{D} the distribution of the 150 agents after 2000 time steps is shown in Figure 3.21. The figure shows that at least some agents with high value μ_i that have

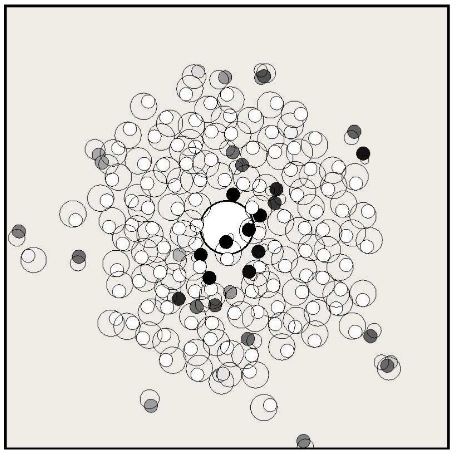


Figure 3.21: Distribution of agents for a system with 150 agents after 2000 time steps using a congestion method C_{D}

visited the service station can move away from it because the agents with small value μ_i leave some space between each other.

3.4.4 Comparison of the Methods

To compare the performances of the control methods with an uncontrolled system, over the first T time steps the total energy consumption of the system P_T is measured. This is done in the following way. Recall, if an agent reaches the service area its value μ_i is increased by adding the value $1 - \mu_i$ so that $\mu_i = 1$ holds afterwards. The sum over all values $1 - \mu_i$ for all i and every time when the value μ_i is increased can be seen as measure of the performance of the system. Since agents that move use energy and agents that can not move do not use energy the total energy consumption is a measure how freely the agents can move on average.

The given results were generated from 20 independent runs for every parameter combination, each lasting 10 000 time steps. Figure 3.22 shows the total energy consumption P_T for a system without congestion control and systems with congestion control. It can be seen that for a small number of agents, since no congestion occurs, the system without congestion control has the highest performance. This is no surprise because the congestion control methods slightly hinder the agents to move freely within the field when there is no congestion. But when the number of agents becomes larger than 100 the performance of the system without congestion control decreases very fast. For more than 130 agents this system has the worst performance. For a medium number of agents the system with method C_{W} is the best. But for a large number of agents this method is not much better than a system without congestion control. For a larger number of agents (more than 210)

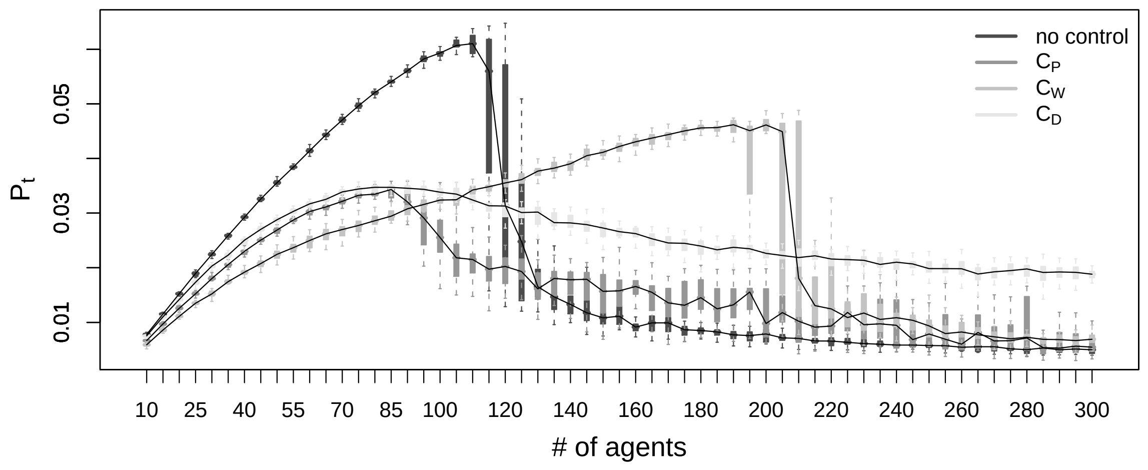


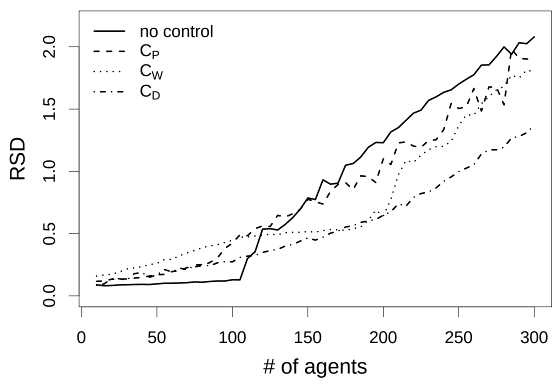
Figure 3.22: Total energy consumption P_T for different number of agents measured over 10 000 time steps for a system without congestion control and systems with the different congestion control methods

the system with method C_{D} is clearly the best. Method C_{P} is better than the system without congestion control for more than 135 agents but it is worse than the two other methods.

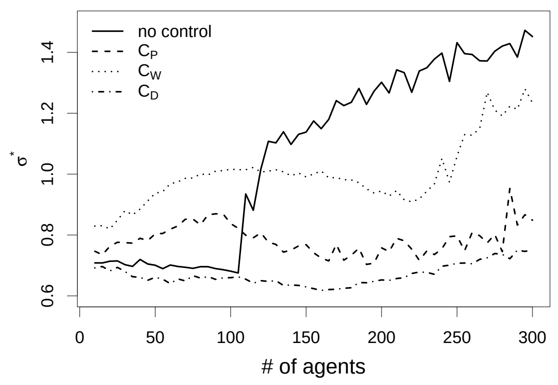
Beside the reduction of congestion, fairness for service is another important measure for the collective behavior of agents. In the considered system, e.g., the waiting times for service have to be similar. We measured the fairness of the system in two different ways. First, at the end of a given time interval of length T for every agent the total amount of values that have been added to μ_i for all its visits of the service station is measured. Then the relative standard deviation (RSD) of these values for all agents has been taken as a measure for the fairness of the system (the lower the variance means the more fair the system is).

The behavior of the systems with respect to this fairness measure is shown in the left part of Figure 3.23. It can be seen that the system without congestion control is most fair for a small number of agents (less than 110 agents). For a larger number of agents the system with the C_{D} method is the best.

The second measure of fairness is defined as follows. Let $\tau(T)$ be the mean waiting time of the agents where the waiting time of an agent is defined as the length of the time interval from the time when its internal parameter (μ_i) becomes zero until the time when it reached the service area (measured over a simulation run over T time steps). Let $\sigma(T)$ be the standard deviation of these waiting times. A dimensionless measure for the fairness is then by $\sigma^*(T) = \sigma(T)/\tau(T)$. Note, that for this measure only the waiting times of the agents that reached the service point are considered. Hence, a congested system may still



(a)



(b)

Figure 3.23: Fairness for different number of agents measured over 10 000 time steps for a system without congestion control and systems with the different congestion control methods; first fairness measure RSD (left), second fairness measure σ^* (right)

be fair, if there is only a small subset of agents that are served and these agents have similar waiting times.

The behavior of the systems with respect to the second fairness measure is shown in the right part of Figure 3.23. It can be seen that the system the C_{D} method is most fair (independently of the number of agents). For small number of agents (less than 110) the system without congestion control is the second most fair system. For larger number of agents the system with method C_{P} is the second best.

3.5 Summary

In complex self-organizing technical systems consisting of many autonomous components emergent effects may occur that are neither wanted nor have it been intended or foreseen in the design phase. To make the systems reliable it is necessary to take care of this problem. A general way is to introduce feedback loops to make the systems self-adaptive.

We introduced another approach for controlling or preventing (unwanted) emergent effects, called *swarm controlled emergence*. This approach uses a swarm of control components / agents introduced additionally into the system. As a proof of concept, we tested our approach by using it to prevent the emergent clustering in the well known model of the clustering behavior of ants. Three different types of control agents have been investigated for this system. Namely, random $\mathcal{AC}$-agents, inverted neighborhood $\mathcal{AC}$-agents, and reverse $\mathcal{AC}$-agents. The inverted neighborhood $\mathcal{AC}$-agents could not prevent the clustering process which shows that simply reversing a part of the standard agents behavior is not enough to prevent the clustering. Using a moderate number of individuals and well chosen parameter values the random and the reverse $\mathcal{AC}$-agents can prevent the clustering successfully. An interesting effect occurred when a medium number of certain reverse $\mathcal{AC}$-agents was introduced into the system. In this case an increased strength of the clustering effect was observed. This is an interesting point for two reasons. First, it shows that its no trivial task to design a control swarm, since its inference with the system may lead to new or even stronger negative emergent effects. Second, concerning the special case of ant based clustering, the investigations show that a system of two types of agents can lead to a stronger clustering than a system with only one type of agents. A fact which may be used for the design of clustering algorithms.

In the last part of the chapter we have studied how to control emergent congestion effects in agent systems with ant inspired movement rules. Three methods which do not or only slightly change the behavioral rules of the agents were proposed. It was shown experimentally that the proposed methods can significantly reduce the congestion and are also fair for systems with a large number of agents.

4 Specialization in Organic Support Systems

This chapter introduces Organic Support Systems. These systems take care of the execution of necessary support and system care tasks in Organic Computing Systems. Organic Support Systems are based on autonomous collaborating components called helpers. We assume that helper components are equipped with reconfigurable hardware in order to adapt to the actual need of the supported system by specializing for the required types of support tasks. The specialization comes with costs because reconfiguration operations take time and configurations supporting certain types of tasks lead to worse performance on other types. Three different aspects of the self-organized and decentralized organization of the helper system are studied: First, inspired from models of task allocation in social insects, we introduce a mechanism for allocating tasks in Organic Support Systems. Then we investigate the stability and the performance of ant queue inspired methods for support tasks partitioning. Finally, as a third aspect, we introduce support task allocation in networks based on a decentralized clustering algorithm.

4.1 Introduction

In the following we study models of Organic Computing Systems which consist of two types of components. A schematic view of the system model is given in Figure 4.1. In the upper part of the figure the worker components are depicted. Worker components model the part of the Organic Computing System that is responsible for the normal work of the system. We do not make many assumptions regarding these components and the work they do, aside from that they need certain types of service/support task to be executed from time to time. As usual for Organic Computing systems, the workers are autonomous, adaptive, and collaborating. The structure of the worker system can be highly dynamic, as workers may disappear or new worker enter the system.

The organic support system depicted in the lower part of Figure 4.1 executes the needed service tasks for the worker system. It consists of so called *helper components* (or just called *helpers*). Helper possess reconfigurable hardware in order to be able to adapt to the actual needs of the worker system. Like the workers, the helpers are assumed to be autonomous and loosely coupled, i.e., there is no predetermined interaction pattern. The structure of the organic support system is assumed to be dynamic, that is, helpers can disappear or

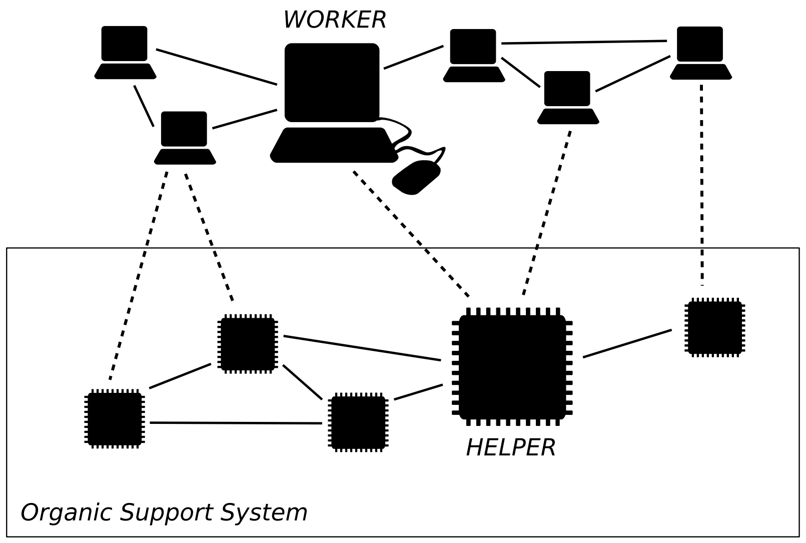


Figure 4.1: Model of the considered computing systems; the Organic Support System consisting of loosely coupled autonomous helper components executes service tasks for the worker components

new helpers can enter the system. Thus, a centralized decision making is not feasible. The decisions about which tasks to execute and how to configure the reconfigurable resources, i.e., how to specialize, are made by the helper components their self.

In this chapter we discuss three different aspects of the decentralized service task allocation in Organic Support Systems. In Section 4.2 we assume that the reconfigurable resources of the helper can be divided into parts of equal size that can be configured independently for different types of service tasks. The more resources configured for a certain task type the faster tasks of this type and the slower tasks of other types can be executed on the helper. We apply local, social insect inspired task allocation and specialization strategies for the helpers and investigate the performance of the modelled systems.

A second aspect comes into play, when the tasks are large and the helpers have not enough resources to execute a whole task at once. In such a case the task can be divided into parts which can be executed by different helpers successively. Inspired by task partitioning methods found in real ants, we introduce and investigate local strategies for the helpers to decide to which subtask to specialize.

As a third point we use the decentralized packet clustering algorithm introduced in Chapter 2.2 to allocate service tasks, if the workers and the Organic Support System are connected via a router based network. The clustering algorithm groups the support task requests based on similar resource requirements. The helpers specialize to a certain group of tasks and in this way the reconfiguration costs can be kept small.

4.1.1 Reconfigurable Hardware

We assume that the helper components exhibit reconfigurable hardware in order to be able to adapt to the worker system. Reconfigurable hardware devices are devices in which the functionality of the logic gates and the connections between them are customizable at run-time. This special type of hardware fills the gap between hardware and software, achieving higher performance than software, while maintaining a higher flexibility than hardware. The most common type of reconfigurable hardware devices are SRAM-based field-programmable gate arrays (FPGAs). They consist of an array of computational elements and routing resources, whose functionality is determined through programmable configuration bits. In order to change the functionality, i.e., to reconfigure the hardware, the new configuration bits have to be transmitted onto the FPGA. For a survey on reconfigurable computing, see e.g., Compton and Hauck (2002).

Please note that the term "task" in the field of reconfigurable computing usually refers to an implementation of a concrete function, for instance the Fast Fourier Transformation (FFT) and the term "task allocation" refers to the question of a good placement of the implementing circuits on the reconfigurable chip (see, e.g., Lu et al., 2009). This is not the problem we are dealing with in this chapter. In our context a task is a concrete instance of a problem which has to be solved, i.e., to calculate the FFT for some concrete data. Task allocation, as used here, means to find an appropriate configured helper component, i.e., a helper implementing circuits for executing the task (efficiently).

Systems of interconnected reconfigurable devices, so called multi-FPGAs, are used mainly for rapid-prototyping of complex ASIC (Application Specific Integrated Circuit) designs and akin applications. The structure of multi-FPGA systems is static and the main research topic is how to partition and distribute a given hardware design over multiple FPGAs. A problem more similar to the problem we are dealing with in this chapter is the problem of load-balancing in systems of interconnected FPGAs (Bakos et al., 2006; Kindratenko et al., 2007). On the other hand compared to the fixed system structure and implementations of the tasks in the multi-FPGA field, Organic Support Systems are highly dynamic systems of interconnected reconfigurable components. Tasks can be implemented several times in the system and can even be implemented in different ways. To the best of our knowledge, to investigate the problem of a decentralized, self-organized task allocation in dynamic systems of reconfigurable components is a new field of research.

The aim of this work is to consider this problem from an abstract complex system point of view. We investigate an abstract model of the worker system and the needed support tasks. No assumptions about how the configuration data gets to helper are made and we abstract from questions concerning the transfer of input/output data and the implicated latencies. The considered helper components do not use a specific type of reconfigurable hardware, e.g., a FPGA from a particular vendor, instead a simple cost model for the reconfiguration and execution times is used.

In the following we will briefly sketch some points of this model. Specific properties of the used models are given in the particular sections, if needed. Whereas in previous implementations of reconfigurable hardware a reconfiguration operation had to specify the configuration bits for the whole chip, today's reconfigurable hardware devices usually are partially reconfigurable. That is, it is possible to reconfigure only a part of the chip and only the configuration data for the changed part has to be transmitted. In our model of the helpers we therefore assume the cost of a reconfiguration depends linear on the fraction of changed resources.

For the cost of task execution we also assume a linear cost model, i.e., the more resources (chip area) can be used for a task, the faster it can be executed. Clearly, this simple cost model does not fit for every real task implementation, since not every problem can be parallelized in arbitrary granularity. But it is a reasonable cost model, since using techniques like loop unrolling and tree height reduction can lead to a linear area-time dependency (see, e.g. Ferrandi et al., 2007) and it has been shown that treating the area-time trade-off for FPGA designs as multi object optimization problems often leads to quasi linear Pareto fronts (Holzer et al., 2007).

4.1.2 Models of Division of Labour in Social Insects and their Application

Social insects organize their work in sophisticated ways. The observed principles are interesting for the design of organic computing systems, because the organization of work in insect colonies shows many desirable properties for technical systems. The organization is based on local communication only and no central control exists. Therefore there is no single point of failure and the system is robust against the loss of single individuals. Colonies of social insects can adapt to a changing environment, e.g., to the loss of individuals or to a changing task composition, in an efficient way.

A colony has to perform a number of tasks, such as feeding the brood, foraging for resources, maintaining the nest and defending the colony. The allocation of individuals to these different tasks requires continuous adjustments in order to response to external changes, like the amount of food available and internal changes, like changing mortality of foraging individuals.

To explain the adaptive self-organizing task division in social insect colonies, several theoretical models have been proposed. Stimulus-threshold models are one standard type of models that are used in the literature (see, e.g. Beshers and Fewell, 2001; Bonabeau et al., 1996, 1998; Theraulaz et al., 1998, 1991). In these models each individual has a personal threshold value for each task. This threshold determines the preference of the individual to start to work for that task. In addition, for every task a stimulus value exists that determines the task's necessity to be done. The probability that an individual works for a task depends on the relative size of its threshold value for the task and the stimulus value of the task. The lower the threshold value and the higher the stimulus value, the more likely the individual starts to work for the task.

Varying response thresholds over the individuals lead to a division of the different tasks over the individuals, since certain individuals are more likely to work for certain tasks. The variation in the response thresholds of the individuals is partly caused by genetic differences (Waibel et al., 2006), but also by the fact, that the individuals are able to learn in a certain extend. To model a simple form of learning the stimulus-threshold models have been extended to the so called threshold reinforcement models, where the thresholds can change over time. It is assumed that the threshold of an individual for a task decreases when the individual works for the task and vice versa it increases when the individual does not work for the task (Bonabeau et al., 1998; Theraulaz et al., 1998). Thus, individuals can specialize over time to certain tasks.

Different functions, determining the probability of an individual to engage in a task as a function of the task stimulus S, are proposed in the literature. Most often functions of the form

$$P = \frac{S^n}{S^n + T^n} \tag{4.1}$$

are used, where T denotes the personal threshold of the individual for the task and usually the parameter $n = 2$ is chosen. Clearly, for $S \ll T$ the probability of engaging task performance is close to 0, and for $S \gg T$ this probability is close to 1.

Many applications of response threshold models in technical systems exist, for example, in scheduling (Cicirello and Smith, 2003, 2004; Kittithreerapronchai and Anderson, 2003; Nouyan et al., 2005), robotics (Agassounon and Martinoli, 2002; Jones and Mataric, 2003; Krieger et al., 2000; Krieger and Billeter, 2000; Labella et al., 2006), sensor networks (Haboush and Shrimpton, 2005), mail retrieval problems (Goldingay and Mourik, 2008; Price and Tino, 2004), and in multi agent systems (Ferreira et al., 2005).

Another important behavioral mechanism in ants is the so called *task partitioning*. To partition a task means, that one particular task is done by different individuals. For example, it can be observed that the task of food collection is shared by workers that collect the food (forager ants) and workers that use or store the food (receiver ants). A direct transfer of material between individuals happens. The time taken to meet a transfer partner, also called queueing delay, is a very important factor that influences the efficiency of the system. A model — called ant queue model — for the task partitioning behavior in ants is proposed in Anderson and Ratnieks (1999a,b). It is shown by simulations that queuing delays occur even when the proportions of foragers and receivers in the colony are optimal meaning that the work capacities of these two groups are equal. It is further shown that the queuing cost may, potentially, act to select against task partitioning in small-colony species, thereby restricting task partitioning with direct transfer to species with large colonies.

Pini et al. (2009) uses task partitioning as a way to reduce the interference between cooperating robots in a spatially constrained harvesting task. The task of delivering prey objects to a home zone is divided into two subtasks. Robots working on the first subtask harvest prey objects from a source area. Within a transfer zone they pass them to robots working on the second subtask, which is to finally store the objects in the home zone. A simple, threshold based method to allocate the individuals to the subtasks is presented. It is shown that task partitioning and thereby the avoidance of physical interference between the robots, can increase the system performance.

An interesting similarity of the task division model and the task partitioning model is that specialization of the agents can increase the throughput of the system, but changing the specialization comes with a cost. The difference between the models is that in the task division models the agents specialize to different types of tasks, whereas in the task partition model the agents specialize to a part (subtask) of a task. In the following two sections we first investigate a self-organized task allocation method for Organic Support Systems that partly relies on the task division model. Thereafter the model of task partitioning is used to inspire a method for the organization of task partitioning in Organic Support Systems.

4.2 Self-Organized Task Allocation

In the following we will propose and investigate methods for the self-organized allocation of service tasks in Organic Support Systems. Self-organized means that no central control component allocates the tasks on the appropriate helpers. Instead, the helpers themselves decide about the acceptance of a requested service task. This is in analogy to social insects, where individuals have to decide wether to engage in an encountered task as well. In some species of social insects individuals can become specialists for a task, adapt, and get better in performing that specific task. Analog to this phenomenon, in case of accepting a task, a helper reconfigures in order to increase the amount of computing resources that are available for the type of task accepted.

Here basic scenario is considered: Workers send requests for service tasks to the support system. These request are randomly offered to helpers until they are accepted. If the helpers are too restrictive in their task acceptance, it can happen that it takes a long time

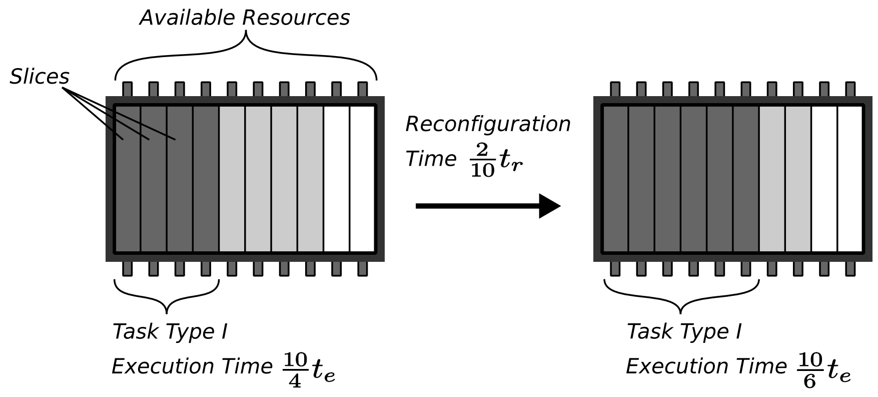

Available Resources
Slices
Reconfiguration Time $\frac{2}{10}t_r$
Task Type I
Execution Time $\frac{10}{4}t_e$
Task Type I
Execution Time $\frac{10}{6}t_e$

Figure 4.2: The execution time of a certain type of tasks depends on the number of slices configured for that type; reconfiguration time depends on the number of reconfigured slices

until a task is executed. If, on the other hand, the decision to accept a service task is made to frivolously, it can happen, that the task is executed on a bad configured helper. This results in a long time until the execution of the tasks as well. Our aim is to create systems that deal with the tradeoff between these cases.

4.2.1 Model of the Helper Components

The reconfigurable resources of a helper, i.e., the area on its reconfigurable device, is divided into q parts of equal size, called *slices*. Service tasks are of different types and each slice can be configured and work for exactly one of these types. A helper can only work at one task at a given time and the number of slices allocated for the task type determines how fast the task can be executed. The more resources configured for a certain task type the faster tasks of this type are executed by the helper. The execution time of a task is given as $\frac{q}{k} \cdot t_e$, where $t_e > 0$ is the execution time of a task on a helper which has all resources allocated to the task type and k is the number of slices configured for the task ($1 \leq k \leq q$).

The slices of a helper can be reconfigured independently from each other. A reconfiguration (operation) has the effect that the type of tasks that can be executed on a slice is changed. When reconfigurating a helper can change the associated task type of any number of slices. The time of a complete reconfiguration of a helper is $t_r > 0$ and the time to reconfigure k slices is $\frac{k}{q} \cdot t_r$.

In Figure 4.2 an example of the helper model is given. The resources of the depicted helper are divided into $q = 10$ slices. The helper has configured four slices to work for task type one. Therefore, to execute a service task of type one takes $\frac{10}{4}t_e$ time. If the helper allocates two more slices for task type one this will take $\frac{2}{10}t_r$ time for the reconfiguration.

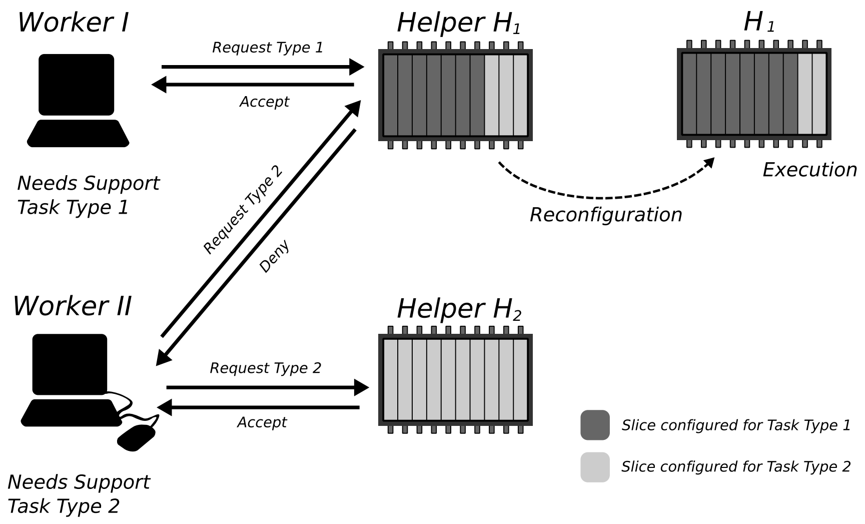


Figure 4.3: A typical scenario in the worker helper system

With the new configuration of six slices for task type one the helper only needs $\frac{10}{6}t_e$ for the execution of a service task of type one.

We denote the helpers of the system with $H_1, \ldots, H_n$, where n is their total number. A helper H, that has a proportion of $\geq 1/2$ of its slices configured for tasks of type i is called *specialized* for tasks of type i. In this chapter we assume that only two types of service tasks (type 1 and type 2) exist. Let $s(H) = i$ if H is specialized to tasks of type i, $i \in \{1, 2\}$. H is *fully specialized* for tasks of type i when all slices are configured for this type of tasks. The *degree of specialization* of a helper is the relative number of slices that are configured for the type of tasks the helper is specialized to. Let s_j, $j \in [1 : n]$ denote the degree of specialization of helper H_j. Let s_{ij}, $i \in \{1, 2\}$, $j \in [1 : n]$ denote the relative number of slices that helper H_j has configured for task type i.

4.2.2 Model of the Computing System

Let m be the number of workers. At each time step a worker needs the assistance of a helper with some probability. This probability is called the *request rate* and denoted by $0 \leq r \leq 1$. The *relative request rate* $0 \leq p \leq 1$ is the probability that a service request is of type one. The relative request rate for service task type two is $(1 - p)$. A worker that needs servicing of type i searches for a helper and requests a service task of type i, $i \in \{1, 2\}$. If the request is accepted the service task is executed by the helper. If the request is not granted the worker will continue to search for a helper. We assume here searching means that a random helper is contacted. The first request that a worker does when it needs service is called *initial requests*. The communication with the helper takes time $t_c \geq 0$ (no matter whether the request was successful or not).

Figure 4.29 depicts a typical scenario in the system. Worker I needs a service task of type one to be executed and sends a service request to helper H_1. H_1 accepts the request, reconfigures one more slices to task type one, and executes the task. Worker II needs a task of type two to be executed and sends the request to H_1. This time H_1 denies the request, since it has only few slices configured for task type two. Worker II sends the request again and helper H_2 accepts to execute the task.

4.2.3 Task Acceptance and Reconfiguration Strategies

The task acceptance strategies and the reconfiguration strategies for the helpers are presented in this section. The task acceptance strategy determines if a helper accepts a offered service task. In case the task is accepted the reconfiguration strategy determines how the helper reconfigures before executing the task.

Optimal Specialization

We first determine the optimal percentage of slices a helper should configure for the two task types for given fixed relative request rates p for task type one and $1 - p$ for task type two.

Let us assume that the (normalized) run time for a task on a fully specialized helper is $t_e = 1$. Then the run time for a task of type i on a helper H_j which has a fraction of s_{ij} of its slices configured for i is $1/s_{ij}$. The expected mean runtime for a task on a helper H_j is calculated from the expected runtimes of both types of tasks:

$$\frac{p\frac{1}{s_{1j}} + (1-p)\frac{1}{s_{2j}}}{2}.$$

Therefore, the optimal percentage $g(p)$ of slices configured for task type one, leading to the lowest expected mean task runtime, is:

$$g(p) = \frac{p - \sqrt{p - p^2}}{2p - 1}. \tag{4.2}$$

A plot of this function is given in Figure 4.4. The optimal percentage of slices configured for task type two is $1 - g(p) = g(1 - p)$, respectively.

Task Acceptance Strategy

An important aspect of the system is the strategy that is used by the helpers to decide whether a request for service should be accepted or not. A helper that gets a request always accepts the request when it is specialized for the corresponding task type, i.e., it has at least $q/2$ of its slices configured for the type: $s_{ij} \geq 0.5$. This is because with this

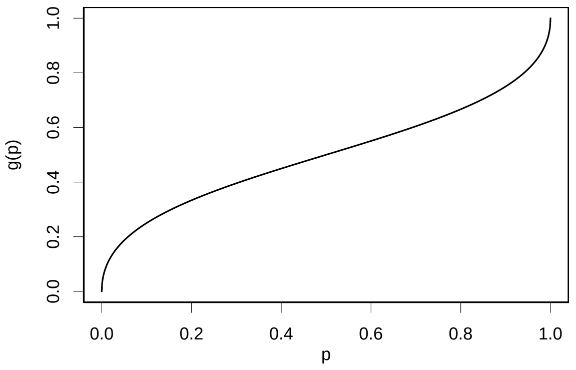


Figure 4.4: Optimal specialization level for one helper depending on the relative request rate

configuration the time to execute the task is at most twice as high as the execution on a fully specialized helper. Otherwise, the probability that it accepts the request depends on a personal threshold value, a stimulus value, the degree of specialization, and the relative request rate of that type of tasks. The *stimulus value* for a type of tasks is the number of tasks of this type minus the number of tasks of the other type counting all tasks that are actually requested by the workers. Let T_{ij}, $i \in \{1, 2\}$, $j \in [1 : n]$ denote the threshold of helper H_j for task type i and S_i the stimulus of task $i \in \{1, 2\}$. Incorporating Formula 4.1 from the threshold response model of task allocation in social insect, the probability that helper H_j accepts a request for task i is defined as

$$P := \begin{cases} \min\left\{1, f_i(p, s_{ij}) + \frac{S_i^2}{S_i^2 + T_{ij}^2}\right\} & \text{if } s_{ij} \leq 0.5 \\ 1 & \text{else} \end{cases} \tag{4.3}$$

where the function $f_i(p, s_{ij})$ is defined in the following.

For defining the function f_i that is used in this formula, we consider the case of a single helper with a fixed configuration, constant service request rates and stimulus values for both tasks of zero. We now define f in a way, that H_j rejects a fraction of tasks (of the type it is not specialized for) that it has an optimal configuration for the resulting relative request rates of both tasks. Without loss of generality assume that H_j is specialized for task type two (i.e., $s_{2j} > 0.5$). Since $S_2 = 0$ it follows from Formula 4.3 that all tasks of type two are accepted and that tasks of type one are rejected with probability $f_1(p, s_{1j})$. The function f is determined such that the relative rates of tasks that are accepted by H_j are optimal for its current specialization, if this is possible. Otherwise, the relative number of requests for task one is too low for an optimal degree of specialization. In this case all tasks of type one are accepted. Observe, that the relative request rate accepted for task type one is $pf_1(p, s_{1j})/[pf_1(p, s_{1j}) + (1-p)]$. Using Formula 4.2 assuming that for this rate the actual specialization s_{2j} is optimal

$$g(pf_1(p, s_{1j})/[pf_1(p, s_{1j}) + (1-p)]) = s_{1j}$$

leads to

$$f_1(p, s_{1j}) = (p/(1-p)) \cdot s_{1j}^2/(1-s_{1j})^2.$$

The case of H_j is specialized to task type one is handled accordingly. Clearly, this definition of f is heuristic and not necessarily optimal for a computing system with several helpers.

Reconfiguration Strategies

In the first proposed reconfiguration strategy a helper that performs a service task always performs a reconfiguration operation so that the number of slices that can execute the corresponding task type is increased by one (unless all slices have already been configured for the corresponding type of tasks). We call this the *1-slice reconfiguration strategy*.

A possible problem of the 1-slice strategy is that the execution time of a service task is very long when only few slices execute it. Therefore we also studied a variant of the 1-slice strategy that is different for the case that a request is accepted when the helper is not specialized for the task. In this case the helper reconfigures itself so that half of the slices are configured for the accepted type, i.e., the helper immediately specializes to the task type, before the execution starts. We call this the *1+half-slice reconfiguration strategy*.

4.2.4 Analysis of a Two Helper Support System

In this section we analyze a computing system with two helpers theoretically. The aim of this section is to determine the configurations and strategies for rejecting requests, in order to minimize the expected total time needed for execution and communication. To make the analysis possible, a few changes of the standard computing system model are made for this subsection:

i) A worker whose initial request for service was rejected, sends the request to the other helper which must accept the request. This is different from the standard model where a rejected request is repeated by sending it to a randomly chosen helper (possibly to the same helper again).

ii) The relative number of slices s_{ij} of helper j configured for task type i can be any real value in $[0, 1]$. This is different from the standard model where the this is a discrete parameter with values in $\frac{0}{q}, \frac{1}{q}, \frac{2}{q}, \ldots, 1$ where q is the number of slices of a helper.

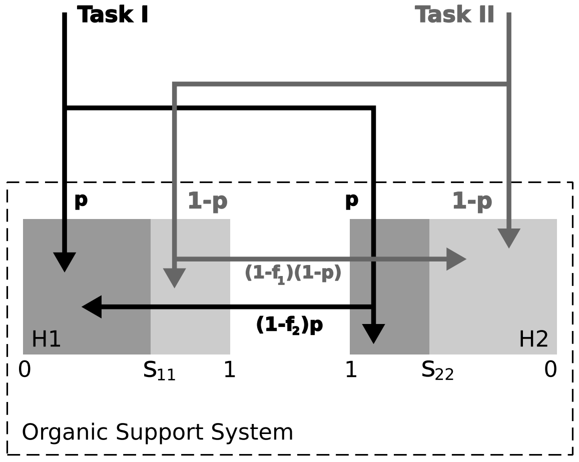


Figure 4.5: Relative request rates for the analysed two-helper support system

iii) The system is static in the sense that the configuration of a helper can not be changed and the relative request rates of the tasks are fixed.

It is assumed that the probability that a worker which needs service first contacts helper H_j is the same for $j = 1$ and $j = 2$. Assume that requests for tasks of type i are always accepted by helper H_j, $j \in [1, 2]$. Recall that s_{jj} is the proportion of slices configured for tasks of type j of helper H_j. Let f_j be the fraction of accepted requests of type i, $i \neq j$, $i, j \in \{1, 2\}$, by helper H_j, depending on the specialization level s_{jj} . Note, that the execution time of each service task on a fully specialized helper is assumed to be 1.

The expected total execution time e for both tasks is

$$e := \left[\frac{p}{s_{11}} + \frac{p(1-f_2)}{s_{11}} + \frac{(1-p)f_1}{1-s_{11}}\right] + \left[\frac{1-p}{s_{22}} + \frac{(1-p)(1-f_1)}{s_{22}} + \frac{pf_2}{1-s_{22}}\right].$$

This time consist of the expected execution time p/s_{11} and $(1-p)/s_{22}$ for the requests of type j that arrive directly at H_j, the expected execution time $p(1-f_2)/s_{11}$ and $(1-p)(1-f_1)/s_{22}$ for the requests of type i that have been rejected by H_j $(j \neq i)$, and the expected execution times $(1-p)f_1/(1-s_{11})$ and $pf_2/(1-s_{22})$ for the requests of i that arrive directly at H_j $(j \neq i)$ and are accepted.

Give the cost of one communication operation c the expected communication costs ω are

$$\omega := 2c + c \cdot [\ p(1-f_2) + (1-p)(1-f_1)\ ]\,.$$

They consist of the costs for the first communication for each request plus the costs for the second communication that is necessary when a request is rejected. The expected total execution and communication time for a task is $C = e + \omega$.

C can be rewritten as

$$C = f_1 \cdot A + f_2 \cdot B + D,$$

where the terms A, B and D only contain the variables p, c, s_{11} and s_{22}. It easy to see, that for any values of these variables in order to minimize C, it must hold that $f_i = 0$ or $f_i = 1$ for $i \in \{1, 2\}$, depending on the signs of the terms A, B and D. This means helper H_j either rejects all request for tasks of type i, $j \neq i$ or it accepts all these tasks. The following four cases can occur:

i. $f_1 = 0, f_2 = 0$:

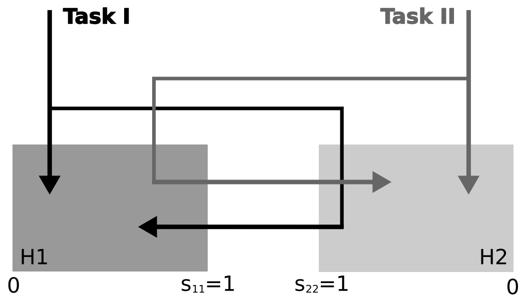


Figure 4.6: Both helpers always reject.

In this case each helper H_i rejects all requests for tasks of type j, $j \neq i$. To achieve a minimal total execution and communication time it is best when both helpers are fully specialized, i.e., $s_{ii} = 1$, $i \in \{1, 2\}$. Then it follows that $C := 2 + 3 \cdot c$.

ii. $f_1 = 1, f_2 = 1$:

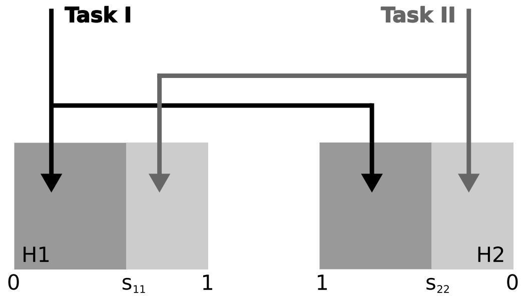


Figure 4.7: Both helpers always accept.

Both helpers accept all requests and no additional communication occurs. For arrival rate p follows the lowest expected total execution and communication time can be reached with specializations $s_{11} = 1 - s_{22} = g(p) = (p - \sqrt{p - p^2})/(2p - 1)$, as shown in Section 4.2.3. Therefore $C = 2(2p-1)^2\sqrt{p(1-p)}/(p - \sqrt{p(1-p)})(p - 1 + \sqrt{p(1-p)}) + 2c$. Note, that $C = 4 + 2c$ for $p = 1/2$ and $\lim_{p=1} C = 2 + 2c$.

iii. $f_1 = 1, f_2 = 0$:

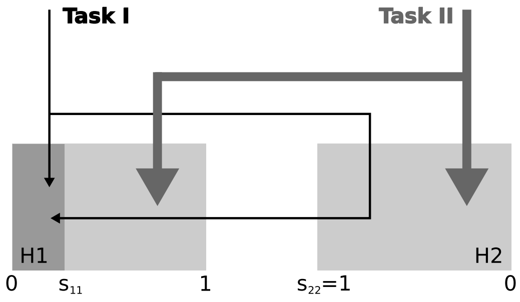


Figure 4.8: Helper H_1 always accepts and Helper H_2 always rejects.

In this case H_2 rejects all request for tasks of type one. Therefore it is optimal when it is fully specialized for task type two ($s_{22} = 1$). For H_1 this leads to arrival rates of $2p$ for requests of type one and $(1 - p)$ for requests of type two. Hence, the relative rate for requests of type one is $2p/(p+1)$ and the optimal value for the specialization of H_1 is $s_{11} = g(2p/(p + 1))$. The formula for the resulting expected total execution and communication costs is omitted because it is lengthy (the cost values are shown in Figure 4.9(a)).

iv. $f_1(.) = 0$ and $f_2(.) = 1$:

Analogous to case iii.

Figure 4.9(a) shows the expected total execution and communication costs for all four cases. In Figure 4.9(b) for each of the treated cases (i)-(iv) the regions of the parameters c and p are depicted where this particular case is optimal, i.e., leads to the minimal expected total execution and communication costs. For large communication costs c and values of p that are neither very small nor very large, it is optimal when all requests are accepted (ii). If the communication costs are small and the values of p are not too extreme then it is optimal when each helper rejects one type of requests (i). For large (small) values of p it is optimal when H_1 rejects (respectively accepts) all requests of type 2 and H_2 accepts (respectively rejects) all requests of type 1 (iii and iv).

In Figure 4.10 the optimal specialization level s_{11} of helper H_1 is shown for different relative arriving rates p for tasks of type one and communication costs c. In case (i) and in case (iv) H_1 is fully specialized for tasks of type one. For $p = 0.5$ and $c > 2$ exactly half of the slices are configured for both of the task types.

Note, that in a dynamic situation where the relative request rates or the communication cost change, there can be a strong difference in specializations necessary in order to have the lowest possible expected total execution and communication costs. For instance, the optimal specialization level of H_1 for $c = 2.25$ and $p = 0.75$ is 0.644, whereas it is 1

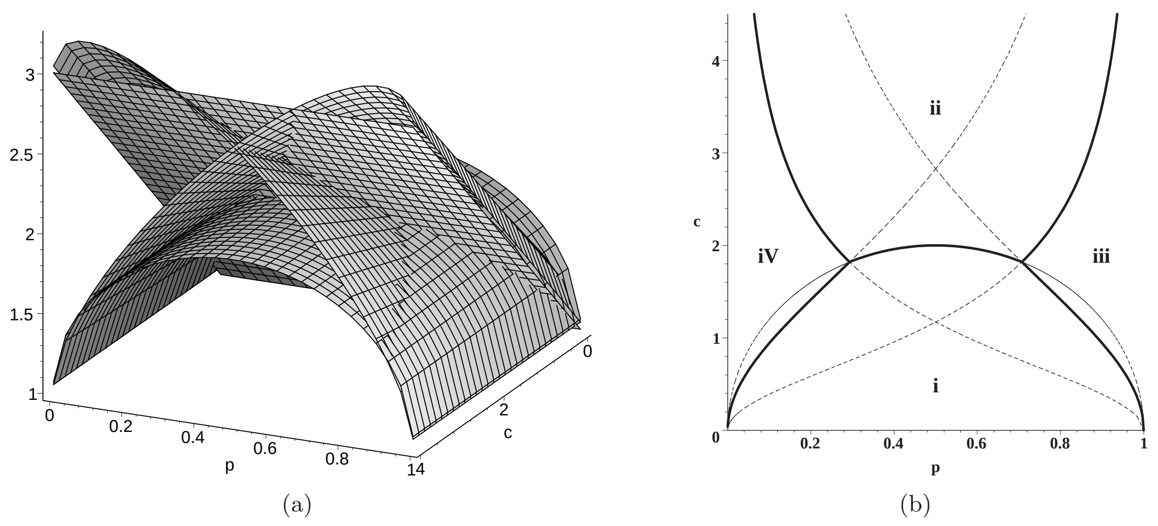


Figure 4.9: Two helper system: hyperplanes corresponding to cases (i)-(iv) show the expected total execution and the additional communication costs divided by the number 2 of helpers

when changing to $p = 0.8$. Therefore, when dealing with dynamic situations and changing environments in real systems, it has to be considered carefully if its necessary to always configure to the optimal configuration. This may lead to larger overall costs, especially in the case of large reconfiguration costs.

4.2.5 Experiments

If not stated otherwise the following parameter values have been used for the simulation of the computing system. The execution time of the two used types of tasks is t_e and the communication cost is $t_c = \frac{1}{10}t_e$. The number of workers was set to $m = 100$ and the number of helpers to $n = 10$. Each helper has $q = 10$ slices and the time to reconfigure all slices of helper is ten times longer than the time for execution, i.e., $t_r = 10t_e$. For all tasks and helpers the same threshold value $T := T_{ij} = 100$, $i \in \{1, 2\}$, $j \in [1, 10]$ was used. All results are averaged over 20 runs. The standard reconfiguration method is the 1-slice strategy.

The average time from the initial requests to the end of the execution of the service tasks is called *sojourn time* and the absolute number of service tasks that have been finished in a certain time is called *throughput*. We refer to the computing system with self-organized task allocation as introduced here just as the *self-organized system*. The simulation results of the self-organized system are compared to a system with static helpers. This system is called $\mathcal{S}$-system and consists of helpers which have allocated an equal number of their

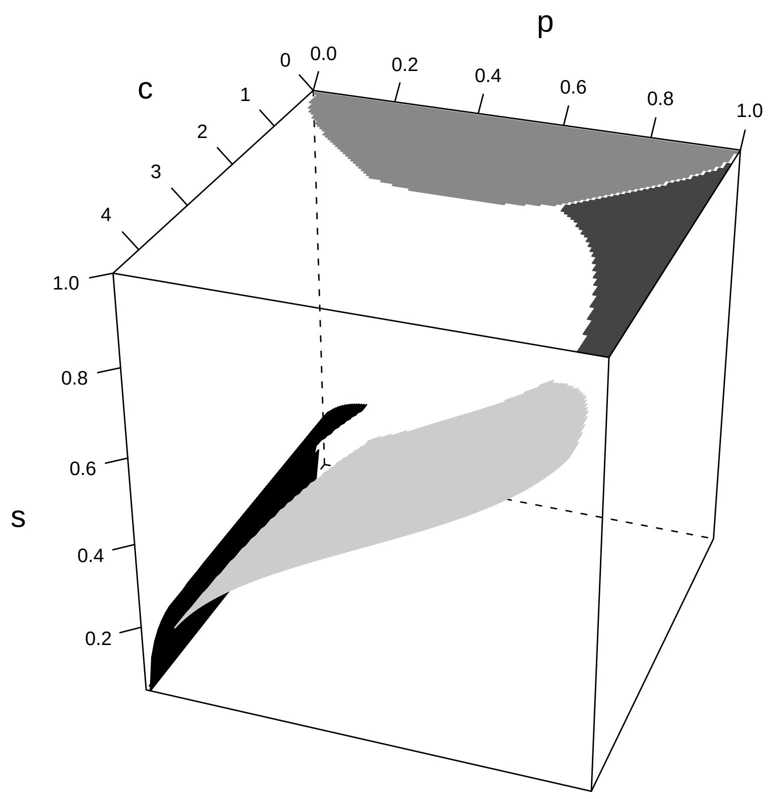


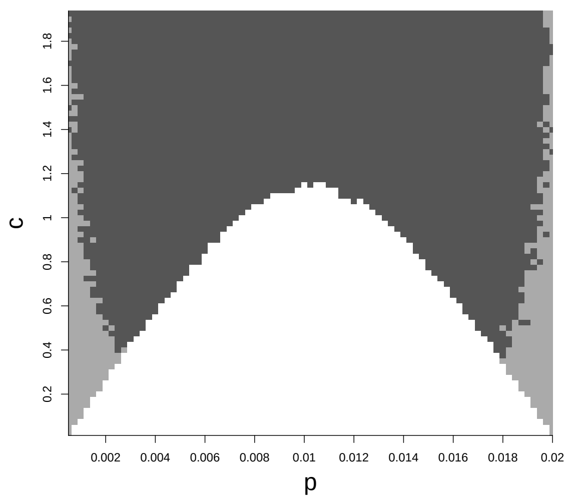
Figure 4.10: Optimal specialization levels s of helper H_1 for different relative request rates p for tasks of type 1 and communication costs c

slices to every task type. In the $\mathcal{S}$-system helpers do not perform any reconfiguration, therefore no reconfiguration costs occur.

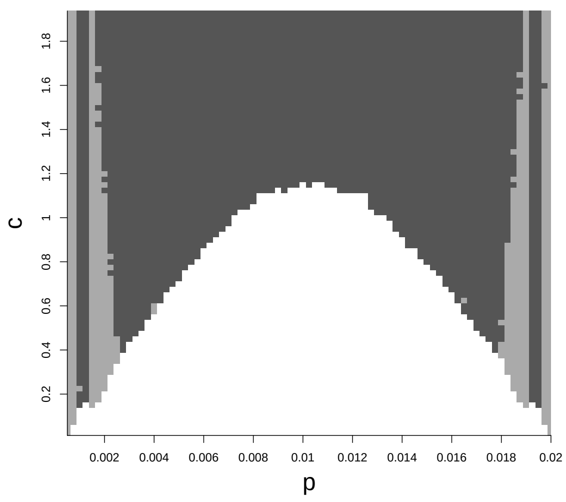
From Theoretical to Empirical Analysis

In order to show the relevance of the theoretical analysis in Section 4.2.4 we investigate a self-organized system with $n = 10$ helpers which are divided in two classes (5 helpers each). Similar as in the theoretical analysis the helpers are either fully specialized or partially specialized with a specialization level as given in Section 4.2.4. This leads to four different systems, depending on the configurations that are used in the two classes. For the analysis of the sojourn times systems using relative request rates $p \in \{0, 0.00025, 0.0005, \dots 0.02\}$ and communication costs $c \in \{0.1, 0.125, 0.15, \dots 2.0\}$ have been simulated. For the analysis of the throughput relative request rates $p \in \{0, 0.0025, 0.005, \dots 0.2\}$ and communication costs $c \in \{0.1, 0.35, 0.6, \dots 20.0\}$ have been tested. The results are averaged over 10 runs.

Figure 4.11 shows which of the four systems has the lowest sojourn times using helpers with (a) 10 slices and (b) 10^6 slices depending on the relative request rate p and the

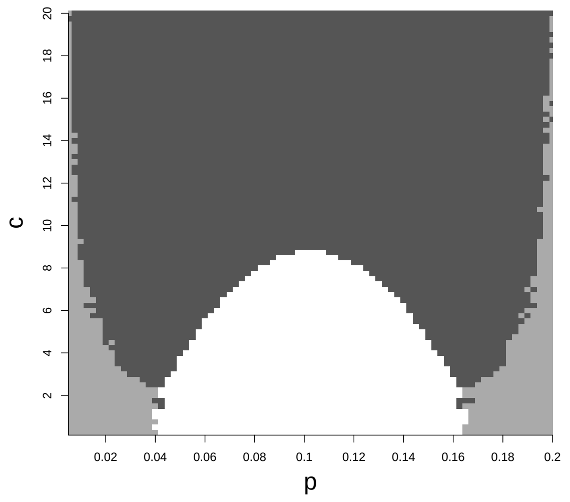


(a) 10^6 slices

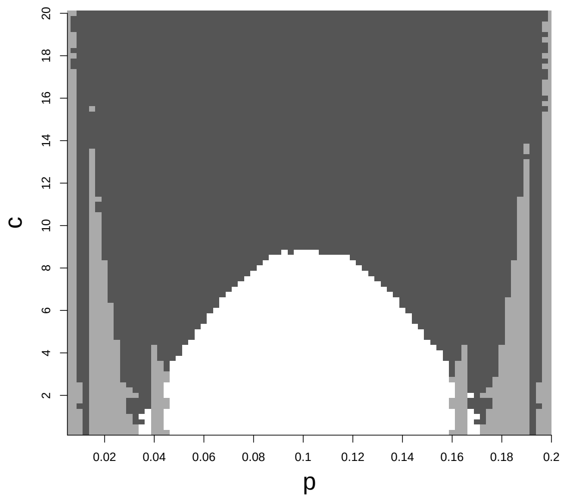


(b) 10 slices

Figure 4.11: Areas with smallest sojourn times for the different system configurations depending on the relative request rate p and the communication costs c

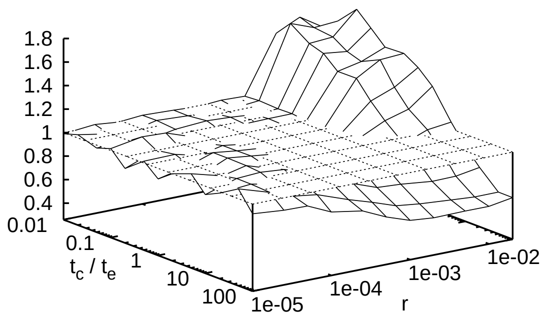


(a) 10^6 slices

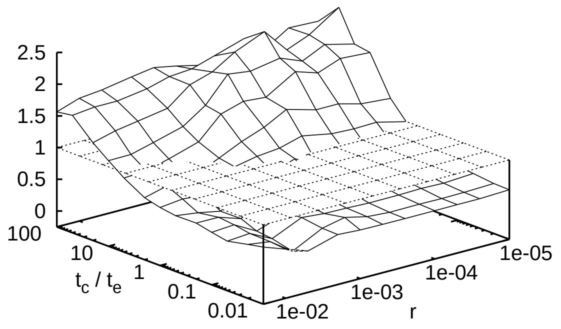


(b) 10 slices

Figure 4.12: Areas with largest throughput for service request rate 0.02 depending on the relative request rate p and the communication costs c



(a) relative throughput



(b) relative sojourn time

Figure 4.13: Self-organized system compared to the static $\mathcal{S}$-system

communication costs c. A service request rate of $r = 0.02$ was used to analyze the sojourn times, because for higher request rates the sojourn times are mainly determined by the communication costs. White colored points represent the system with fully specialized helpers in both classes, the darkest color represents the system where all helpers are partially specialized to be able to execute both types of tasks and in the grey areas stand for the systems which are a mixture of a fully and a partially specialized class. Similar as in the theoretical analysis, four different areas can be observed (compare Figure 4.9(b)). For every area one of the four configuration strategies leads to the smallest sojourn times. Comparing both figures it can be clearly seen, that there is a discretization effect in case the helpers have only 10 slices. This small number of slices leads to more than 4 different areas of optimal behavior. The reason for this is, that the overall costs represented as hyperplanes in the landscape of communication costs and arrival rates are more step-like when less slices are used. The intersections of these hyperplanes for finding the best configuration strategy leads to a structure that can be see in the Figure 4.11(b).

In Figure 4.12 it is shown which of the four systems has the largest throughput depending on the relative request rate p and the communication costs c. In this case the (high) service request rate $r = 0.2$ has been used. In contrast to the empirical analysis of sojourn times, too small arrival rates lead similar throughput rates for all four systems, since the systems are able to fulfill all requests. Again, like in the theoretical analysis, four different areas occur for which one of the four configuration strategies leads to the largest throughput. Also here, a discretization effect appears if the helpers have a small number of slices.

Static Request Rates

We compare the self-organized system and the $\mathcal{S}$-system in situations with fixed service request rates and fixed relative request rates. First the influence of the communication costs t_c on the throughput of the systems is investigated. Figure 4.13(a) shows the throughput of the self-organizing system divided by the throughput of the $\mathcal{S}$-system for service request rates $r \in \{0.01, 0.005, 0.002, \ldots, 0.00001\}$ and communication costs $t_c/t_e \in \{0.01, 0.02, \ldots, 20\}$. It can be seen that for high service probabilities and small values of t_c/t_e (< 1), the throughput of the self-organized system is almost twice as high as for the $\mathcal{S}$-system. Note, that high service request rates are the case when an Organic Support System is particularly important. The throughput of the $\mathcal{S}$-system is better only for small rates r and high relative communication costs ($t_c/t_e \gtrsim 2$). The reason is that in the self-organized system the requests are rejected with some probabilities and this implies additional communication costs. For small rates r and communication costs of $t_c/t_e \lesssim 10$ the performance of both systems is similar.

Figure 4.13(b) compares the sojourn times of the self-organized system and the $\mathcal{S}$-system (note that Figure 4.13(a) and 4.13(b) show the same parameter space but the axes are inverted). The performance of both systems differs significantly for most parameter combinations. For small values of t_c/t_e (< 1) the sojourn times of the self-organized computing system are smaller for all tested probabilities of service. Note that for a small service probability (e.g., $r = 10^{-5}$) both systems have the same sojourn times for $t_c/t_e = 1$. The reason is, that in the self-organizing system a worker needs approximately 2 requests to find a fully specialized helper and in the $\mathcal{S}$-system all requests are accepted but need twice the time to be executed. For approximately $t_c/t_e = 1$ this behavior leads to the same sojourn times ($\approx 2{\cdot}t_c{+}t_e$ in the self-organized system and $\approx t_c{+}2{\cdot}t_e$ for the $\mathcal{S}$-system). For higher communication costs ($t_c/t_e \gtrsim 1$) the sojourn times of the $\mathcal{S}$-system become better because the additional communication operations needed in the self-organizing system become to expensive in comparison to the speedup in execution.

In the following an oscillation effect that is typical for many self-organized systems is demonstrated. First we consider a very simple system with only one helper that has only one slice in a situation with a service request rate of 0.01. Figure 4.14(a) shows the number of actual requests for tasks of type 1 and type 2 over time. It can be seen, that these values oscillate. The reason is that a helper configures for one type of tasks and rejects requests for the other type. When the number of requests for the rejected type increases, the corresponding stimulus value increases as well. This leads to an increased probability that the helper reconfigures its slice and executes the tasks. Compared to a simple $\mathcal{S}$-system with one helper that has two slices the figure shows that the actual number of requests that are waiting is smaller in the self-organized system.

The oscillating behavior occurs also in more complex systems of more than one helpers. This can be seen in Figure 4.14(b) for a system with $n = 10$ helpers where the reconfiguration time is 30 times larger than t_e. In this case of high reconfiguration costs the system reacts slowly.

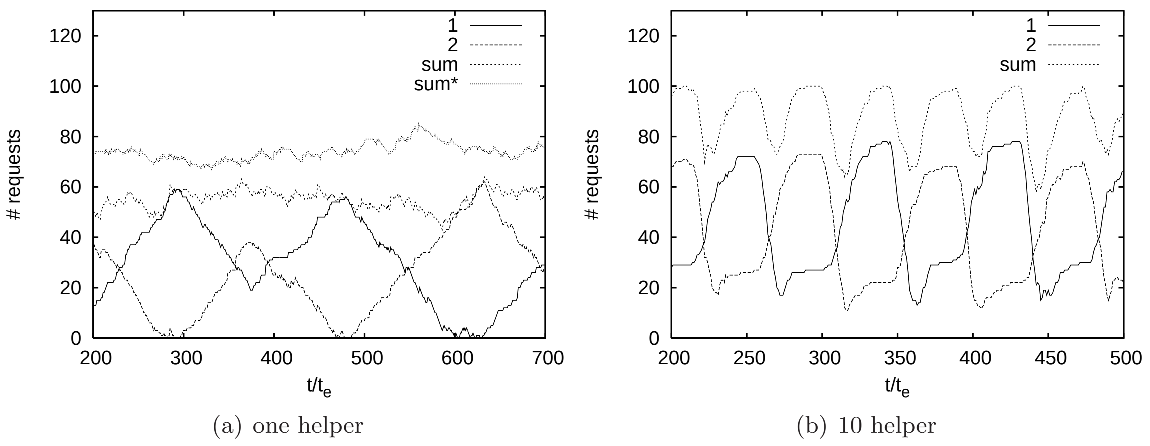


Figure 4.14: Number of actual requests in the self-organized system compared to the respective $\mathcal{S}$-system; 1,2: actual number of request of the type, sum (sum*): total number of actual requests for the self-organized system (respectively for the $\mathcal{S}$-system)

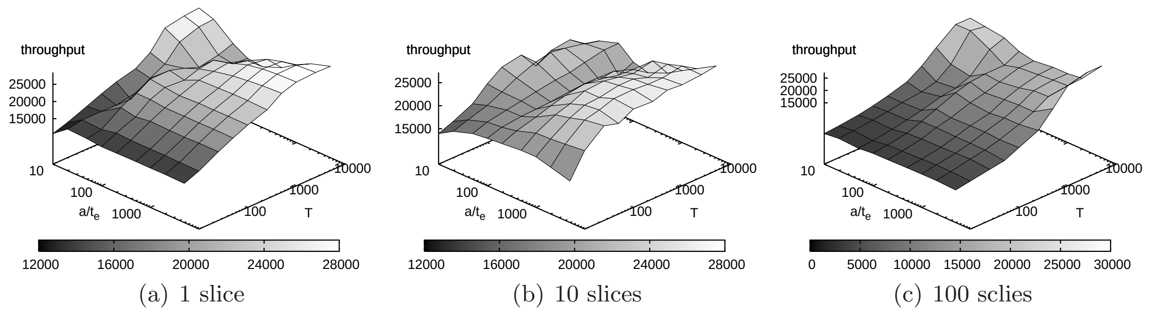
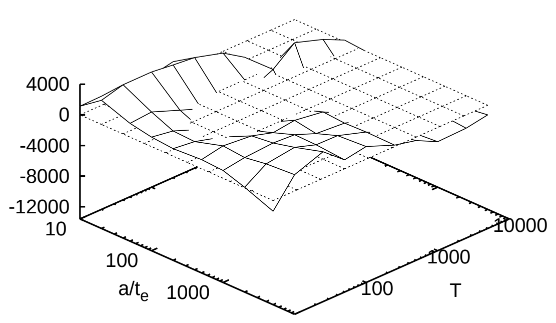


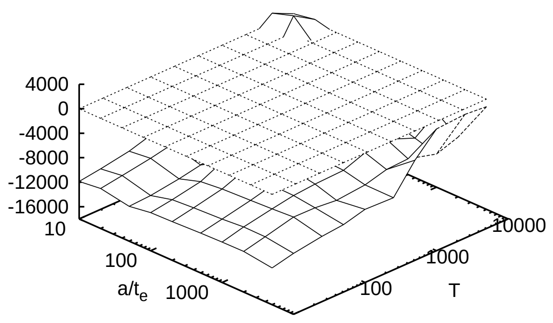
Figure 4.15: Self-organized computing system: throughput for different thresholds T and different degrees of dynamics a

Environment with Changing Service Probabilities

To investigate the behavior of the self-organized system in dynamic situations the case of changing relative request rates was investigated. In the experiments the service request rate was set to $r = 0.2$ and the relative request rate was exchanged between $p = 0.2$ and $p = 0.8$ every $a/t_e = \{10, 20, 50, 100, 200, 500, 1000, 2000, 5000\}$ time steps. Obviously the threshold parameter $T = T_{ij}$ has a strong influence on the adaptiveness of the support system. The higher T the more unlikely that the helpers reconfigure (comp. Equation 4.3). Threshold values $T \in \{10, 20, 50, 100, 200, 500, 1000, 2000, 10\,000\}$ were investigated. For each combination of a/t_e and T the throughput within $3000 \cdot t_e$ time steps was measured. Note, that $a/t_e = 2000$ was a situation where the request rate was changed only once in the simulated time interval.



(a) 10 slices



(b) 100 slices

Figure 4.16: Self-organized service system: throughput difference between the standard system (using the 1-slice reconfiguration strategy) and a system using the 1+half-slice reconfiguration strategy for different thresholds T and different degrees of dynamics a

The experimental results regarding the self-organized system using a reconfiguration time $t_r/t_e = 10$, communication cost $t_c/t_e = 0.1$ and different number of slices $q \in \{1, 10, 100\}$ are depicted in Figure 4.15. The throughput of the self-organized system has to be compared to the average throughput of ≈ 14100 that is achieved by the $\mathcal{S}$-system.

For very large threshold values T the self-organized service system is not very adaptive. In all tested cases the best performance is achieved for the extreme values of $a/t_e = 10$ or $a/t_e = 2000$. The reason is that using $a/t_e = 2000$ does not require a strong adaptiveness, since the relative request rate only changes once. On the other side, $a/t_e = 10$ switches the relative request rate that fast, that in the end the situation is the same as for a fixed relative request rate of $p = 0.5$. Such a situation also does not require adaptiveness and hence a high value of T leads to a good performance. The worst throughput is achieved in situations with many slices and small value for T. In such situations it is likely that a helper excepts a request, even when it has a bad configuration for the corresponding type of tasks. This can lead to large execution times of the tasks (recall that using only one out of k slices for a task increases the execution time by factor k compared to an execution with full specialization).

Figure 4.16 compares the results of the standard self-organized system (using the 1-slice reconfiguration strategy) with one where the 1+half-slice reconfiguration strategy is used. Since both strategies are the same for 1 slice results are shown only for systems with 10 and 100 slices. The motivation to introduce the 1+half-slice reconfiguration strategy was to make the system more easily adaptive for changes of the relative request rates for different types of tasks. It can be seen in the figure that the 1+half-slice reconfiguration strategy obtains for 10 slices a higher throughput for higher threshold values (and not too small values of a, recall that $a/t_e \leq 30$ leads to a situation that is similar the a situation with constant service probabilities). For 100 slices (where a higher adaptivity is even more important) the 1+half-slice reconfiguration strategy is better than the 1-slice strategy. Only for situation with very high threshold values $T > 5000$ and very small values of $a/t_e \leq 30$ the standard reconfiguration strategy is better.

4.3 Collective Decision Making in Organic Support Systems

As pointed out in the last Section in case of very high communication costs the helper do best, if they accept all service tasks directly and do not reject any request, because rejecting is very expensive in this case. When its obvious a priori, that the communication cost exceed the task execution times, the support system does not need have the ability to reject service requests. In such a case, for given arrival rates of the different task types, there is one optimal configuration that should be used for all helpers. This configuration must be chosen in a way, that the helpers are able to execute the incoming task mix most efficiently. Assuming dynamic changes in the arrival rates of the tasks and non-trivial relationships between the number of slices configured for a task type and the task execution time, the search for and the decision about the best configuration for all helpers becomes a hard task.

Brutschy et al. (2008); Brutschy (2007) propose ant-inspired strategies for solving this problem. This work applies principles that are found to be used by the ant *Temnothorax albipennis* when looking for a new nest site. Particularly, a part of the helper components in the support system is assigned to be so called *scouts*. These scouts configure themselves to new configurations and evaluate them by executing the actual task mix. If a scout considers a configuration to be superior than the actual used helper configuration, it starts to recruit other scouts. These scouts also assess the new configuration and may also start to recruit other scouts. As soon as a certain number of scouts prefer a particular candidate configuration, the scouts switch their behavior and start to recruit the remaining helper components to the new configuration.

These ant inspired strategies for making a collective decision about what configuration to employ are analyzed experimentally and are compared to a non-adaptive reference strategy. On all tested environments, the ant-inspired adaptive strategy proves to be versatile and very robust. Using the ant-inspired strategies, the scout components are able to find good configurations even in complex configuration spaces.

4.4 Task Partitioning in Organic Support Systems

In the first investigation we assumed that a task can be executed on a fraction of the resources of a helper. In this section we will investigate the case that the resources of a helper are not sufficient to finish a whole task using one configuration. In such a situation the tasks have to be partitioned into subtasks that can be executed successively. When a helper has finished a subtask there are two possibilities. First, the helper can reconfigure in order to meet the resource demands of the next subtask or second, the task, i.e., the result of the partly execution, can be transferred to another helper that is already appropriately configured. For the execution of a whole task all its subtasks have to be executed in the right order and on appropriately configured helpers.

To keep things simple, we assume that only one type of support tasks is needed by the worker system and an unlimited number of these tasks wait in the worker system to be executed. In order to maintain a good performance of the support system, i.e., a high throughput of tasks, it is necessary to minimize the time helpers are not working. Beside executing subtasks, helpers can wait for other helpers or can perform a reconfiguration. A helper has to wait for other helpers if it has finished its subtask but no appropriate configured helper for the next subtask is available or in case the helper has to wait for a new task after it has finished and transferred its previous task successfully.

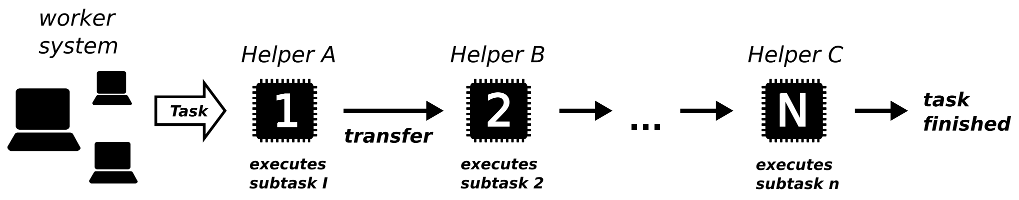


Figure 4.17: Schematic view of the execution of a task in the task partitioning system

To minimize waiting times (also called queueing delays) it is crucial to equalize the work capacities of the groups of helpers specialized to the different subtasks. This means that in a given time interval the Organic Support System executes the same number of every type of subtask. For example, if a subtask takes much more time to be executed than the other subtasks, the throughput of the system is best, if most helpers are working for that type of subtask. In the case of equal subtask execution times the system has the lowest waiting times, if there is the same number of helpers working for the different subtasks.

To ensure that the support system executes roughly the same number of the different subtasks within a certain time interval, the system needs to allocate the appropriate number of helper for the different types of subtasks. Since there is no global control, the helpers have to decide on their own for which subtask to configure, i.e., the system of helpers has to self-organize their specialization. These decisions can be made using different strategies (called reconfiguration strategies). A simple strategy would be that a helper never transfers a task to an other helper and instead, by repeatedly reconfigurating, it does all the work on its own. Clearly this is not the best strategy for the realistic case that the reconfiguration time is much higher than the subtasks execution time. More complex reconfiguration strategies incorporate locally observed information like the actual queueing delay of the helper.

Not only the question how well the desired task partitioning is reached has to be considered when analysing different reconfiguration strategies. It is also interesting how fast a strategy can adapt and how stable it is in case of perturbations. Perturbations can occur if helpers leave or new helpers enter the support system (because of malfunctions or a spatial separation) or, for example, in the case of changing execution times of the subtasks. The reconfiguration strategies also have to be investigated with regard to their scalability, e.g., their application in support systems of different size.

4.4.1 Model of the System

The support tasks are divided into I subtasks which have to be executed successively, i.e., subtask i has to be executed before subtask $i + 1$. A helper is always configured in a way that it can perform exactly one type of subtasks. A helper is called to be of type i if it is configured for subtask i. It is assumed that the transfer of a task takes a certain but negligible amount of time.

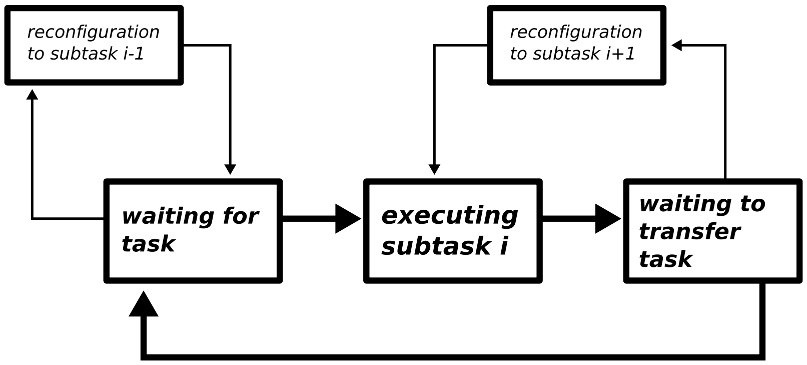

reconfiguration to subtask i-1
reconfiguration to subtask i+1
waiting for task
executing subtask i
waiting to transfer task

Figure 4.18: Possible states of a helper configured for subtask type i

Figure 4.18 shows the possible states of a helper. If a helper of type i has executed a subtask it goes into waiting state for transferring the task to a helper of type $i + 1$. If there is a suitable configured helper (already waiting or showing up after a while) the task is transferred to this other helper. After transferring a task to another helper the helper itself goes in waiting state for a transfer of a task to execute subtask i again. Exceptions are helpers for subtask 1 since it is assumed that they can always start a new task and helpers for the last subtask I, because these do not need to transfer the task to other helpers. Thus, all other helpers can be in two different waiting states, either a helper is waiting to get a new task or it is waiting for an appropriate helper transfer the task.

A helper can decide to reconfigure to another subtask. If a helper has finished the execution of subtask i and is in waiting state for transfer there is also the possibility that the helper reconfigures to type $i + 1$. After that reconfiguration the next subtask of the actual task can be executed immediately. On the other side if a helper of type i was waiting for a new task it can also reconfigure to subtask $i - 1$ and go into waiting state for this type.

The decision to reconfigure to a certain subtask depends on the used reconfiguration strategy. This strategy is based on local information only, i.e., without any explicit knowledge of the number of helpers configured for the different subtasks. The local information taken into account by the helper is the experienced queueing delay. Long waiting times for getting new tasks, observed by helpers configured for subtasks type $i + 1$, are caused by too few helpers for subtask i. It is reasonable for some helpers of type $i + 1$ to reconfigure to subtask i. This reconfiguration will decrease the number of helpers for subtask type $i + 1$ immediately and will raise the number of helpers for the subtask i after a certain time needed for reconfiguration. This delay in the effect of the reconfiguration operations can lead to an overreaction of the whole system. On the other hand if the strategy of the helpers is too rigid, the system stays in a suboptimal state for too long after a perturbance. The reconfiguration strategy must balance between these effects.

In the following an example of the mentioned overreaction of the system is given. Assume a support system of eight helpers which have to execute tasks divided into two subtasks of equal run time. The reconfiguration of a helper needs two time steps. The reconfiguration strategy of the helper is to reconfigure with a probability of 0.5 in case they have to wait, i.e., when no transfer partner was found. In Figure 4.19 an example of the effect of this strategy is given. At the first shown time step six helpers are configured for subtask one and only two helpers can work on subtask two. Such an imbalanced situation may occur, for instance, if accidentally helpers disappear. Helper H_2 transfers its task to H_3 and H_7 to H_5. Since there is no helper configured for subtask two left, helper H_1, H_4, H_6 and H_8 have to wait. Using the reconfiguration strategy these four helper decide with probability 0.5 to change their configuration to subtask two. This leads to H_4 and H_8 deciding to reconfigure. In the next shown time step both helpers are in reconfiguration mode, but still too few helpers are available to execute subtask two. Again two helpers (H_1 and H_2) have to wait. This time H_2 decides to switch its type to subtask two. In the following time step H_4 and H_8 have finished their reconfiguration and there are four helper of type two but only three helper of type one. One helper of type two (H_3) which has to wait switches to task one now. Finally, in the last shown time step an equal proportion of helpers configured for the two task types is present in the system. The system started in a situation where

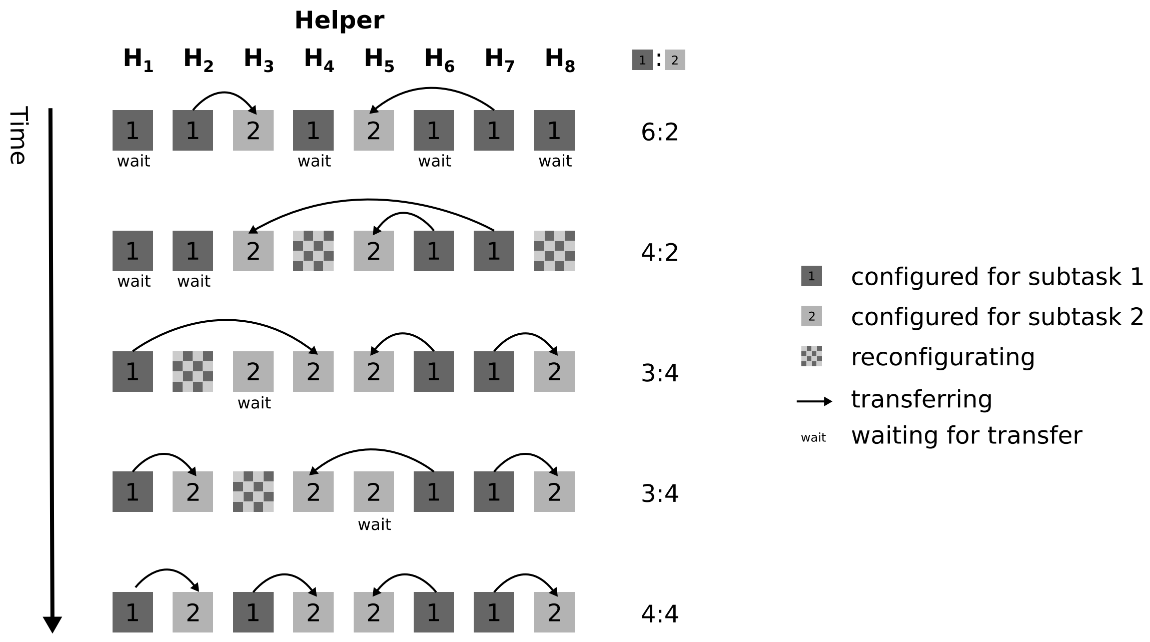


Figure 4.19: Example of a system run of the support system; depcited are eight helper starting in a configuration of six helper of type 1 and two helper of type 2

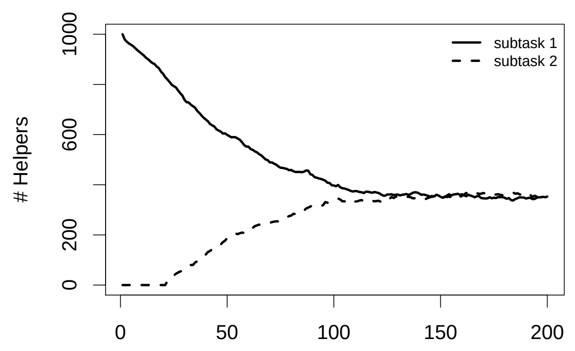
there were too few helpers configured for subtask two. Due to local decisions of the helpers and the assumed reconfiguration delay, the systems reaction was too strong and even led to an intermediate situation where there were to few helper configured for subtask one.

Clearly performance and adaptivity of the systems depends on the used reconfiguration strategy of the helpers. In the following different reconfiguration strategies are investigated. We start with a highly unbalanced distribution of helpers for the different subtasks, more precisely we set the configuration of all helpers to subtasks one, and observe how fast and how well the systems can recover. To avoid artifacts, the initial start time of a helper, i.e., the time when it starts executing the first subtask, is randomly distributed in $[0, t_e]$, where t_e is the mean execution time for a subtask on this helper.

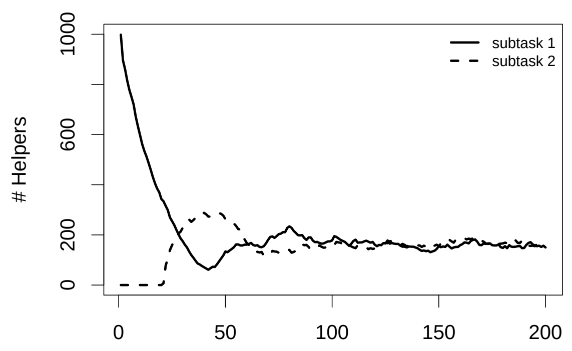
If not stated otherwise, the following parameters were used for the simulations. The number of subtasks is $I = 2$, the number of helpers is $h = 1000$, the reconfiguration time is $\tau = 20$. The execution time for the subtasks t_e is normal distributed with mean 1 and variance 0.01, i.e., $t_e \sim N(1, 0.01)$. The time for transferring a task is set to zero.

4.4.2 Reconfiguration Strategy `simple`

In the first investigated reconfiguration strategy called `simple` a certain percentage of the helpers always reconfigures, i.e., to switch its specialization from one type of subtasks to another type. A fixed parameter $p \in (0, 1]$ determines the switching rate, i.e., the



(a) $p = 0.01$



(b) $p = 0.05$

Figure 4.20: Number of active helpers for both subtasks over time when using strategy `simple`; 1000 helpers were used

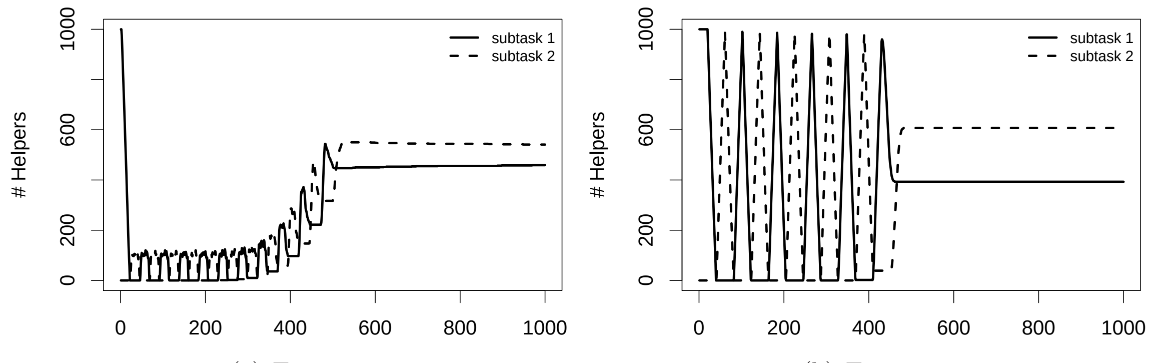
probability for a helper to switch from type i to type $(i + 1)$ *mod* I before starting with the next subtask. If a helper decides to switch when executing a subtask it finishes this subtask and switches afterwards. The idea behind this strategy is to equalize the number of helpers for every subtask. Observe, that strategy `simple` implies that for a large number of helpers always some helpers exist that are in a phase of reconfiguration.

In Figure 4.20 the behavior of the support system using the `simple` strategy is shown for 1000 helpers and switching rates $p = 0.01$ (left) and $p = 0.05$ (right). The figure shows the number of active helpers (i.e., helpers that are not in reconfiguration mode) for both subtasks. When using a small value for p (0.01) the numbers of active helpers for both subtasks slowly converge to around 400. When using the larger switching rate (0.05) the number of helpers for both subtasks converge to 200. This is because larger p values lead to more helpers switching their type and while in reconfiguration mode they can not execute subtasks. The oscillation of the number of helpers around time step 200 with a decreasing amplitude for $p = 0.05$ is an example of the mentioned overreaction.

Since always a fraction of the helpers are in reconfiguration mode, the performance of systems using the `simple` strategy is not optimal. In the next section we introduce a strategy which lead to a decreasing fraction of reconfigurating helpers over time.

4.4.3 Reconfiguration Strategy `wait`

In the second reconfiguration strategy, called `wait`, each helper has a so called *waiting threshold* that determines the maximal time interval that the helper will stay in waiting state. For helper j configured for subtask type $i > 1$ the waiting threshold T_j is the maximal time the helper waits for being contacted by a helper of type $i - 1$ to get a new task. The threshold T_j is also the maximal time helper $i < I$ waits for a helper of type $i+1$ to transfer the task to. If a helper has been waiting for time T_j it starts to reconfigure



(a) $T_j = 2$ (b) $T_j = 20$

Figure 4.21: Number of active helpers for both subtasks over time when using strategy `wait` with identical waiting thresholds T_j for all helpers; 1000 helpers were used

to change its type. If a helper of type $i < I$ has reconfigured to subtask $i + 1$ it starts immediately with the execution of subtask $i + 1$ of the task that it has. If a helper of type 2 has reconfigured itself to subtask 1 it starts immediately with the execution a subtask of type 1 (Recall, that it is assumed that always a new task exists). If a helper of type $i > 2$ has reconfigured itself to subtask $i - 1$ it goes into waiting state again.

Waiting Threshold Distribution for the `wait` Strategy

First we investigate the `wait` strategy using the same fixed waiting thresholds for all helpers. In Figure 4.21 the behavior of the system is shown for waiting thresholds $T_j = 2$ and $T_j = 20$. Two interesting observations can be made:

First, the number of helpers is strongly oscillating for time $t < 600$ and $T_j = 2$ (respectively $t < 400$ and $T_j = 20$). For example, for $T_j = 20$ the number of helpers of type 1 oscillates 5 times between ≈ 0 and > 990 before a more stable situation is reached. The reason for this is, that often a large number of helpers decide to respecialize within a very small time interval. This is obviously an unwanted behavior.

Second, in the stable state the difference in the number of helpers of both types can be large. For larger values of T_j this effect is stronger. For $T_j = 2$ the difference between the number of helpers of type 2 and type 1 in the stable state is 541-459=82. For $T_j = 20$ this difference is 607-393=214. The reason for this is, that when all waiting delays become smaller than the thresholds T_j no helper will switche anymore. Consequently, the optimal proportion of helpers can not be reached.

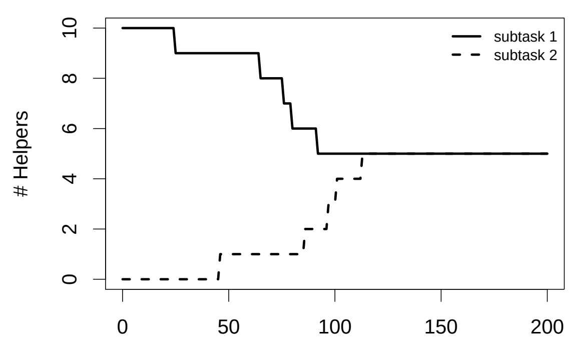
With identical thresholds for all helpers the `wait` strategy leads to oscillations because to many helpers start to reconfigure at the same time. To overcome this problem we new use different waiting thresholds for the helper. Using an exponential distribution for the waiting thresholds $T_j \sim \text{Exponential}(p)$ (i.e., $f(x) = p \cdot e^{-px}$ is the corresponding density function) leads to a constant rate of helpers that switch their subtask, when the waiting time for one type of subtasks increases. The parameter p is fixed and does not change over time. If a helper gets into a waiting state it generates a new random (exponentially distributed) threshold T_j. All following investigations for the `wait` strategy are made with exponentially distributed waiting thresholds.

Number of Helpers

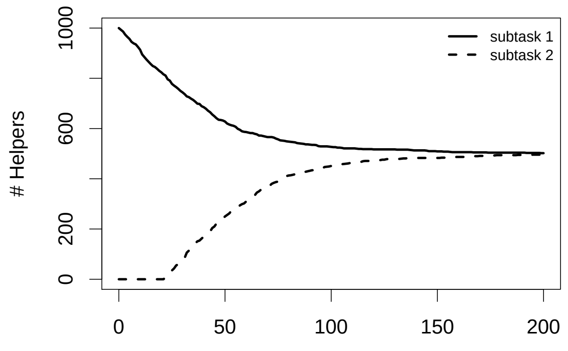
The influence of the number of helpers in the support system is depicted in Figure 4.22. Three different values for the switching rate parameter $p \in \{0.01, 0.05, 0.2\}$ were investigated for systems of $h \in \{10, 1000\}$ helpers. The results of systems with 10 helper do not differ very much from the results obtained with 1000 helper, beside the fact that the curves are more steplike. Using the (small) value $p = 0.01$ the number of the helper for the two subtasks slowly approach each other (Figure 4.22(a) and 4.22(b)). A higher value $p = 0.05$ leads to a slight overreaction of the system (Figure 4.22(c) and 4.22(d)). The desired equilibrium is already reached at around time step 150, which is faster than in the systems with $p = 0.01$. Figure 4.22(f) shows strong oscillations in the number of helpers for $p = 0.2$ and 1000 helpers. For $h = 10$ helpers these oscillations occur very roughly too, as can be seen in Figure 4.22(e). Although showing different dynamics finally all parameter settings lead to a convergence in the desired state where the same number of helper is specialized for the two subtasks.

Different Subtask Execution Times

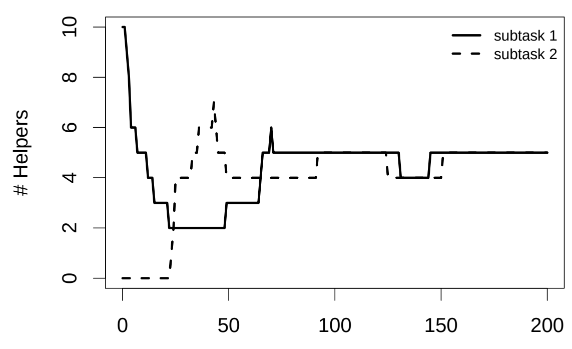
So far in the investigated systems the assumed execution times for both subtasks were equal. In this case the systems equalize the number of helpers for both subtasks. Note, that this adaptation leads to a maximal performance, as no helper will switch its type, and therefore no reconfiguration overhead will occur. In the following we investigate situations where the execution times for both types of subtasks differ. To reach an optimal performance, the system needs to converge to a state where the relation between the number of helpers of both types is the same as the relation of their mean execution times. In the experiments the execution time for subtasks of type 1 is set to $t_e \sim N(1, 0.01)$. The execution time for subtasks of type 2 are set to $t_e \sim N(0.5, 0.005)$ (respectively $t_e \sim N(0.1, 0.001)$). The switching rate parameter for the exponential distribution of the waiting times T_j was set to $p \in \{0.01, 0.05, 0.2\}$. Results are depicted in Figure 4.23. The adaptation process shows the desired behavior. For example, when the execution time of subtask type 1 is two times higher than the execution time for subtasks of type 2, the



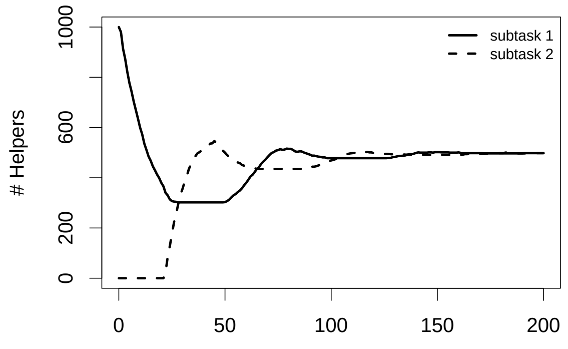
(a) 10 helper, $p = 0.01$



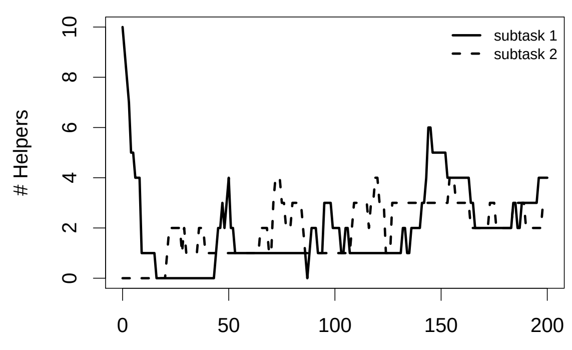
(b) 100 helper, $p = 0.01$



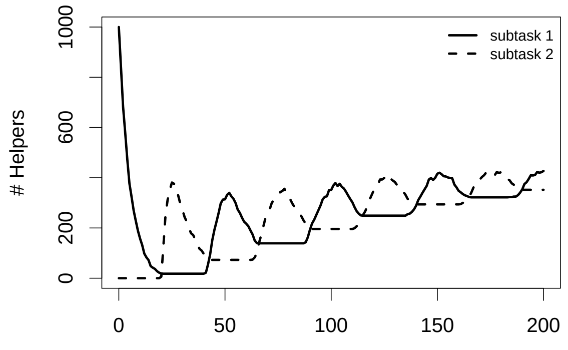
(c) 10 helper, $p = 0.05$



(d) 100 helper, $p = 0.05$

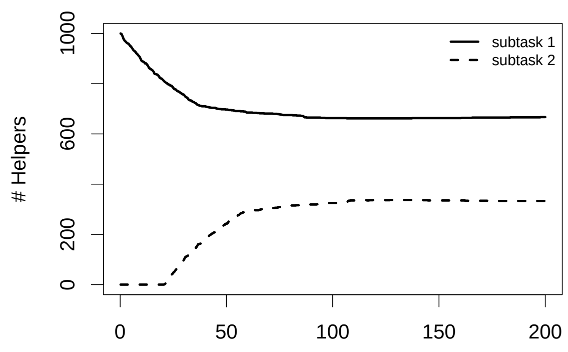


(e) 10 helper, $p = 0.2$

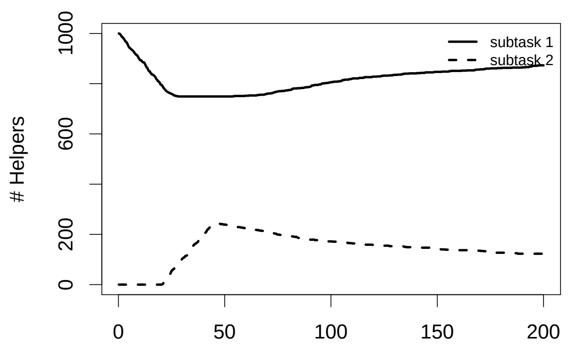


(f) 100 helper, $p = 0.2$

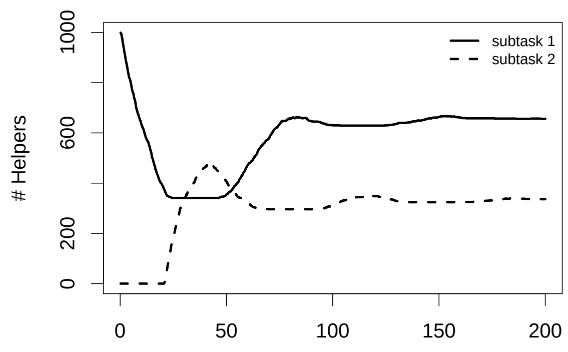
Figure 4.22: Number of active helpers for both subtasks over time when using strategy `wait` using exponentially distributed waiting thresholds



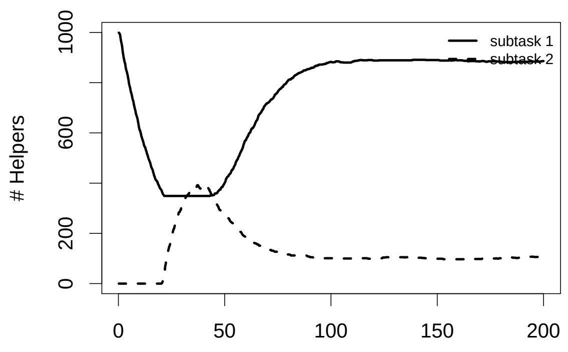
(a) t_e of subtask 1 two times higher, $p = 0.01$



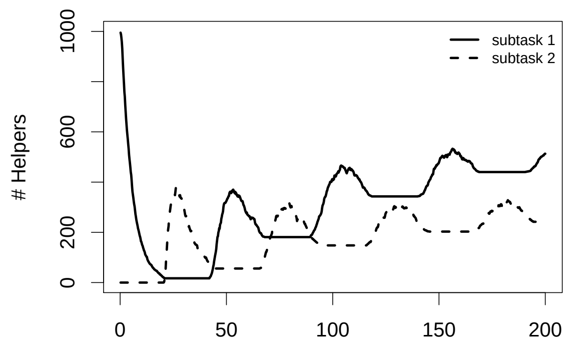
(b) t_e of subtask 1 ten times higher, $p = 0.01$



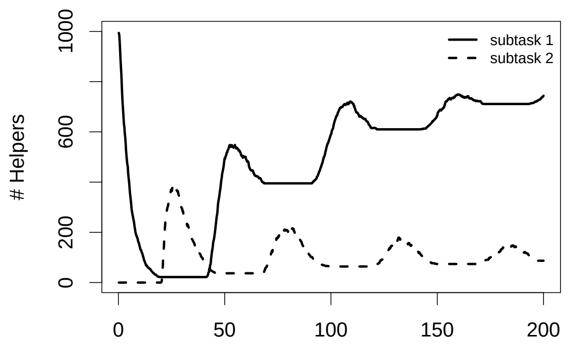
(c) t_e of subtask 1 two times higher, $p = 0.05$



(d) t_e of subtask 1 ten times higher, $p = 0.05$



(e) t_e of subtask 1 two times higher, $p = 0.2$



(f) t_e of subtask 1 ten times higher, $p = 0.2$

Figure 4.23: Number of active helpers for both subtasks over time when using strategy `wait` for different subtask execution times; mean execution time for subtask type 1 two times higher (left) or ten times higher (right) than for subtask type 2

number of helpers for subtask 1 converges to 667 and the number of helpers of type 2 to 333. For the different settings of p again different dynamics show up.

Dynamically Changing Execution Times

To investigate the adaptiveness of systems using the `wait` strategy we studied the influence of dynamically changing execution times. We used $t_e \sim N(1, 0.01)$ for one type of subtasks, and $t_e \sim N(0.25, 0.0025)$ for the other type of subtasks. The execution times were alternated at times $t = 200$, $t = 400$, and $t = 600$. Again, we used $p \in \{0.01, 0.05, 0.2\}$ for the switching rate parameter. Results are given in Figure 4.24. For small $p = 0.01$ the adaptation process is too slow to reach a converged state before the next change of subtasks execution times. For large $p = 0.2$ again oscillations can be observed, and for medium $p = 0.05$ a smooth and fast adaptation is possible. In all experiments the parameter p had a strong influence on the dynamics of the system. Later in this chapter we will investigate the influence of p on the performance, i.e., the number of executed tasks.

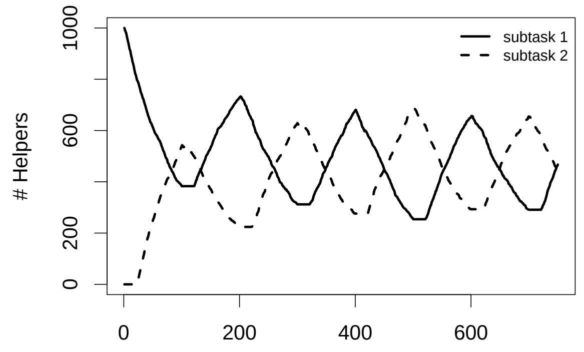
Number of Subtasks

Results concerning support systems which divide a task into more than two subtasks are depicted in Figure 4.25. Again reconfiguration strategy `wait` was used for service tasks with 5 and 10 subtasks. Note, that a helper that is configured for subtask i may reconfigure only to subtask of type $i - 1$ or $i + 1$, if possible. To which type of subtasks the helper reconfigures depends on whether it could not find a helper of type $i + 1$ to transfer its tasks or whether the helper could not find a helper that is ready to execute a subtask of type $i - 1$. In Figure 4.25 it can be seen that the system manages to converge to a stable state where approximately a fraction of #helpers/#subtasks of all helpers are configured for every type of subtask (equal execution times for the subtasks were used).

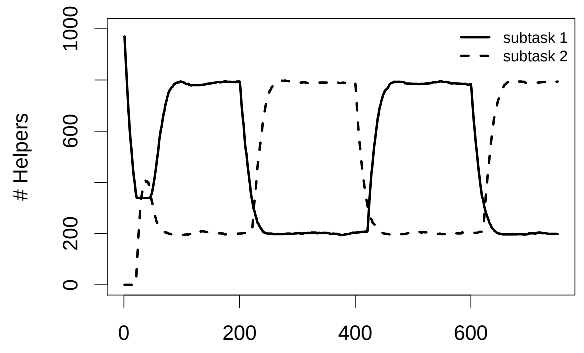
4.4.4 Theoretical Analysis of Reconfiguration Strategies

In this section we model the proposed system using delay differential equations. In the first part of the section the `simple` strategy is analyzed with respect to its fixed-points and its stability. After that we will show numerical solutions for initial value problems of the strategy `simple` and the strategy `wait` with exponential distributed waiting thresholds.

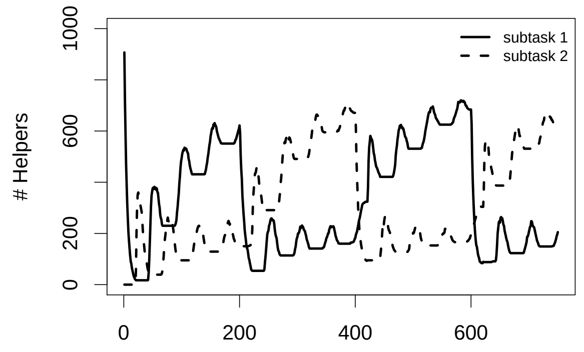
We assume in this section that there are only two subtasks. Let $\gamma_i : T \mapsto [0, 1]$, $i \in \{1, 2\}$ be the fraction of helpers of type i in the system, that are working for subtask i at time $t \in T$. Let τ be the time needed for reconfiguration of a helper. Then a helper that switches at time $t - \tau$ goes into reconfiguration mode and will increase the number of the helpers configured for its new task type at time t. Formally, we have the following delay differential system to describe the system when using strategy `simple`.



(a) $p = 0.01$

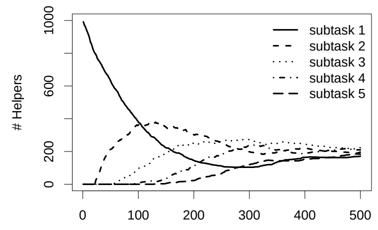


(b) $p = 0.05$

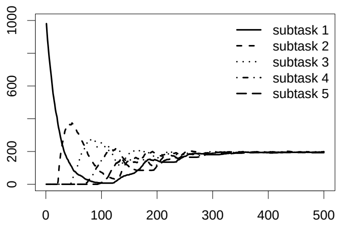


(c) $p = 0.2$

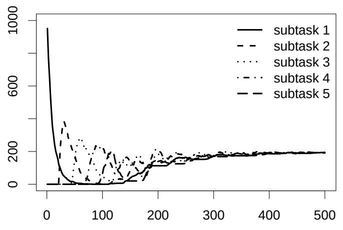
Figure 4.24: Number of active helpers for both subtasks over time when using strategy `wait` under the condition of dynamically changing execution times; execution times for both subtasks normally distributed with $N(1, 0.01)$ and $N(0.25, 0.0025)$; execution times changed at time step $t = 200$, $t = 400$, and $t = 600$



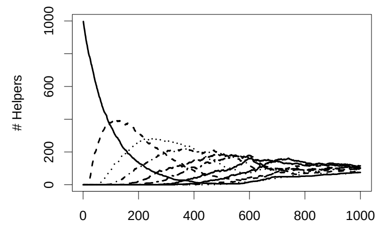
(a) 5 subtasks, $p = 0.01$



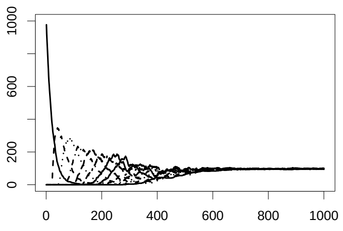
(b) 5 subtasks, $p = 0.05$



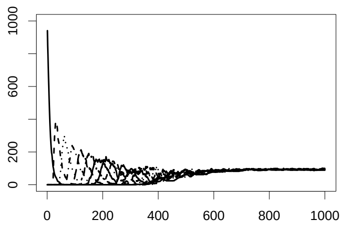
(c) 5 subtasks, $p = 0.1$



(d) 10 subtasks, $p = 0.01$



(e) 10 subtasks, $p = 0.05$



(f) 10 subtasks, $p = 0.1$

Figure 4.25: Number of active helpers over time when using strategy `wait` and 5 (top) or 10 (bottom) subtasks

$$\begin{aligned} \frac{d\gamma_1}{dt} &= -p \cdot \gamma_1(t) + p \cdot \gamma_2(t-\tau) \\ \frac{d\gamma_2}{dt} &= -p \cdot \gamma_2(t) + p \cdot \gamma_1(t-\tau) \end{aligned} \tag{4.4}$$

Standard Approach of Analyzing the Stability

One way of analyzing the stability of a delay differential equation system is similar as for ordinary differential equation systems. Let J_0 be the Jacobian matrix with respect to $(\gamma_1(t), \gamma_2(t))$ evaluated at the equilibrium point, and let J_τ be the Jacobian matrix with respect to $(\gamma_1(t-\tau), \gamma_2(t-\tau))$ evaluated at the equilibrium point. Using the Jacobian matrices J_0 and J_τ of the system, one can calculate the characteristic equation of the equilibrium point. The characteristic functions of such delay differential systems are so called quasi-polynomials which have the form $P(\lambda) + Q(\lambda)e^{-\lambda\tau} = 0$ with P and Q being polynomials in λ. Formally the characteristic equation is

$$\det\left[J_0 - \lambda I + e^{-\lambda\tau} J_\tau\right] = 0 \tag{4.5}$$

where I is the identity matrix. Applied to our system we look for solutions of

$$\det\begin{bmatrix} -p-\lambda & p \cdot e^{-\lambda\tau} \\ p \cdot e^{-\lambda\tau} & -p-\lambda \end{bmatrix} = \lambda^2 + 2\lambda p + p^2 - p^2 \cdot e^{-2\lambda\tau} = 0 \tag{4.6}$$

If any of the solutions of the characteristic equation has positive real parts, then the equilibrium point is unstable. If they all have negative real parts, then the equilibrium point is asymptotically stable. It is easy to see that the quasi-polynomial in Equation (4.6) has a solution $\lambda = 0$, hence the stability is undecidable to linear order.

Improved Approach of Analyzing the Stability

Instead of using the delay differential equation system (4.4), that describes the behavior of the number of helpers at a certain time, we investigate the number of active helpers (i.e., the number of helpers, that are not in reconfiguration mode) and the difference between the number of helpers of type 1 and type 2. Formally, let $\Gamma_1(t) := \gamma_1(t) + \gamma_2(t)$ and $\Gamma_2(t) = \gamma_1(t) - \gamma_2(t)$. This results in two delay differential equations that can be written as follows.

$$\frac{d\Gamma_1}{dt} = -p \cdot \Gamma_1(t) + p \cdot \Gamma_1(t-\tau) \tag{4.7}$$

$$\frac{d\Gamma_2}{dt} = -p \cdot \Gamma_2(t) - p \cdot \Gamma_2(t-\tau) \tag{4.8}$$

In Cooke and Grossman (1982); Freedman and Kuang (1991) delay differential equations of the following type are analyzed:

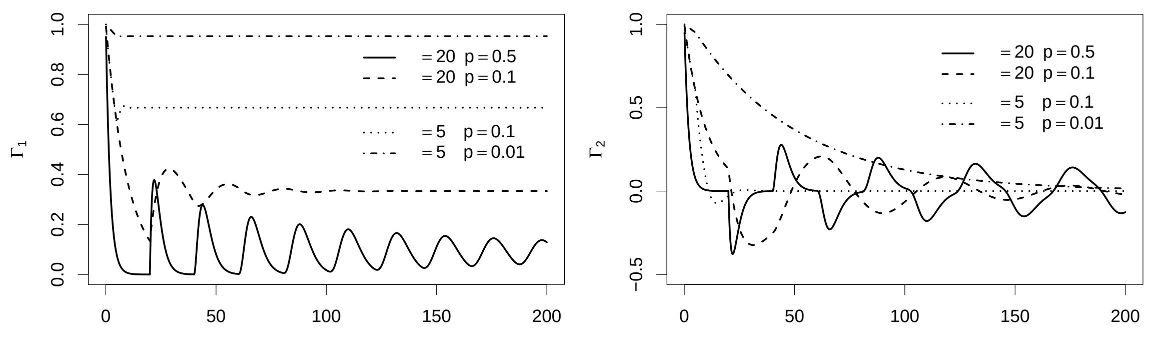
$$\frac{d\Gamma(t)}{dt} + \alpha\frac{d\Gamma(t-\tau)}{dt} + \beta\Gamma(t) + \gamma\Gamma(t-\tau) = 0 \tag{4.9}$$

where $\tau, \alpha, \beta, \gamma$ are real constants. Equation (4.8) can be written in the form of (4.9) with $\alpha := 0$, $\beta := p$, and $\gamma := p$. It was shown in Freedman and Kuang (1991), that there can be no switch in stability in Equation (4.8) when the delay τ is changed. For $\tau = 0$ any point is an equilibrium point for Equation (4.7), and only $\Gamma_2(t) = 0$ is an equilibrium point for Equation (4.8). Hence, a stability analysis of the ordinary differential equation in Equation (4.8) with $\tau = 0$

$$\frac{d\Gamma_2}{dt} = -p \cdot \Gamma_2(t) - p \cdot \Gamma_2(t) = -2 \cdot p \cdot \Gamma_2(t)$$

can be used to characterize the stability of the difference of the number of helpers of the different types. For this equation the roots of the characteristic equation $\lambda + 2 \cdot p = 0$ have all negative real parts (if $p > 0$). Therefore the equation is asymptotically stable. This leads immediately to the fact that Equation (4.8) is asymptotically stable for every reconfiguration time τ. Thus, for every starting condition the difference of the functions γ_1 and γ_2 will converge to 0.

The delay differential equation as given in Equation (4.7) is more complicated, as $\lambda = 0$ is a solution of the characteristic equation. Equation (4.7) can be written in the form of



(a) Equation 4.7 (b) Equation 4.8

Figure 4.26: Dynamics of Equation 4.7 (a) and Equation 4.8 (b) over time; Equation (4.7) is stable but not asymptotically stable, it converges to different values for different values of p or τ; Equation 4.8 is asymptotically stable, it converges to 0 for all starting conditions and all values p and τ

Equation (4.9) with $\alpha := 0$, $\beta := p$, and $\gamma := -p$. It was shown in Freedman and Kuang (1991), that for $|\alpha| < 1$ and $\beta + \gamma = 0$ Equation (4.9) is stable, but not asymptotically stable. Hence, the sum of helpers that are active at time t as described in Equation (4.7) is a stable (but it is not asymptotically stable).

We have shown that the difference of active helpers for both tasks $\Gamma_2(t)$ is asymptotically stable and converges always to zero. Together with the fact that the sum of helpers $\Gamma_1(t)$ that are active at time t is also stable, it follows that the number of helpers for the two task types $\gamma_1(t)$ and $\gamma_2(t)$ have to be stable.

The stability analysis for the difference (respectively sum) of the number of active helpers is illustrated in Figure 4.26 for different values of parameters p and τ. Note, that for large reconfiguration times τ and large values of p the number of active helpers converges to a relatively small value. The smaller τ and p become, the less helpers will be in reconfiguration mode, when the system has converged.

Initial Value Problems

For reconfiguration strategy `simple` the stability could be analyzed analytically. Unfortunately, we were not able to find an explicit solution for the given differential equation system. Therefore, we investigate numerical solutions to initial value problems in the following for both reconfiguration strategies.

Consider the reconfiguration strategy `wait` with exponentially distributed waiting thresholds. Assume t_1 and t_2 are the execution times for the two subtasks. Then within a time unit the helpers specialized for subtask $i \in \{1, 2\}$ can execute γ_i/t_i subtasks. Without loss of generality assume helpers for subtask two can not execute as many tasks as helpers for subtask one, then the number of helpers for subtask one which are in waiting state is $\gamma_1(t) - \frac{t_1}{t_2} \cdot \gamma_2(t)$. Since the waiting thresholds of these helpers are exponentially distributed with parameter p the system can be modeled with the following delay differential equations

$$\begin{aligned} \frac{d\gamma_1}{dt} &= -p \cdot \max\{0, \gamma_1(t) - \frac{t_1}{t_2} \cdot \gamma_2(t)\} + p \cdot \max\{0, \gamma_2(t-\tau) - \frac{t_2}{t_1} \cdot \gamma_1(t-\tau)\} \\ \frac{d\gamma_2}{dt} &= -p \cdot \max\{0, \gamma_2(t) - \frac{t_2}{t_1} \cdot \gamma_1(t)\} + p \cdot \max\{0, \gamma_1(t-\tau) - \frac{t_1}{t_2} \cdot \gamma_2(t-\tau)\} \end{aligned} \tag{4.10}$$

In Figure 4.27 the numerical solutions for the number of helpers are depicted for reconfiguration strategies `simple` (left column) and `wait` (right column). The initial conditions for the delay differential equation system are such, that $\gamma_1(t) = 1$ and $\gamma_2(t) = 0$, $t \in [-\tau, 0]$, i.e., all helpers are specialized for subtasks of type 1. The reconfiguration time was set to $\tau = 20$, and parameter p was set to values 0.01 (weak switching intensity), 0.05 (medium switching intensity), and 0.2 (strong switching intensity). For $p = 0.01$ and both reconfiguration strategies the number of helpers smoothly converges to its final value. For strategy `wait` γ_i converges to 0.5. This is not the case for strategy `simple`. Note, that as shown for strategy `simple` the number of active helpers is stable, but not asymptotically stable. For strategy `simple` the number of helpers in reconfiguration mode strongly depends on the value of p. For $p = 0.05$ the numerical solution converges to 0.25, and for $p = 0.2$ it converges to 0.1.

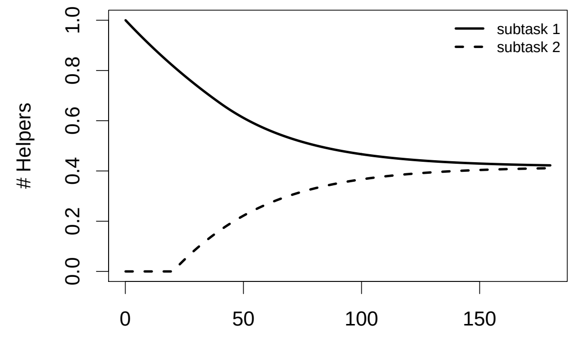
Comparing the numerical solutions of the delay differential equation system (Figure 4.27) with the discrete event simulation for many helpers (number of helpers $h = 1000$, Figure 4.20 and Figure 4.23) , it can be seen, that the differential equations approximate the proposed system very well. If the number of simulated helpers becomes too small, this close correspondence becomes weaker.

System Performance

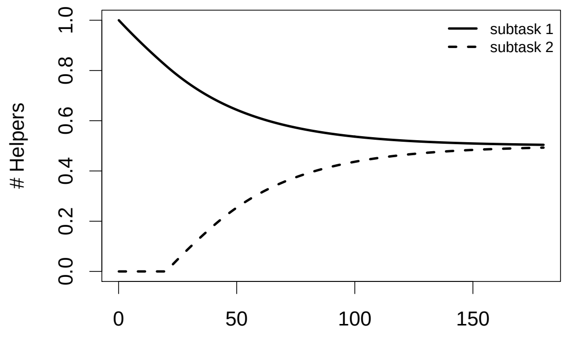
The bottleneck of the system is the type of subtask where the least tasks are executed. Hence, the normalized performance $\phi(T)$ of the system in the time interval $[0, T]$ can be measured as

$$\phi(T) = \frac{1}{T} \int_{t=0}^{T} \min\left(\frac{\gamma_1(t)}{t_1}, \frac{\gamma_2(t)}{t_2}\right)$$

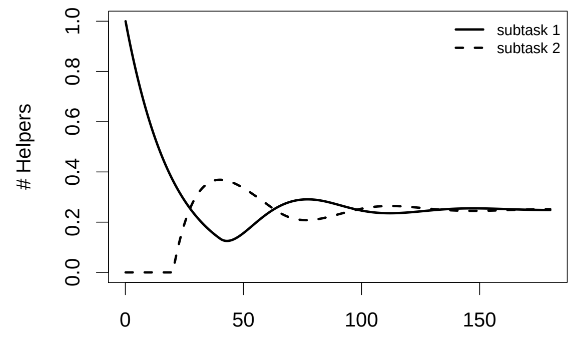
The maximal performance that can be achieved for $t_1 = t_2 = 1$ is $\phi(T) = 1/2$. In this case 50% of the helpers work for each type of subtask and no helper is in reconfiguration mode.



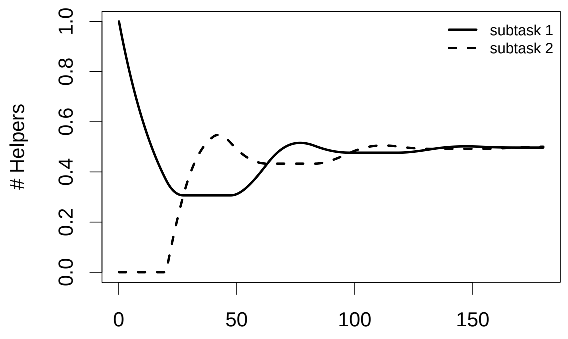
(a) `simple`, $p = 0.01$



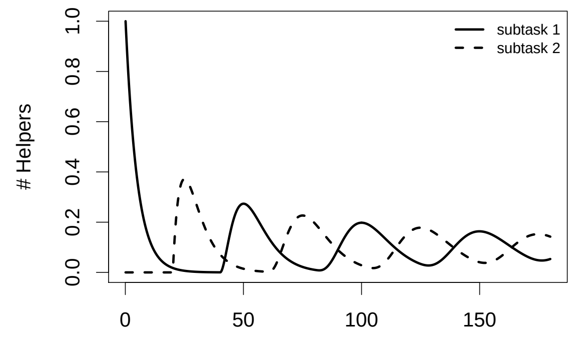
(b) `wait`, $p = 0.01$



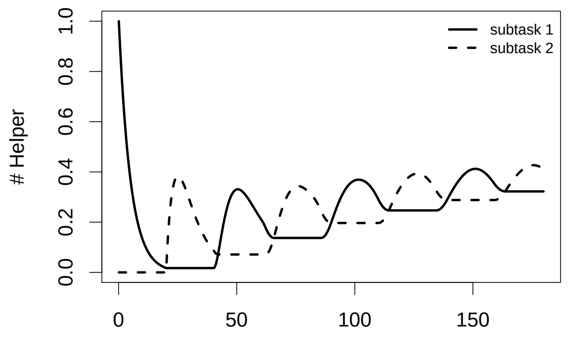
(c) `simple`, $p = 0.05$



(d) `wait`, $p = 0.05$



(e) `simple`, $p = 0.2$



(f) `wait`, $p = 0.2$

Figure 4.27: Solving the initial value problem; comparison for number of active helpers ($\gamma_1(t)$ and $\gamma_2(t)$) in systems using reconfiguration strategy `simple` and `wait`

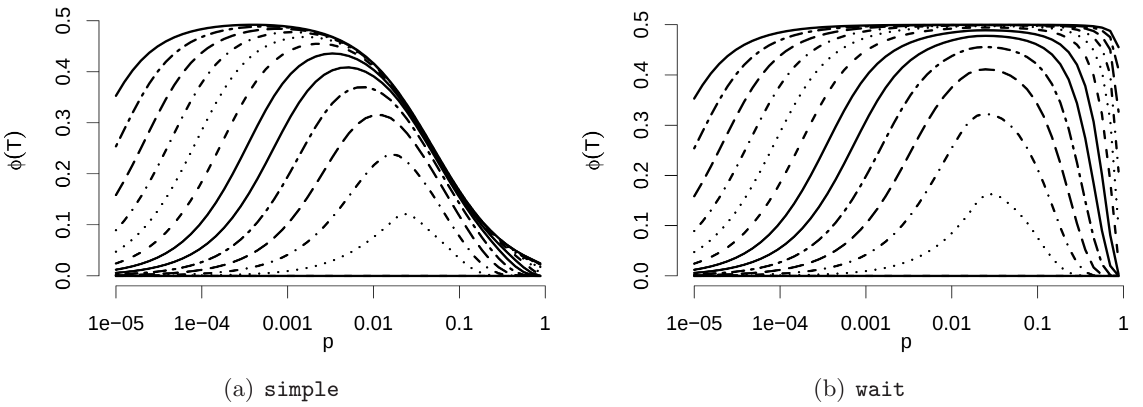


Figure 4.28: Solving the initial value problem; depicted is the normalized performance $\phi(T)$ measured over interval $[0, T]$ with $T \in \{20 \cdot 2^i, i \in [0 : 12]\}$ ($i = 0$: bottom line $i = 12$: top line) for the two different reconfiguration strategies

The normalized performance $\phi(T)$ is depicted in Figure 4.28 for both reconfiguration strategies and different length of time intervals $T \in \{20 \cdot 2^i \mid i \in [0 : 12]\}$ for the case $t_1 = t_2 = 1$. It can be seen, that a good performance can be achieved for strategy `simple` only if p is small (less than 0.01). For $p = 0.1$ the maximum normalized performance $\phi(T)$ is only 0.173. In contrast to this, for strategy `wait` the normalized performance $\phi(T)$ converges to 1/2 for each value of p when T is getting larger. Unfortunately, we could not derive a general explicit form for the normalized performance $\phi(T)$ analytically. Nevertheless, an analytical analysis has been done for the case $T = 2 \cdot \tau$ and both strategies. We present the analysis for strategy `wait` with subtask execution times $t_1 = t_2 = 1$ in the following. In the beginning all helpers are specialized for subtask 1 (i.e., $\gamma_1(t) = 1$ and $\gamma_2(t) = 0$ for $t \in [-\tau, 0]$). Hence, $\gamma_2(t) = 0$ and $\phi(t) = 0$ holds for $t \in [0, \tau]$. Thus, an explicit form for γ_1 can easily be derived by solving the differential equation $\frac{d\gamma_1}{dt} = -p\gamma_1(t)$ with initial value $\gamma_1(0) = 1$. The solution is $\gamma_1(t) = e^{-pt}$. Since $p \cdot \gamma_2(t - \tau) = 0$ holds for $t \in [0, 2 \cdot \tau]$ this solution is valid in the interval $t \in [0, \min(2 \cdot \tau, \chi)]$ where χ is the first intersection point of γ_1 and γ_2 with $t > 0$. Considering γ_2 on the interval $[\tau, \chi]$ we have to solve the differential equation $\frac{d\gamma_2}{dt} = -p\gamma_2(t) + pe^{-p(t-\tau)}$ with initial value $\gamma_2(\tau) = 0$. The solution $\gamma_2(t) = 1 - pe^{-p(t-\tau)}$. Since there is only one intersection of γ_1 and γ_2 possible for $t \in [0, 2 \cdot \tau]$ the normalized performance $\phi(2 \cdot \tau)$ can be computed as $1/(2 \cdot \tau)(\int_{t=\tau}^{\chi} \gamma_2(t) + (2 \cdot \tau - \chi) \cdot \gamma_1(\chi))$. For larger T an explicit solution was not possible.

Numerically it can be shown (compare Figure 4.28), that the normalized performance for strategy `wait` reaches its maximum value for the value $p \approx 0.025$ for every investigated value of T. This shows that there is no need to adapt the parameter p in systems that use the strategy `wait`, regardless of low or high dynamically changing environmental conditions.

4.5 Networks with Reconfigurable Helper Units

The task allocation model investigated in Section 4.2 assumed that the workers choose randomly a helper to request service. But in real systems which have a decentralized organization the workers and helpers might be connected via a network without directly knowing about each other. In this section we study a more complex scenario by assuming a network infrastructure that consists of worker nodes, routers, and helper nodes. If the workers need service they create request packets that include information about the resource demands of the service task they need to be executed. The service request packets are send into the network and are forwarded (randomly) by the routers to the helpers. The helpers receiving the requests can decide about accepting them or forwarding them into the network again.

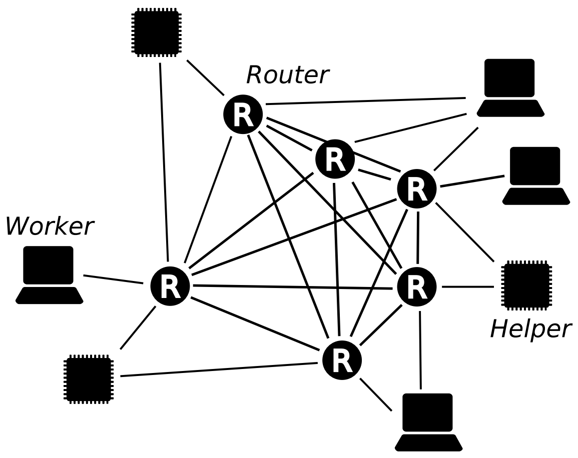


Figure 4.29: Model of the computing system; Workers and Helpers are connected via a fully connected network of routers

The aim of the system is to execute as many service tasks as possible while maintaining small overall reconfiguration costs for the helpers. We use the network, i.e., the routers, to help to organize the task allocation, based on the decentralized clustering algorithm as presented in Section 2.2. By clustering the service requests into groups of tasks with similar resource demands, the helpers can specialize to tasks of a certain cluster and will have only small reconfiguration costs. We compare this method with two simple task allocation methods that only take the local reconfiguration cost into account.

4.5.1 System Model

In the model presented in Section 4.2 it was assumed that a helper can execute a task on a fraction of its resources. The more resources allocated to the specific task type the faster this type of service tasks can be executed in this model. In Section 4.4 we investigated how to partition tasks in the organic support system, if tasks are to large to be executed on one helper. In this case the helper have to use all of their resources for executing the subtask they are specialized to. In the following we assumed that in order to execute a task, the helper has to use all its resources and meet exactly the resource demands of the service task, i.e., the needed configuration of slices. The execution time of the tasks is assumed to be one and is the same for all tasks. On the other hand, the reconfiguration time depends on the number of resources that must be reconfigured.

Each packet P_i has an actual hop counter m_i. The hop count gives the number of helpers that have been visited by the packet. If a packet is rejected by a helper this counter is increased by one. If the counter of a packet exceeds a threshold value TTL (time to live) the service request packet is dropped. The fraction of dropped packets (in relation to all packets) is called the (packet) drop rate.

Let D be the number of different resources needed for executing a service task on a helper. The i-th service request packet P_i in the computing system is characterized by a vector $v_i := (v_i^1, \ldots, v_i^D)$ that describes the resources needed for the corresponding service task, i.e., the helper configuration that is required to execute the service task. For the sake of convenience we denote $P_i = (v_i, m_i)$. Note, that w.l.o.g we can assume $\sum_{j=1}^{D} v_i^j = 1$. If a service task with resource demand v_i is to be executed by a helper, this helper node must be configured such that the fraction v_i^1 (respectively $v_i^2, \ldots, v_i^D$) of its slices is mode 1 (respectively mode 2, mode 3, …, and mode D). Hence, if $w_j = (w_j^1, \ldots, w_j^D)$ with $\sum_{j=1}^{D} w^j = 1$ is the actual configuration of helper H_j (w^h is the fraction of slices that are configured in mode h), then $||v_i - w_j||$ denotes the costs for a reconfiguration of helper H_j from its actual configuration to the new configuration that is required by service packet i.

In the following it is assumed that service request packets P_i determine $D = 3$ resource demands, i.e., v_i is a three dimensional vector ($v_i = (v_i^1, v_i^2, 1 - v_i^1 - v_i^2) \in [0,1]^3, v_i^1 + v_i^2 \leq 1$). We define the measure $||.||$ for costs of the reconfiguration of H_j from configuration $(w_j^1, w_j^2, 1 - w_j^1 - w_j^2)$ to configuration $(v_i^1, v_i^2, 1 - v_i^1 - v_i^2)$, as the number of slices which have to be changed:

$$\max(|v_i^1 - w_j^1|, |v_i^2 - w_j^2|, |(1 - v_i^2 - v_i^1) - (1 - w_j^2 - w_j^1)|). \qquad (4.11)$$

When $D = 3$, configurations can be visualized as a point in an equilateral triangle with height 1 (see Figure 4.30). For every point in the triangle the sum of the distances to the right, bottom, and left line is 1. Let the distance from the bottom (respectively left and right) line equal v^1 (respectively v^2 and $1 - v^1 - v^2$).

The system model is investigated in a step-based simulation. The simulation steps are realized in two main phases. Within the first phase all communication operations take place. We do not explicitly model the worker components in our simulation model. Instead

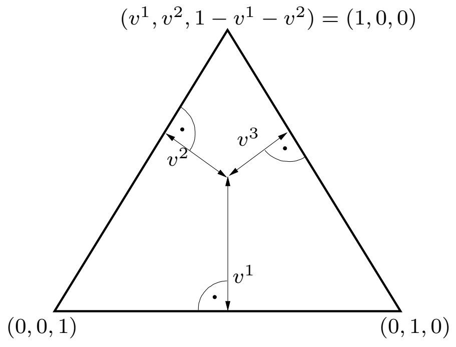


Figure 4.30: The resource requirements of a service request (v^1, v^2, v^3) with $v^3 := 1-v^1-v^2$ are depicted within an equilateral triangle with height 1

a fixed number of service request packets with random resource demands are generated and send to random routers. The number of newly created service packets per time step is also called the (packet) arrival rate. Moreover in the first phase all operations in the network are performed with all packets that are currently in the network. Especially, if the used task allocation method applies the d-DPClust algorithm for clustering the request packets, this is also done in the first phase by the routers of the network. In the second phase of the simulation step all helpers accept or reject the request packets according to the implemented task allocation method and, if necessary, the reconfiguration of the helpers is done.

4.5.2 Simple Task Allocation Method (S-TAM)

The simple task allocation method (S-TAM) is straightforward: The decision on acceptance or rejection of a service request is based on the locally measured reconfiguration costs that would occur if the request is accepted. If the costs would exceed a given threshold, the helper rejects the request. Formally, each helper H_j has an associated parameter r_j called acceptance distance. Let w_j be the actual configuration of helper H_j. A service request packet P_i is accepted if the reconfiguration cost do not exceed the acceptance distance, i.e., $||v_i - w_j|| \leq r_j$. If the reconfiguration is considered to be too expensive, i.e., $||v_i - w_j|| > r_j$, then packet P_i is rejected by helper H_j and is sent to another node in the network or dropped. Using small parameter values r_j leads to many rejected requests and therefore to a high drop rate. On the other side, large values of r_j cause high total reconfiguration costs. Hence, a tradeoff exists between the drop rate and the reconfiguration costs.

We extend this approach to make it adaptive in order to cope with dynamically changing resource demands. This is done by allowing the thresholds r_j, that are used for the decision of accepting a request, to change over time. Helpers locally estimate the number of rejected requests of the system and change their acceptance threshold based on the estimated system behavior. The following method is used for an adaptable version of S-TAM, denoted as A-S-TAM:

A helper H_j can change r_j only due to a local estimation of the drop rate of the system. This estimation is done by using simple exponential smoothing on the locally observed drop rate of a helper. If the hop counter of a requesting packet is zero ($m_i = 0$), then the corresponding helper is the first helper that was requested for help. If the hop counter of a packet is TTL ($m_i =$TTL), then the corresponding helper is the last one that is requested for help, i.e., if this last helper does not accept the service request packet, the packet will be dropped.

We equip every helper with an estimation $t_{\max}$ (respectively t_0) of the fraction of packets in the system that have a hop counter of TTL (respectively zero). If a packet with $m_i =$TTL arrives at the helper, the estimation $t_{\max}$ is modified according to $t_{\max} := \rho \cdot t_{\max} + (1 - \rho)$, otherwise $t_{\max}$ is changed according to $t_{\max} := \rho \cdot t_{\max}$. The parameter ρ determines the influence of recent packet requests. Note, that $t_{\max}$ tends towards 1 if only packets with a maximal hop counter are observed, and $t_{\max}$ tends towards 0 if no such packets request arrive. Analogously, the smoothed estimation t_0 of the fraction of packets with hop counter zero is modified according to $t_0 := \rho \cdot t_0 + (1 - \rho)$ when the hop counter of an arriving packet is 0 and $t_0 := \rho \cdot t_0$ otherwise. A locally estimated measure for the percentage of dropped packets is then determined by $d := t_{\max}/t_0$. Let γ be the drop rate, that should not be exceeded by the A-S-TAM (γ is a parameter of A-S-TAM). If a helper identifies an estimated drop rate d in the system, that is too high (i.e. $d > \gamma$), then it increases its acceptance distance by a factor r_+. If a helper identifies a drop rate in the system, that is too small (i.e., $d \leq \gamma$), then it decreases its acceptance distance by a factor $r_- := 1/r_+$. The pseudo code of A-S-TAM that is executed in one simulation step in each helper is given in Algorithm 5.

4.5.3 Clustering based Task Allocation Method (C-TAM)

In the C-TAM method the service packets that are sent through the network are clustered according to their resource needs. For this purpose the Decentralized Packet Clustering introduced in Section 2.2 is used. The clustering puts packets for services that have similar resource requirements into the same cluster. Each helper can specialize to service requests from one of these clusters and preferably perform only the requests. Since the resources that are needed for the service tasks within one cluster are similar with respect to their resource demands, this will lead to small reconfiguration costs.

To cluster the service requests the decentralized clustering algorithm d-DPClust$_{zc}$ is used. Recall that for this method the packets contain a cluster number additionally to

Algorithm 5 A-S-TAM in each helper (one simulation step)

```
1: GIVEN:
      $w$: current configuration state of the helper
      $r$: current acceptance distance of the helper
      $\gamma$: drop rate not to be exceeded
      $t_{\max}$: smoothed estimation for the fraction of packets with $m_i = \text{TTL}$
      $t_0$: smoothed estimation for the fraction of packets with $m_i = 0$
      $\rho$: influence of old estimations
      $\mathcal{I}$ : set of indices of request packets $P_i = (v_i, m_i), i \in \mathcal{I}$
      TTL: time to live
2: accept:=TRUE
3: for $i \in \mathcal{I}$ do
4:    $t_{\max} := \rho \cdot t_{\max}$
5:    if $m_i == \text{TTL}$ then
6:       $t_{\max} := t_{max} + (1 - \rho)$
7:    end if
8:    $t_0 := \rho \cdot t_0$
9:    if $m_i == 0$ then
10:      $t_0 := t_0 + (1 - \rho)$
11:   end if
12:   $m_i := m_i + 1$
13:   compute costs needed for reconfiguration $c := ||w - v_i||$
14:   if c≤r and accept==TRUE then
15:      packet $P_i$ is accepted:
16:      - reconfigure helper: $w := v_i$
17:      - execute $v_i$ in this simulation step
18:      - do not accept any further packets in this step: accept:=FALSE
19:   else
20:      packet $P_i$ is rejected:
21:      if $m_i > \text{TTL}$ then
22:         drop $P_i$
23:      else
24:         send $P_i$ to another helper
25:      end if
26:   end if
27: end for
28: estimate drop rate: $d := t_{\max}/t_0$
29: if $d > \gamma$ then
30:    increase acceptance distance: $r := r \cdot r_+$
31: else
32:    decrease acceptance distance: $r := r/r_+$
33: end if
```

Algorithm 6 C-TAM in each helper (one simulation step)

```
1: GIVEN:
       $w$: current configuration state of the helper
       $c_h$: current cluster number of the helper
       $p$: probability of cluster number changing
       $\mathcal{I}$: set of indices of request packets $P_i = (v_i, c_i, m_i), i \in \mathcal{I}$
2: accept:=TRUE
3: while $i \in \mathcal{I}$ do
4:     set $c_h := c_i$ with probability $p$
5:     if $c_h == c_i$ and accept==TRUE then
6:         packet $P_i$ is accepted:
7:         - reconfigure helper: $w := v_i$
8:         - execute $v_i$ in this simulation step
9:         - do not accept any further packets in this step: accept:=FALSE
10:    else
11:        packet $P_i$ is rejected:
12:        if $m_i ==$ TTL then
13:            drop $P_i$
14:        else
15:            $m_i := m_i + 1$
16:            send $P_i$ to another helper
17:        end if
18:    end if
19: end while
```

the data vector. Formally, a service request packet $P_i = (v_i, c_i, m_i)$ for the C-TAM is thus characterized by v_i (the vector that describes the resources needed), the associated cluster number $c_i \in \{1, \dots, n_c\}$ and the hop counter m_i.

Each helper in the network has an associated number c_h relating to the cluster the helper is actually specialized to. If the cluster number of a service request packet that is received by a helper is identical to its c_h value, the service task is executed by the helper. If, on the other hand, the c_h is different from the packet's cluster number, there is a fixed probability p that the helper changes its specialization to the cluster of the packet. If the helper does not change its specialization the service request is rejected and the packet is sent to another node in the network. If the hop counter of the packet equals the time to live the packet is dropped. Note, that a service request is also rejected if the helper is already executing another service request at the same simulation time step. The pseudo code of C-TAM is given in Algorithm 6. Note, that the cluster number of a request packet is changed only by the routers according to the estimated centroids of the clusters. Hence, this is not a part of the algorithm that is executed in each helper.

4.5.4 Influence of Parameters on C-TAM

The experimental setup and parameters for the simulation of the model were the following. If not stated otherwise all test runs in this section were performed in a scenario, where all service requests are from up to four different areas, called classes, of the configuration space. A snapshot from a typical test scenario is depicted in the right part of Figure 4.30.

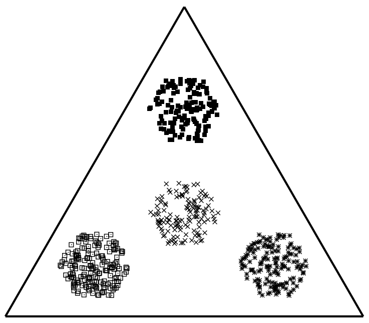


Figure 4.31: Resource requirements of service requests of four different classes

Note, that a partitioning of the service requests that leads to small costs is not given in advance, as packets have a random cluster identity when they are created. Within each class of service requests the individual requests are chosen uniformly distributed. The center of request class 1 is $(1/3, 1/3, 1/3)$, the center of classes 2 (respectively 3 and 4) are $(2/3, 1/6, 1/6)$ (respectively $(1/6, 2/3, 1/6)$ and $(1/6, 1/6, 2/3)$). Clearly this is an artificial problem instance and realistic requests will have more dimensions and not regularly or even completely randomly distributed resource requirements. But the main properties of the problem are given: distributed requests in a reconfiguration space and an according similarity measure.

If not stated otherwise in each simulation step 50 service requests packets were sent into the network. 50 helpers components and 50 routers nodes were used. The probability p that a helper changes its cluster was set to 0.01 (if not stated otherwise). Parameter β that influences the update of the centroid estimation in a router was set to 0.1. If a centroid estimation has not changed by the last 100 packets that arrived at a router, the new centroid estimation is set to the corresponding configuration of the next arriving packet. Each result that is given in the following is averaged over 10 simulation runs, i.e., each pair of cost/drop values is averaged over 10 independent simulations. Simulation runs were performed over 10 000 steps. The shown reconfiguration costs are the overall reconfiguration costs that were spent by the helpers when executing the service requests (Equation 4.11) divided by the number of all service requests, that have been created during a simulation run and the number of simulation steps.

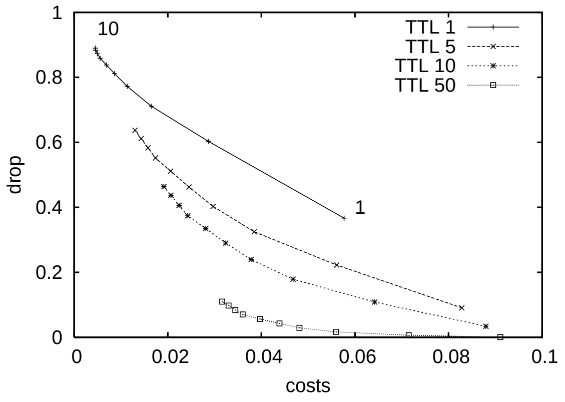
Number of Clusters

We investigated the behavior of C-TAM with respect to the drop rate of the service request packets and the reconfiguration costs for test runs with different number of service request classes. For 1 request class (class 1), 2 request classes (classes 3 and 4), or 4 request classes (all classes) the number of clusters that are used by the decentralized clustering algorithm has been varied with $n_c \in \{1, 2, \dots, 10\}$ (see Figure 4.32(a), 4.32(c), and 4.32(e)). The results are depicted in the left column of Figure 4.32 when using a maximal life time of the packets TTL $\in \{1, 5, 10, 50\}$. It can be seen that there is a clear trade-off between the drop rate and the reconfiguration costs. When using a larger TTL value, the drop rate is reduced significantly. The reduction of the reconfiguration costs for an increasing number of clusters n_c depends strongly on the number of request classes. When 2 or 4 request classes are used there is a sharp bend in the corresponding curves, as the algorithm utilizes its adaptability. When n_c is smaller than the number of request classes, then some helpers have to execute service request of more than one class. This leads to relatively high reconfiguration costs as can be seen in Figures 4.32(c) and 4.32(e), where packets from 2 or 4 service request classes were put into the network. For example, when 2 request classes and TTL=5 are used, the costs are reduced from 0.28 when using $n_c = 1$ to 0.07 when $n_c = 2$ is used. A further increase of n_c (larger than the number of service request classes) reduces the costs only slightly. The small reduction results from the fact that the service requests within one class vary slightly with respect to their resource requirements. Therefore the reconfiguration costs of the helpers can be reduced slightly when the service requests of one class are split into several clusters. The disadvantage is that the packet drop rate increases with a higher number of cluster.

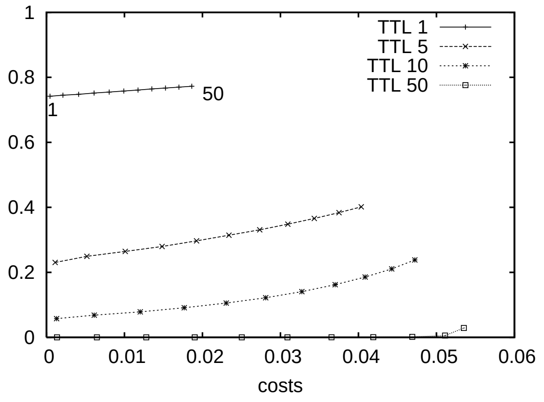
Work Load

In the following we compare simulations for the C-TAM where the computing system has different work loads, simulated by using the different arrival rates $\{1, 5, 10, 15, \dots, 50\}$. The number of clusters for the decentralized clustering algorithm was set to $n_c = 4$ and similar to Subsection 4.5.4 the number of service request classes was 1,2, or 4. The results are depicted in the right column of Figure 4.32.

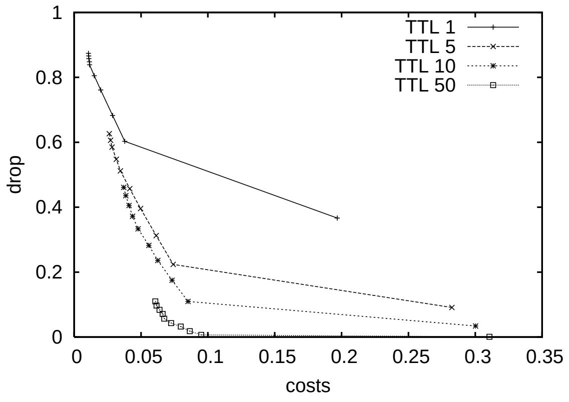
Obviously, when using a very small (and unrealistic) value of TTL=1 the drop rate is very high (always larger than 0.69). This value is interesting because it shows the average fraction of packets that are not executed by a single helper. The small number of service requests that are executed leads only to small reconfiguration costs. When using a higher TTL the drop rate decreases significantly, e.g., for TTL=5 it is less than 0.3 in all cases. The increase in reconfiguration costs is relatively small in this case (less than 0.13 when using 4 service request classes and an arrival rate of 10). When the value of TTL is 50



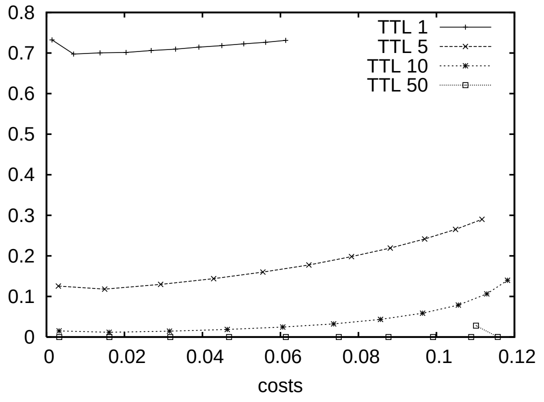
(a)



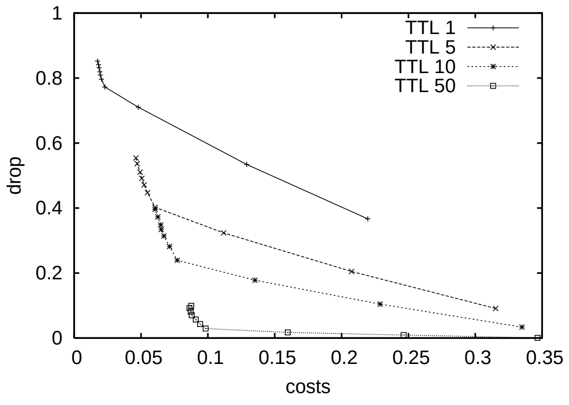
(b)



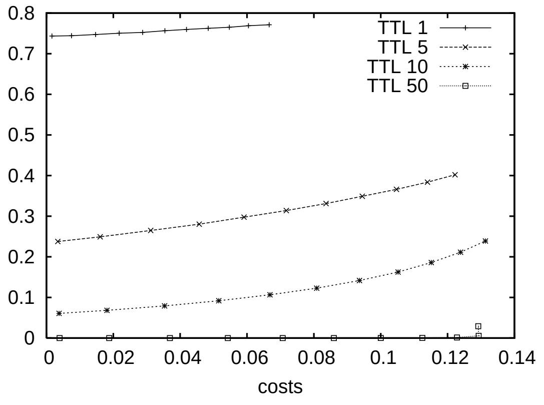
(c)



(d)



(e)



(f)

Figure 4.32: C-TAM: Drop rate/reconfiguration cost trade-off for different scenarios; left column: dots on lines correspond to number of clusters $n_c \in \{1, \ldots, 10\}$ (from left to right); numbers at dots indicate number of clusters used; right column: dots on lines correspond to arrival rates of $\{1, 5, 10, 15 \ldots, 50\}$ (from right to left) packets per simulation step; numbers at dots indicate arrival rate; number of service request classes: 1 (top), 2 (middle), 4 (bottom)

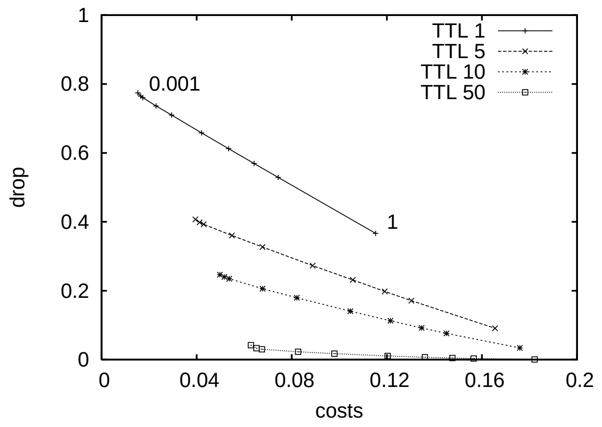


Figure 4.33: C-TAM: Drop rate/cost trade-off for different probabilities p that a helper changes its cluster when an arriving packet has a different cluster number; $p \in \{0.001, 0.005, 0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 1.0\}$ (from left to right)

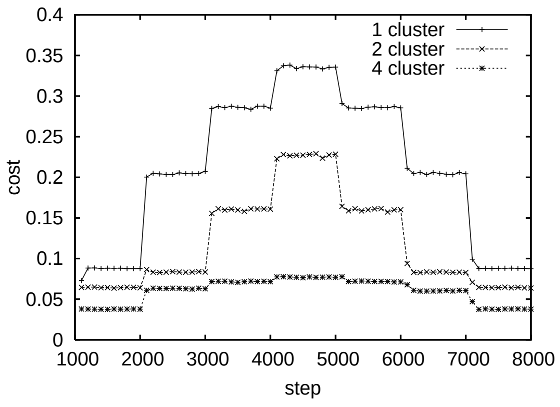
nearly no packets are dropped in all the investigated scenarios. Also the reconfiguration costs are small in this case (always < 0.13).

Changing Cluster Number of Helper Units

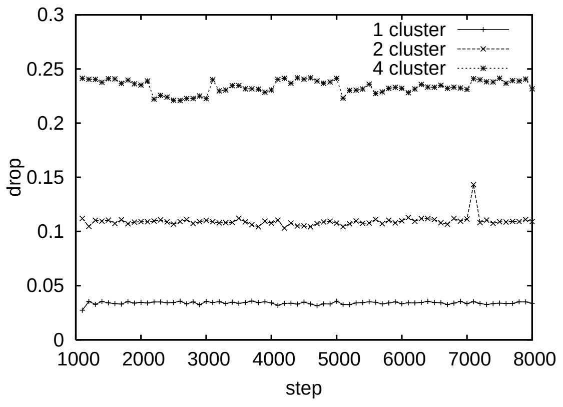
A strong influence on the adaptability of the helpers has the parameter p, which is the probability that a helper specializes to the cluster of an arriving packet. When p becomes larger the number of rejected packets decreases and the reconfiguration costs increase. Note, that when using $p = 1$ no service request is rejected due to its cluster identity (only when the helper is executing another service request a packet is rejected). Rejecting a large number of packets leads to an increased drop rate. Drop rates and reconfiguration costs were measured when using a cluster changing probability of $p \in$ {0.001, 0.005, 0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 1.0}. The results are depicted in Figure 4.33 for TTL values of 1, 5, 10, and 50. The smaller the TTL values the stronger the decrease of the drop rate with increasing p. E.g, when using TTL=5 the drop rate is decreased from 0.4 (for $p = 0.01$) to 0.1 (for $p = 1$). But for high values of p the reconfiguration costs become large (they increase from 0.040 to 0.17 for TTL=5 when p increases from 0.01 to 1).

Dynamically Adding and Removing Request Classes

To show the adaptability of C-TAM a dynamic scenario was investigated where the set of service request classes changes. Starting with only one request class (class 1) we successively added classes 2, 3, and 4 every 1000 simulation steps (i.e., packets of the corresponding classes are created and send to the network). After that, the classes $2, 3$, and 4 were removed again successively every 1000 steps. In Figure 4.38 the results are depicted



(a)



(b)

Figure 4.34: C-TAM: Reconfiguration costs (a) and drop rate of packets (b) shown over a simulation run where service request classes are added successively (simulation steps 2000, 3000, and 4000) and then removed (simulation steps 5000, 6000, and 7000); initially (steps 0-999) only one service request class is used; results are given for $n_c \in \{1, 2, 4\}$

for $n_c \in \{1, 2, 4\}$ clusters. When the number of cluster is set to one each additional request class increases the reconfiguration costs significantly, as the helper have to be reconfigured between the different service request classes very often. When dividing the requests into $n_c = 4$ clusters, the average reconfiguration costs are much smaller. The additional reconfiguration costs that occur after a new class has been added are due to the fact, that request classes have to be partitioned with less clusters (or are not partitioned at all). This leads to higher intra-class reconfiguration costs. These reconfiguration costs are much smaller than the inter-class reconfiguration costs. These results show the fast adaptive behavior of the decentralized clustering component based on DPClust in the given scenario.

4.5.5 Comparison of C-TAM, S-TAM, and A-S-TAM

In this subsection we only use one request class with center $c = (1/3, 1/3, 1/3)$ and investigate the influence of the (dynamic) change of the size of this class. Similar as in the previous Subsection for a new service request $(v_i^1, v_i^2, 1 - v_i^1 - v_i^2)$ the value v_i^1 (respectively v_i^2) is chosen randomly from the interval $[c^1 - \Delta, c^1 + \Delta]$ (respectively $[c^2 - \Delta, c^2 + \Delta]$). Here different values for Δ are used. The maximal value for Δ, namely $\Delta_{\max}$ was set to 1/3. Typical snapshots of a request class with maximal size ($\Delta = 1/3$) and minimal size ($\Delta = 1/30$) are given in Figure 4.35.

Note, that in this subsection relative reconfiguration costs will be used to make the reconfiguration costs of request classes with different value of Δ comparable and summable. More exactly, if the value of Δ for the request class is reduced by a factor k (relative to its

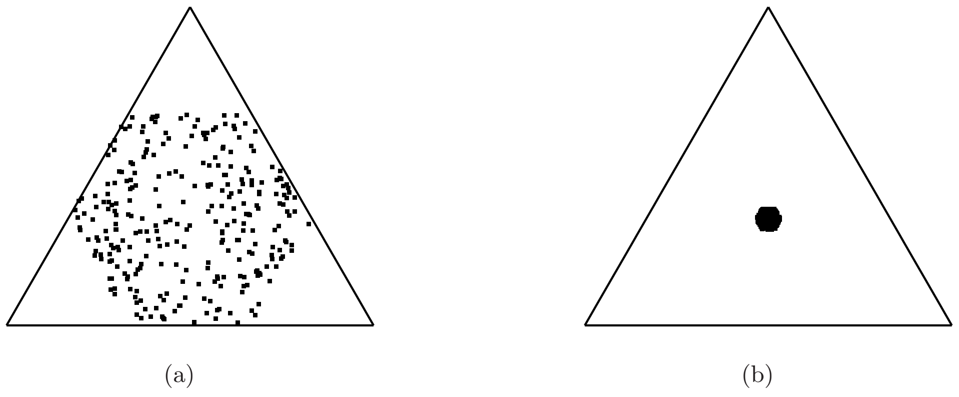


Figure 4.35: Snapshot for resource requirements of a service request (v^1, v^2, v^3); one class of service requests; service request class is scaled over time from (a) $\Delta = 1/3$ to (b) $\Delta = 1/30$

maximal size for $\Delta = 1/3$), the reconfiguration costs are multiplied by the same factor k. If not stated otherwise all further experimental settings were chosen as in Section 4.5.3. The parameter ρ for the estimated drop rate was set to 0.99.

Influence of Δ

We first investigated the performance of C-TAM and S-TAM for different but fixed values of $\Delta \in \{1/30, 2/30, \ldots, 1/3\}$. C-TAM was run with $1, \ldots, 10$ clusters. For S-TAM we used 6 different acceptance distances $r \in \{0.01, 0.02, 0.05, 0.1, 0.2, 0.5\}$. For each simulation run the drop rate and the relative reconfiguration costs were measured within 10 000 simulation steps. Results are depicted in Figure 4.36. The three dimensions of the figure are the drop rate, the relative reconfiguration costs per service request, and the value of parameter Δ. It can be clearly seen that the performance of C-TAM is independent of Δ: the drop rate and the relative reconfiguration costs are nearly identical for all 10 different values of Δ. Depending on a predefined acceptable drop rate (or predefined relative reconfiguration costs) the number of clusters in the C-TAM should be chosen. While using only one cluster leads to a drop rate of 0.04 and relative reconfiguration costs per request of 0.691, using 10 clusters increases the drop rate to 0.486 but decreases the relative reconfiguration costs per request to 0.126. In contrast to that, S-TAM is strongly dependent on Δ. For example when using the acceptance distance $r = 0.1$ the drop rate ranges from 0.554 (when using $\Delta = 1/3$) to 0.037 (when $\Delta = 1/30$ was applied).

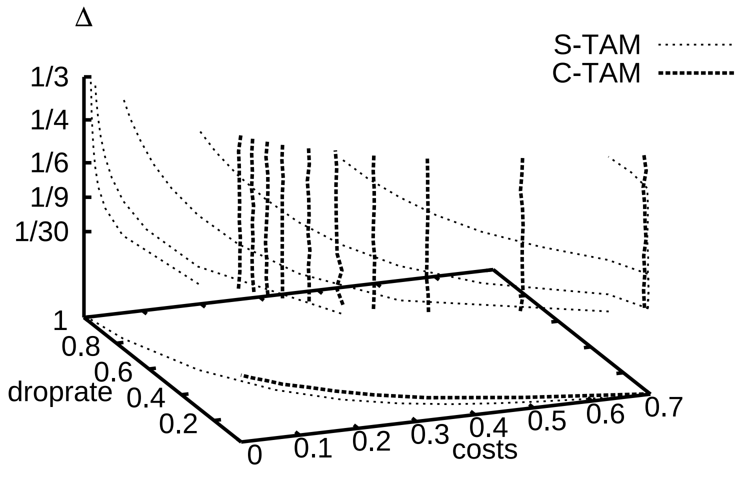
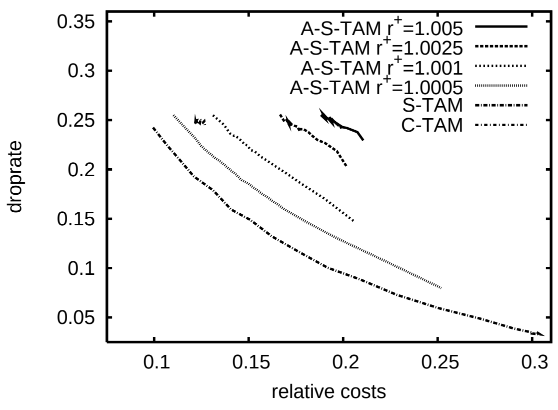


Figure 4.36: C-TAM and S-TAM: Reconfiguration costs and drop rate; S-TAM: using fixed acceptance distances $r \in \{0.01, 0.02, 0.05, 0.1, 0.2, 0.5\}$, C-TAM: number of clusters $1, \ldots, 10$; simulation runs with 10 different request classes ($\Delta \in \{1/30, 2/30, \ldots, 1/3\}$); the smaller the number of clusters in C-TAM, the smaller the drop rate becomes; the larger the acceptance distance r in S-TAM, the smaller the drop rate becomes; the projection on the bottom plane is determined from the results for $\Delta = 1/3$

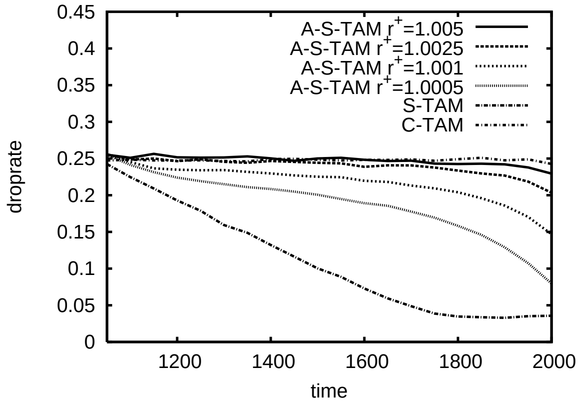
Dynamically Changing the Value of Δ

In contrast to S-TAM, in the A-S-TAM method the acceptance distances in the helpers are increased if the drop rate becomes too large. Hence, A-S-TAM may handle many scenarios much better. This is studied in the following experiment. For the first 1000 iterations Δ had a constant value of $1/3$. The request class was then shrunk by linearly changing the value of Δ from $1/3$ to $1/30$. This linear decrease of Δ was done from iteration 1001 to iteration 2000. (This leads to an overall value of 100 000 service request from time step zero).

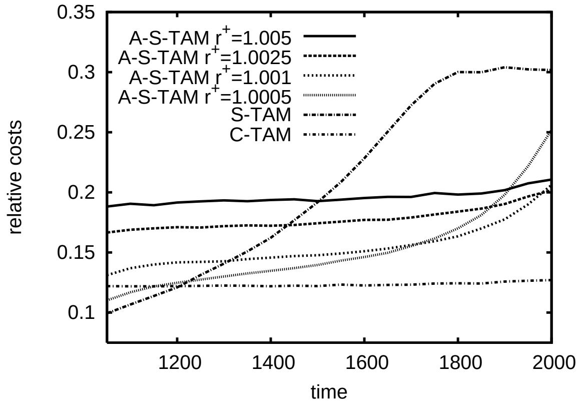
In Figure 4.38 the drop rate and the relative reconfiguration costs are given over time (simulation steps 1001 to 2000) for i) C-TAM when using 4 clusters, ii) S-TAM with acceptance distance $r = 0.2$, and iii) A-S-TAM with parameter r_+ for adapting the acceptance distance with $r_+ \in \{1.0005, 1.001, 1.0025, 1.005\}$. The threshold for the drop rate to be achieved was set to $\gamma = 0.25$, i.e., if the locally measured drop rate d is larger (respectively smaller) than 0.25, the acceptance distance of the corresponding helper is increased (respectively decreased). As seen in the previous subsection, the C-TAM is nearly independent of Δ and the drop rate (respectively relative reconfiguration costs) remains basically



(a)



(b)



(c)

Figure 4.37: A-S-TAM, S-TAM, and C-TAM: (a) Reconfiguration costs and drop rate; (b) drop rate and (c) reconfiguration costs over time; dynamically changing the request packets: until iteration 1000 Δ was fixed to 1/3, from iteration 1001 to 2000 Δ was decreased linearly from 1/3 to 1/30; C-TAM using 4 clusters; S-TAM using acceptance radius 0.2; A-S-TAM with different values for adaptation factor r_+

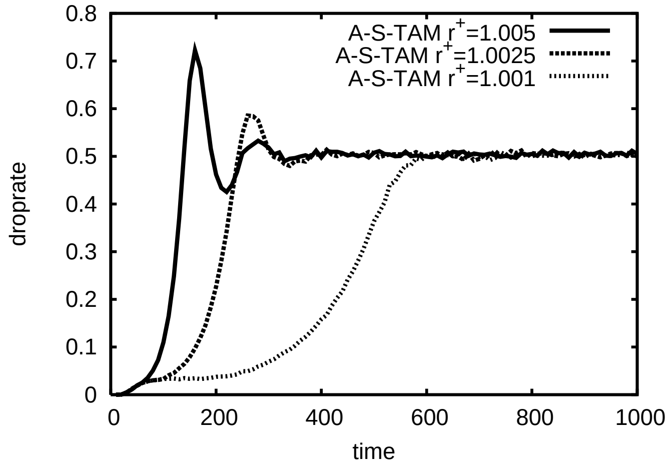


Figure 4.38: A-S-TAM: drop rate over time when using different adaptation values $r_+ \in \{1.001, 1.0025, 1.005\}$

constant at ≈ 0.25 (respectively ≈ 0.12). Depending on Δ, S-TAM (or A-S-TAM with $r_+ = 1.001$) may perform better than C-TAM, but the relative reconfiguration costs are increased from ≈ 0.1 (time step 1001) to ≈ 0.3 (time step 2000), which clearly shows the lack of an adaptive behavior. With increasing value of r_+ A-S-TAM becomes more adaptive, but the overall performance gets worse. For $r_+ = 1.005$ the relative reconfigurations costs are always larger than 0.18.

Furthermore, r_+ in the A-S-TAM has to be chosen with care. If r_+ is too large, then the average acceptance distances of the helpers may underestimate (respectively overestimate) the real drop rate. The reason is, that the influence of changing the acceptance distance needs some time to show effect. In such a case the acceptance distances will be decreased (respectively increased) too much. This will lead to a too large (respectively too small) drop rate and the system shows an oscillating behavior as shown in Figure 4.38. In the given scenario the aimed drop rate γ is 0.5 and all helpers start with an acceptance distance of 1. All other parameters were chosen according to the default values. For $r_+ = 1.005$ and $r_+ = 1.0025$ it can be clearly seen, that the aimed drop rate is not achieved as smoothly as for $r_+ = 1.001$, but only by an oscillating behavior of the acceptance distances in the helpers, that lead to the depicted oscillations of the drop rate.

Summarizing the empirical investigation we can say, that the C-TAM shows a very good adaptive behavior and a very good performance. While achieving nice results for situations where adaptation to the environment is not necessary, the S-TAM fails to show good performance in dynamically changing scenarios. Using the A-S-TAM overcomes this problem, but the parameters of the A-S-TAM have to reflect the degree of dynamics of a given scenario. In contrast to that, the C-TAM achieves a very good trade off for drop rate and reconfiguration costs, which remain basically constant also in case of a strong dynamical change.

4.6 Summary

In this chapter we introduced the idea of Organic Support Systems. These systems consist of a number of so called helper components that are responsible for the execution of service and support tasks for a given worker system. Using reconfigurable hardware the helper components are able to adapt to the actual needs of the supported system by specializing on the required types of support tasks. We studied different aspects of the organization of such systems.

Inspired from models of task allocation in social insects, we introduced a mechanism for the allocation of the support tasks to the helper components. We studied different strategies for the helpers to decide about acceptance of service tasks and about how to reconfigure their resources. For a system of two helpers analytical results were presented. The optimal degree of specialization under different costs for reconfiguration and communication relative to the execution time has been derived theoretically. For systems with a larger number of helpers we presented experimental results. It was shown that these systems can adapt to dynamic situations with changing rates for service requests. For certain parameters the systems can show an oscillation effect, which must be considered carefully if applying the methods.

In the second part of the chapter it was assumed that single service tasks are split into subtasks that have to be executed successively. Helper components decide locally for which subtasks to specialize. An uneven distribution of workload over the helper leads to unnecessary waiting times. It was shown, that a threshold based reconfiguration strategy with exponentially distributed threshold values performs better than simple strategies that use the same threshold for all helpers or use a fixed probability for changing the specialization. Modelling the systems as delay differential equation systems, we were able to proof the stability of a simple probability based strategy. The numerical solution of initial value problems has shown that the theoretical models fit the simulation outcome of the systems well.

The third part of this chapter studied Organic Support Systems in which the workers and helpers are connected via a router based network without directly knowing about each other. Workers send request packets including resource demand information of the service task they need to be executed into the network. We empirically compared a method for task allocation, based on the decentralized clustering algorithm presented in Section 2.2, thoroughly with other allocation schemes. The simulations have shown that the clustering based system has a strong adaptive behavior in static and dynamic scenarios and that the decentralized clustering is able to reduce the reconfiguration costs significantly.

5 Specialization and Cooperation in Systems of Memory Constrained Components

Evolutionary Computation, Machine Learning, and, as a combination of both, Learning Classifier Systems provide powerful tools for the creation of self-organizing and adaptive computing systems. This chapter starts with a brief introduction on how these methods have been applied in Organic Computing so far.

In the second part of the chapter we use interacting Pittsburgh-style Learning Classifier Systems to evolve rule sets for solving classification problems on computing systems consisting of distributed, autonomous, memory constrained components. Using this approach the components become specialists for parts of the classification problem and solve the whole problem in cooperation. A deeper look at the structure and properties of the evolved rule sets and the way the components share their knowledge is taken. The influence of different communication topologies and the consideration of communication costs on the emerging patterns of cooperation and on the obtained classification performance of the whole system is studied.

5.1 Evolutionary Computation and Machine Learning in Organic Computing

To reach the design goals of Organic Computing, i.e., to create autonomous, adaptive, life-like systems, researchers do not solely rely on inspirations from nature, they also use knowledge from other fields of computer science. Especially the fields of *Machine Learning* and *Evolutionary Computation* provide a wide variety of different solutions for creating Organic Systems which can learn and evolve and in this way adapt to changing environments. Independently of initial designs or external interventions such systems can learn about their environment over time, adapt to their user, survive breakdowns and attacks, and react sensibly, even if they encounter a new situation for which they have not been programmed explicitly. In the following we will give an overview on how these methods have been applied to Organic Computing yet.

5.1.1 Evolutionary Computation

Evolutionary Algorithms (EA) mimic biological evolution principles such as reproduction, mutation, recombination, natural selection, and survival of the fittest in order to solve (optimization) problems. In EAs a set of candidate solutions to a problem forms a population. In an iterative process new populations are evolved by repeatedly selecting good solutions and generating new solution based on the selected ones. The selection mechanism acts as a force to increase the quality of the solutions, since the probability for worse individuals to be selected is usually smaller. On the other hand, the generation of the offspring by recombination and mutation creates the necessary diversity and thereby facilitate novelty. For an introduction to evolutionary computation see, e.g., De Jong (2006).

Several applications of EAs in Organic Computing can be found. First there are the aleady mentioned works Branke and Schmeck (2008) and Komann and Fey (2009). Kaufmann and Platzner (2007) proposes an architectural concept for intrinsically evolvable embedded systems. In this approach within a reconfigurable hardware system new configurations are evolved online using EAs and in this way, the system can adapt to dynamic environments. König et al. (2006) proposes the concept of Organic Sensing Systems, based on medium granularity field-programmable mixed-signal arrays. Beside other metaheuristics it is suggested to use genetic algorithms implemented in the unit responsible for the reconfiguration algorithm. Igel and Sendhoff (2008) uses evolutionary algorithms for the design of Artificial Neural Networks that are specialized to certain problem classes. The authors claim that the resulting Neural Networks are able to adapt to a specific problem of this problem class in a very efficient and robust way and that this makes such a "second order learning" interesting for the application in Organic Computing systems.

5.1.2 Machine Learning

Machine Learning as a branch of computer science is concerned with the development of algorithms that allow computers to learn, based on given data, to make intelligent decisions (for an introduction see, e.g., Alpaydin, 2004). An example of the application of machine learning, more precisely of a so called supervised learning method, in Organic Computing is the utilization of Hidden Markov Models in Krüger et al. (2008). This work presents a system consisting of autonomous collaborating units for dynamic gesture recognition relying on Organic Computing principles.

Another branch of Machine Learning is the so called Reinforcement Learning (RL), the computational approach to learn from the interaction with an environment. RL means "…learning what to do - how to map situations to actions - so as to maximize a numerical reward signal …" (Sutton and Barto, 1998). Reinforcement Learning algorithms view the target problem as an unknown environment that provides feedback in terms of a numerical reward signal and attempt to solve the problem by interacting with the environment and by trying to obtain as much reward over time as possible.

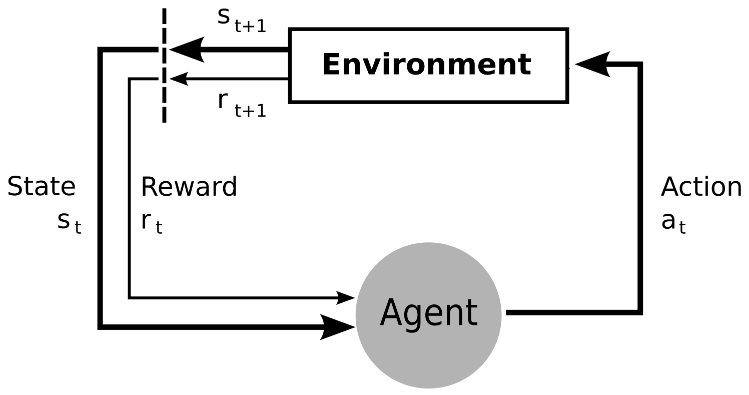


Figure 5.1: Interaction between an agent and the environment in the Reinforcement Learning framework.

Usually the learner respectively decision-maker is called an agent. Everything outside the agent, i.e., the things it interacts with, is called the environment (compare Figure 5.1). The agent interacts with the environment at a sequence of discrete time steps $t = 0, 1, 2, 3 \ldots$. At each time step t, the agent can perceive the state s_t of the environment through its sensors and can perform an action a_t of actions available in state s_t. As a consequence of this action the environmental state changes to s_{t+1} and the agent receives a reward signal r_{t+1}. This reward signal is the most important aspect of RL and separates it for example from supervised learning, since never the correct input/output pairs are presented, nor sub-optimal actions are explicitly corrected.

Reinforcement Learning methods can be used to solve classification problems. Informally spoken, to solve a classification problem is to assign a labeling to a set of objects that fulfills certain criteria. If the properties of the given objects form the environment for the agent and the possible actions represent different classes (labels), the agent can learn to classify (label) objects correctly. For this purpose it is only necessary that correct classifications (actions) lead to a higher reward than incorrect ones.

5.1.3 Learning Classifier Systems

Learning Classifier Systems (LCS) combine Reinforcement Learning and Evolutionary Algorithms and were first described in Holland (1992) (for an overview see, e.g., Sigaud and Wilson, 2007). LCS compute solutions consisting of rules or entire rule sets and apply reinforcement learning to estimate their quality in terms of problem solution. A

Genetic Algorithm is used to discover rules/rule sets that improve the current solutions. There are two main approaches for the design of LCSs, namely the Michigan and the Pittsburgh approach. In Michigan-style learning classifier systems the Genetic Algorithm operates on a set of rules and every rule has an associated fitness value. In contrary, in Pittsburgh style learning classifier systems whole rule sets are evolved (De Jong and Spears, 1991).

Comparing two representatives of these both approaches, namely XCS (Wilson, 1995) as a Michigan-style LCS and GAssist (Bacardit, 2004) as a Pittsburgh style LCS, both systems show comparative performance results (Bacardit and Butz, 2005). While GAssist has the tendency to ignore additional problem complexity by evolving compact rule sets, XCS tends to over-fit the training data. On the other hand GAssist has slight problems with handling many output classes as well as huge search spaces.

Learning classifier systems were used for several applications in Organic Computing. For instance, Bernauer et al. (2008) incorporates a modified XCS into a design methodology for a Autonomic-System-on-a-Chip (ASoC). Implemented in hardware the classifier system can learn to find the optimal operating point (performance, temperature, power consumption, and soft error rate) of an AMD Opteron Quadcore. The resulting system is self-configuring and can adapt to its environment or to unforeseen situations. The Organic Traffic Control approach presented in Prothmann et al. (2008) attempts to find and apply good parameters for traffic light controllers depending on specific traffic demand situations. An Evolutionary Algorithm is used for an off-line optimisation of parameters by simulation. This is combined with a Learning Classifier System that selects and evaluates parameters on-line. A work concerning adaptive network protocol configuration using a modified variant of the XCS system is presented in Tomforde et al. (2009). The classifier system is used to find a parameter sets that ensure the best possible system performance in dynamic environments. Like in the Organic Traffic Control the system evolves new rules offline via simulation. The authors demonstrate the usefulness of the proposed approach by applying the system to a Peer-to-Peer protocol and evaluate the achieved results. Schöler and Müller-Schloer (2005) uses Fuzzy-XCS a fuzzy variant of the XCS classifier system to monitor the status of an adaptive protocol stack for mobile terminals. The classifier system is able to detect protocol stack performance degradation and can modify the parameters of the system in order to establish normal performance again. Richter et al. (2008) reduces the complexity of learning tasks by dividing it into smaller sub-problems. An Organic Computing related multi-agent scenario is used to show improvements in learning speed.

Most of these examples have a centralized system structure. But Organic Computing systems often consist of many loosely coupled components which have to self-organize and cooperate to reach a given system goal. If these components need to exhibit (learned) knowledge about a specific problem, the question arises how to organize the way they work together in a reasonable way.

Plenty work on related questions usually referred to as cooperative multi agent learning (for an overview see, e.g., Panait and Luke (2005)) exists. Specifically in the field of Organic Computing Buchtala and Sick (2007) presents an architecture of so-called Organic Nodes that face classification problems. These nodes cooperate by exchanging functional knowledge, acquired from local observations of the environment using radial basis function neural networks. Richert et al. (2005) investigates a multi agent scenario where no direct knowledge exchanges occurs, rather the agents learn action sequences through imitation. Imitation occurs by means of observing other agents and applying sequences of observed basic behaviors. In both papers, locally acquired knowledge is spread over the components of the OC systems. Such an approach is reasonable if spreading knowledge is more easy than acquiring knowledge on every component on its own and if the components have enough resources to store the all the information.

In the following we investigate the opposite case. We assume the components of the systems to be restricted in memory and therefore in the possible knowledge they can store. In such a case it is reasonable that the individual components store only parts of the available knowledge and cooperate by "asking" each other when facing problems they can not solve on their own. In our approach to organize such a cooperation the knowledge when and who to ask is also learned and not predetermined.

5.2 An Approach to Evolve Cooperating Classification Rules Sets

Classification is an important task for many computing systems and a main field of application for Machine Learning techniques. In the following we study how to solve classification problems on decentralized computing systems consisting of autonomously acting components with communication abilities. The components are assumed to have a restricted memory size. An example for such a system is a sensor network, were the sensors can use wireless communication but typically have a memory that can store only a small amount of information. Binary strings, called classification requests, arrive randomly at the components of the system. This models for example local sensory data at sensor nodes. The goal of the system is to yield correct classifications of these classification requests.

Here we propose to use a rule based approach to solve the classification task. The knowledge of a component is represented as a single set of rules. Every rule consists of a condition and an action part. If an incoming request matches the condition of a rule, the action propagated by this rule is executed. As common in classifier systems, these actions

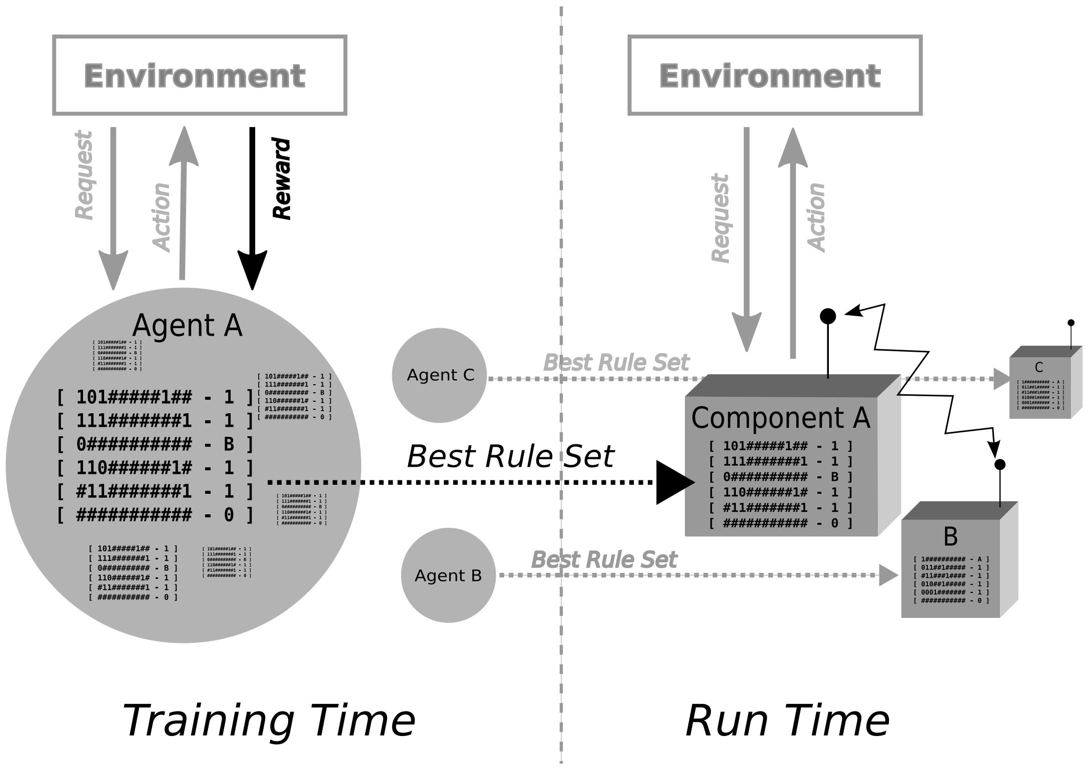


Figure 5.2: The best rule sets evolved in the training phase using coevolving Pittsburgh-style learning classifier systems can be deployed onto the memory constrained components.

represent the demanded classifications. Since only a limited number of rules can be stored on the memory constrained components, only a limited number of correct classifications can be done by a single component. In order to solve more complex classification problems, we allow the components to cooperate. In this way the components can specialize for parts of the classification problem and use the knowledge of other components by delegating classification requests they can not solve on their own. The possible actions of the rules are extended by special actions that can propagate to delegate the matched request to a certain component. In this way a request can be forwarded over several components until a component is found which has a rule to classify the request.

In this chapter we investigate the question how appropriate rule sets for the components, i.e., how specialists for certain parts of the problem and the needed cooperation, can be generated. Figure 5.2 gives a schematic view of our approach. In a so called training phase the components of the system are simulated as agents in a multi-agent system. In the simulation the knowledge of every single agent is represented as a Pittsburgh-style learning classifier system. This means every agent exhibits and evolves an entire population of rule sets. In the training time the LCS systems of the agents learn how to solve the given classification problem by evolving fixed length rule sets that might cooperate by the delegation of requests. For example, in Figure 5.2 agent A has evolved a rule which propagates the action `B`, that is, a rule to delegate the matched requests to agent B. The size of the rule sets is restricted in order that a single rule set fits into the limited memory of the components. After the (offline) evolution in the training phase every agent's best rule set can be deployed on the respective component. The generated rules contain knowledge about correct classifications and knowledge about correct cooperation (which leads to correct classifications). In the following investigations we will mainly focus on the training phase and investigate the structure and properties of the evolved rule sets and the way the components share their knowledge.

To use a Pittsburgh-style classifier system with fixed length rule sets guaranties that the evolved rule sets fit into the component's memory and also have a high strength in terms of classification performance. The drawback is, that because Pittsburgh-style classifier systems need much memory the approach only works offline in a simulation of the system.

In order to design systems capable of online learning, i.e., the co-evolution of cooperating rule sets directly on the components, it would be necessary to use a classifier systems which can deal with a limited memory. To the best of our knowledge this problem has not yet been addressed in the literature. It is an interesting question if and how LCSs, for example Michigan-style systems, like the accuracy based XCS or the strength based ZCS (Wilson, 1994), can be modified to work with a limited memory and to generate a bounded number of rules which still give a good overall reward. The modifications of XCS presented in Dawson (2003) can reduce the population size needed by the system. But still the number of rules always exceeds the number of rules needed to solve the investigated problems optimally. The question, if these systems can generate the best possible rule sets in case the used population sizes are smaller than the needed number of rules for solving the considered problem optimally, is not answered yet.

The idea behind co-evolving cooperating LCS systems in this form is quite new and the intention of the results presented here is to illustrate the power and usefulness of such systems in principle. Many interesting questions regarding evolving cooperating LCS systems remain which could not be covered in the scope of this thesis, but will be addressed in future research. At some points possibly interesting questions, design alternatives, and research directions are mentioned.

In the literature little work about co-evolving Learning Classifier Systems can be found (Bull, 2001). Potter and Jong (2000); Potter et al. (1995) investigates systems of several co-evolving populations of cooperating classifier systems. In contrast to the work presented here, the communication within these systems is limited to an occasional broadcast of representative individuals. In Bull (1999) cooperating Pittsburgh classifier systems with the possibility to communicate are used to evolve the control of a wall climbing robot. As in our work it is not predetermined which agents communicate, instead the communication has to be learned, too. The need for cooperation in this system comes from the fact that the different populations of classifiers are assigned to different functions (legs of the robot) and have to work together to achieve the global goal of movement. The novelty of our investigation is the assumption that the agents have only limited resources (memory). This limits the possible "knowledge" of the agents and leads to a pressure for cooperation.

5.2.1 A Simple Pittsburgh-style Classifier System

The used Learning Classifier System is a Pittsburgh-style LCS and shares many similarities with the GAssist System.

A *rule* consists of a condition and an action part and is denoted by: `[condition` $\rightarrow$ `action]`. The condition part is a string with a fixed length over the alphabet $\{\texttt{0}, \texttt{1}, \texttt{\#}\}$. The action part is a character from the alphabet $\{\texttt{0}, \texttt{1}, \texttt{A}, \texttt{B}, \ldots\}$. An action '`0`' ('`1`') represents a classification into the class 0 (respectively, 1). Note that the to solve the classification problems studied in this work only two classes are required. But if classification problems with more classes are to be solved, the possible actions can easily be extended. Agents are denoted by `A,B,C,...` . The action for sending a request to a specific agent is denoted by the name of the agent. For instance, in a system with three agents the possible actions for the rules of agent B are `'1','0','A'` and `'C'`, where action '`A`' (resp. '`C`') stands for sending the request to agent A (resp. agent C).

A classification request is a string over the alphabet $\{\texttt{0}, \texttt{1}\}$. All conditions and all requests have length $m \geq 1$. A rule matches a request iff for every $i \in [1, m]$ the i^{th} character of the condition equals '`#`' or is the same as the i^{th} character of the request. For example the rule [`#0` $\rightarrow$ `1`] matches the requests '`00`' and '`10`', but not '`01`' and '`11`'.

A *rule set* consists of a fixed length list of rules. The rules of a rule set have a fixed order. Like in GAssist these rule sets work as decision lists (Rivest, 1987), which means a request is compared from the first (top) rule to the last (bottom) rule until it first matches a rule. The last rule, called default rule, has a condition of the form `##...#` and therefore matches all requests. Note, that this rule is not counted into the rule set size.

The LCS uses a near-standard generational Genetic Algorithm (GA), which operates on a set of 300 individuals (rule sets). Within one cycle of the GA first the fitness of all individuals is calculated. Thereafter, a new generation is formed by repeatedly selecting two parents with a high fitness, generating two offspring rule sets from these parents, and inserting these offsprings into the new population. The offspring is derived from the parents by applying a crossover and a mutation operator. Note, that the best individual of a generation is taken directly into the offspring generation (elitism). How these operators are implemented in detail is given in the following.

Fitness Calculation

Every rule set of the population has an associated fitness value that reflects how well this rule sets performs in solving the classification problem. The fitness of a rule set is the mean reward the rule set gets when all possible problem instances are matched. As an example consider the problem to classify all 4-bit numbers which are "$\geq$ 8 or odd" as '1' and all other 4-bit numbers as '0'. A correct classification leads to a reward of 100 and a wrong one it gets zero reward. Consider the following example rule sets:

Rule Set I	Rule Set II
`1###` $\rightarrow$ `1`	`#1##` $\rightarrow$ `1`
`#1#1` $\rightarrow$ `1`	`###1` $\rightarrow$ `1`
`####` $\rightarrow$ `0`	`####` $\rightarrow$ `0`
Fitness 87.5	Fitness 75

Rule Set I classifies all strings which represent numbers ≥ 8 correctly (first rule), but the odd numbers 1 and 3 (strings '`0001`' and '`0011`') are classified wrongly as '0' since they only match the default rule. On the other hand Rule Set II classifies all odd numbers correctly but the strings '`0100`' and '`0110`' (numbers 4 and 6) are classified as '1' and the strings '`1000`' and '`1010`' (8 and 10) are classified as '0', which is wrong in both cases. Rule Set I classifies 14 of 16 instances correctly which gives a fitness of $7/8 * 100 = 87.5$ and Rule Set II is correct on 12 of 16 instances which leads to a fitness of $3/4 * 100 = 75.0$.

As stated before, as a consequence of the constrained memory of the components the system uses fixed length rule sets, i.e., rule sets with a fixed number of rules. If the number of rules is too small for a given problem, the rule sets can not classify all requests correctly and therefore will not have the maximal fitness. For instance, if the rule set size is 1 in the above given example no rule set can get the maximal fitness of 100.

Selection

The LCS uses the so called Tournament Selection (Goldberg and Deb, 1991). This operator holds a tournament between a fixed number of individuals randomly chosen from the population (in this work three individuals are chosen). The individual with the highest fitness from this group is selected as one parent. For the second parent the same selection mechanism is applied again.

Crossover

Like in GAssist the crossover operator is taken from GABIL (De Jong et al., 1993). This operator defines a cut point within the two parent rule sets to combine them to two offspring rule sets. For example a crossover of the two example rule sets with the cut point after the first character of the second rule looks like this:

```
Rule Set I          Rule Set II         Offspring I

1### → 1                                 1### → 1
#              +      ##1 → 1     ⟶      ###1 → 1
                     #### → 0            #### → 0

                                         Offspring II

                     #1## → 1            #1## → 1
 1#1 → 1       +     #            ⟶      #1#1 → 1
#### → 0                                 #### → 0
```

Note, that offspring I classifies all instances of the toy problem correctly and has the maximal fitness of 100.

The crossover operator is applied with a probability $\rho = 0.6$ on the two selected parents. If it is not applied the parents form the offspring directly.

Mutation

The mutation operator flips a randomly chosen position inside the rule set. If this position is in the condition part of a rule the corresponding character is altered into one of the remaining two possibilities with equal probability. To mutate the action part of a rule the new action is chosen uniformly from all possible actions (without the actual action). The mutation operator is applied with an individual wise probability of $\mu = 0.6$ on the offspring before inserting them into the new population.

5.2.2 Training Phase

Like stated before, the evolution of appropriate rule sets which solve a desired classification problem takes place within the training phase of the system. In this phase the memory constrained components are simulated in a multi agent scenario. Every agent, representing a specific component, exhibits a Learning Classifier System of the kind described in the previous subsection. Thus, an agents encapsulates a set of rule sets that co-evolve with the rule sets of other agents.

Within one GA step for every agent the fitness of all its rule sets is calculated. For this purpose every rule set of an agent is matched against every possible request of the considered problem exactly once. How the possible requests look like depends on the specific classification problem. This is computationally expensive, since a problem size of m can lead to up to 2^m different request strings. On the other hand this ensures the exact calculated fitness for this study and eliminates a possible influence of a random request arrival (which would be a more realistic scenario) on the results. Facing more complex problems, i.e., problems with a large number of possible request strings, it can be reasonable to use more elaborate ways for the training, for example a windowing mechanism like ILAS (Bacardit and Garrell, 2003).

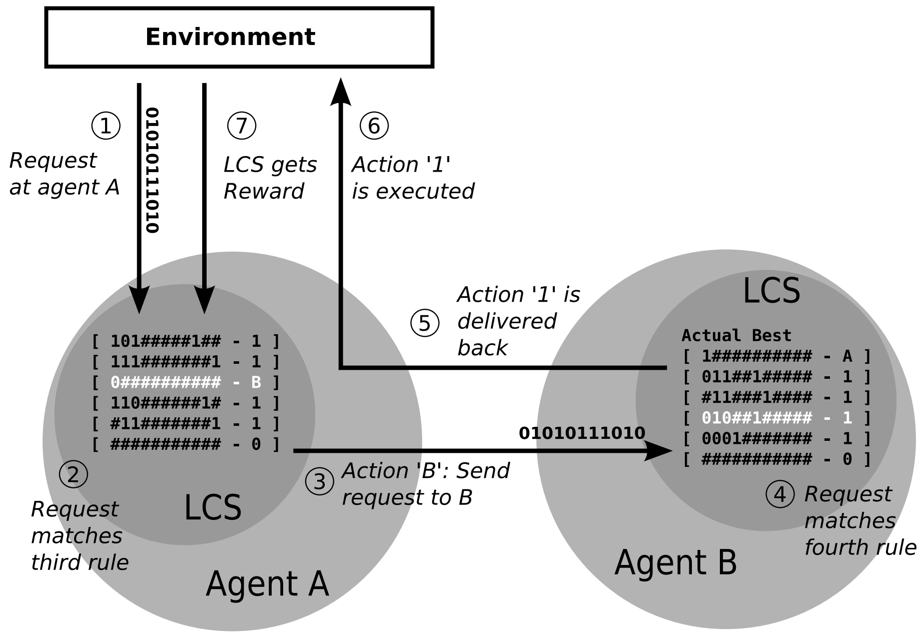


Figure 5.3: Schematic view of the classification of a request

An example classification process of a request during the fitness calculation is given in Figure 5.3. Note that the LCSs of both agents hold and work on whole populations of rule sets, but only one rule set of every agent is depicted in the figure. For agent A the depicted rule set is the one for which the fitness is to be calculated. For agent B the rule set which has actually the highest fitness is depicted. This rule set is also called *actual best* rule set. Agents always use their actual best rule set to answer requests from other agents.

In the figure first a binary request arrives at agent A (1). This request is matched by the third rule of the actually considered rule set of agent A (2). Like stated before a specific request is compared against every rule of the rule set from top to bottom. The action of the first matched rule is executed. Execution means that either the request is classified or the request is send to another agent specified by the action. In the example the propagated action 'B' refers to send the request to agent B (3). In the example the request is matched against the actual best rule set of agent B (4). If a matched rule propagates an action which is a classification (either the class '0' or '1') this result is delivered back. In the example the classification result '1' is delivered back to agent A (5). Agent A sends the result to the environment (6). At the end the LCS of the agent which gets the request from the environment is rewarded. In the example this is the LCS of agent A (7). Note, when the evolved rule sets are deployed and work on the real components, the classification process works exactly the same except that there is no reward. To avoid unlimited request delegation, agents which get a request twice classify this request as a dummy class which gives reward zero in every case.

Recall, the fitness of a rule set is defined as the mean reward generated classifying all possible requests within one GA step. It is irrelevant for the reward if the classification is done directly or through the interaction with other agents. From the point of view of the LCS there is no difference between actions for classifications and actions for delegating requests to other agents. The LCS matches a problem instance (a request), propagates an action and gets a reward associated with this action. This is an important point, because this is the usual way a LCS works and no modification is needed at this point. The knowledge about rewarding delegations is learned in the same way and stored in the same rules as the knowledge about the rewarding classifications.

After the fitness of all rule sets of all agents is calculated the genetic algorithm of every agent is invoked to generate new populations of rule sets. Since the fitness of rule sets is connected to the fitness of other rule sets within other agents, the genetic algorithm co-evolves populations of cooperating rule sets.

When the training phase is finished every agent has evolved an actual best rule set. To calculate the fitness of the whole system a situation is considered as it would be if the evolved rule sets are applied to the components (for example the sensors of a sensor network). In this case only the best rule sets of the agents are transferred onto the according components. It is assumed that in average every possible request arrives at every component. Therefore the fitness of the whole system is the mean of the reward all components get when classifying all possible requests. This value is called *system fitness*.

5.3 Experiments

If not stated otherwise for given parameter sets the systems are trained 1000 steps (fitness evaluations and invocations of the GA). All results are made of 10 independent runs.

5.3.1 The Incremental Multiplexer Problem

The m-multiplexer is a boolean function defined for strings of length $m = n + 2^n$. The first n bits are used to encode an address in the remaining 2^n bits and the value of the function is the value of this addressed bit. For example the 6 multiplexer has two address bits and $2^2 = 4$ data bits. Here the value of `101011` is 1 since the first two bits 10 represent the index 2 (in base ten) which refers to the third bit of the last 4 bits (index 0 refers to the first bit). Given a binary string to the learning classifier system it will response with a classification of the string, which is a value of 0 or 1. This response will lead to a high reward of 1000 if it is the multiplexer function of the input string and lead to a low reward 0 otherwise. Multiplexer problems are commonly used problems in learning classifier system research. These problems are considered to be interesting because the function to be learned is irregular but does allow for generalizations to be made. Generalization means the introduction of the `#` symbol into the rules at positions which does not contribute to the solution. For example, a perfect minimal solution rule set of the multiplexer problem of size 6 may look like this:

```
001### → 1
01#1## → 1
10##1# → 1
11###1 → 1
###### → 0
```

The rule `10##1#` $\rightarrow$ `1` matches the previously given example '`101011`' but also it matches for example '`101111`' or '`100010`' and always gives the correct classification '`1`'.

The mostly used sizes of the multiplexer problem are $m = 3, 6, 11$ and 20. In Davis et al. (2002) an extension called incremental multiplexer problem (IMP) of the original multiplexer problem is introduced to allow for more intermediate problem sizes. The m-IMP is the same as the m multiplexer for $m = n + 2^n$. For other L the L-IMP uses as many address bits as the $m + 1$ multiplexer, but only addresses are allowed which code integers smaller or equal to $m - n - 1$. For instance, the string 10010101 can be found in the 8-IMP, since the address $100 = 4$ refers to the last bit of the string, on the other hand the string 10110101 will not be found.

m k	6	9	11
1	625.00	583.33	562.50
2	750.00	666.67	625.00
3	875.00	750.00	687.50
4	1000.00	833.33	750.00
5	1000.00	916.67	812.50
6	1000.00	1000.00	875.00
7	1000.00	1000.00	937.50
8	1000.00	1000.00	1000.00

Table 5.1: Highest possible fitness a single agent can earn with a rule set of size k at the m-IMP

2 rules on 6-IMP	4 rules on 9-IMP
`11###1` $\rightarrow$ `1`	`#01#1###1` $\rightarrow$ `1`
`#0##1#` $\rightarrow$ `1`	`0001#####` $\rightarrow$ `1`
`######` $\rightarrow$ `0`	`#11###1##` $\rightarrow$ `1`
	`#10##1###` $\rightarrow$ `1`
fitness: 750.0	fitness: 833.33

Table 5.2: Examples of rules sets generated by systems with one agent; both rule sets have the highest possible fitness for the given number of rules

The highest possible fitness a single agent can have with fixed length rule sets of size k at the m-IMP (incremental multiplexer problem of size m) is given in Table 5.1 (Note again, the number given as the size of the rule sets is without counting the default rule).

In the following some typical examples of evolved rule sets are given and discussed. In Table 5.2 two rule sets generated by a system with one agent are shown. The left rule set was generated on the 6-IMP problem and the possible number of rules the agent was allowed to use is 2. The resulting rule set has a fitness of 750.0. Note, the rule [`11###1` $\rightarrow$ `1`] can be found in an optimal solution generated by a system without a constrained number of rules, too. All requests matching this rule will be classified correctly. On the other hand the second rule [`#0##1#` $\rightarrow$ `1`] is too general. It classifies requests of the form `10##1#` and `001#1#` correctly but the requests `000#1#` get a wrong classification. Nevertheless an agent with this rule set still have the highest possible fitness. This is because with the more precise rule [`10##1#` $\rightarrow$ `1`] both `001#1#` and `000#1#` would be matched by the default rule which is correct in only one case.

The right rule set given in Table 5.2 was generated by one agent on the 9-IMP using a rule set size of 4. The resulting rule set has the maximal possible fitness obtainable by

Agent A	Agent B
`011###1## → 1`	`##1###### → A`
`001#1#### → 1`	`1#0####1# → 1`
`101#####1 → 1`	`#1###1### → 1`
`##0###### → B`	`00#1##### → 1`
`######### → 0`	`######### → 0`
fitness: 1000	fitness: 1000

Table 5.3: Example of rules sets generated by two agents with a rule set size constrained to 4 on the 9-IMP problem

a single agent (833.33). In Table 5.3 examples of evolved rule sets for a system with two agents solving the same problem with the same restriction (rule set size 4) are given. It can be seen that both agents specialize to a part of the problem. Agent I classifies all request with character 1 at the third position and sends requests with a 0 at this position to Agent II and vice versa. In the end both agents can classify all possible requests and this leads to the maximal fitness of 1000. Using two rules for sharing the problem, the two agents use in sum 8 rules to act as good as one agent with 6 rules could (6 rules are needed to solve the 9-IMP correctly).

Two examples of evolved rule sets for a system with two agents solving 9-IMP and a rule set size of 3 are given in Table 5.4. On the top the two agents can both get a fitness of 833.3 which is the same as one agent with 4 rules can get (compare Table 5.1). Again there is a “cooperation” rule in both agents which divides the problem into two parts. The example on the bottom of Table 5.4 shows that it is also possible, that the system evolves and gets stuck in less good solutions. Agent I evolved a new default rule (a rule matching every request) and sends requests not matched by its first two rules to Agent II. Agent II has no “cooperation” rule and thus Agent I profits, acting as a “parasite” and gets a high fitness. For the system it is nearly impossible to get out of this. Agent II will not evolve any “cooperation” rule because most requests send to Agent I are “reflected” and this leads to zero reward. On the other hand, for Agent I there is no need for cooperation since this would lead to less overall reward for Agent I. The system fitness is 812.5, which is less than the fitness of 833.3 a system with two cooperating agents can reach.

In Table 5.5 three cooperating rule sets evolved on the 11-Multiplexer problem are given. As can be seen it is also possible that an agent evolves more than one delegation rule (Agent A evolved two delegation rules).

In Figure 5.4(a) the system fitness of systems solving the 9-IMP problem with 1, 2, and 3 agents depending on the rule set is given. It is obvious that agents with a rule set size of 1 can not get a higher fitness by cooperation, but already agents with two rules can. It

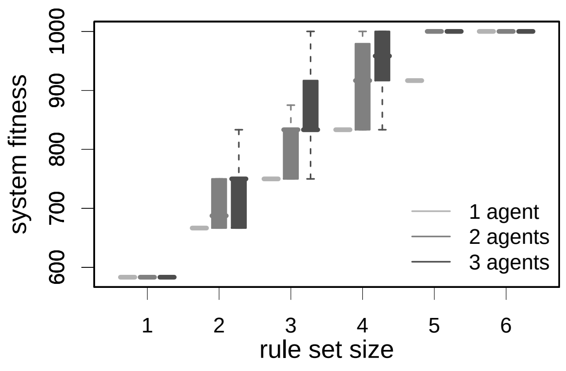
Agent A	Agent B
01####### → B	#10##1### → 1
##1#1###1 → 1	#0####### → A
##0####1# → 1	011###1## → 1
######### → 0	######### → 0
fitness: 833.3	fitness: 833.3

Agent A	Agent B
1#1#####1 → 1	10#####1# → 1
011###1## → 1	00##1#### → 1
######### → B	0#0##1### → 1
######### → 0	######### → 0
fitness: 875.0	fitness: 750.0

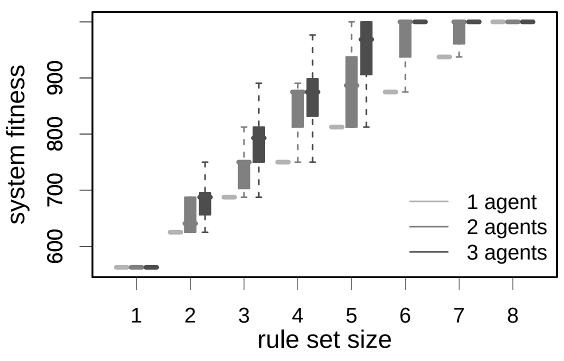
Table 5.4: Two Examples of rules sets generated by two agents with a rule set size constrained to 3 on the 9-IMP problem

Agent A	Agent B	Agent C
010##1##### → 1	110######1# → 1	001#1###### → 1
##0######## → B	##1######## → C	##0######## → B
0#1######## → C	10#####1### → 1	101#####1## → 1
#0######1## → 1	0001####### → 1	#1####1#### → 1
########### → 0	########### → 0	########### → 0
fitness: 937.5	fitness: 875.0	fitness: 937.5

Table 5.5: Example of evolved cooperating rule sets on the 11-Multiplexer Problem for three agents



(a) 9-IMP



(b) 11-IMP

Figure 5.4: System fitness on the Incremental Multiplexer Problem of sizes 9 and 11 using one, two, and three agents as a function of the rule set size

can be observed that in general systems with more agents can get a higher reward when the restricted rule set size is smaller than the number of rules the problem needs to be solved in the best possible way. But naturally there is a bound when the allowed size of the rule set reaches the number of needed rules cooperation has no effect anymore. For example to solve the 9-IMP problem optimally only 6 rules are needed. Thus, in systems with a rule set size of 6 cooperation can not lead to better results at the 9-IMP. But for the larger problems like the 11-IMP sure cooperation in systems using rule set size 6 can improve the system fitness (see Figure 5.4(b)).

In Figure 5.5 the time step at which the best reached fitness for 1, 2, and 3 agents on the 9-IMP problem is given. It can be seen that the more rules the rule sets have the more time it takes to reach the best value. But at a certain rule set size the time steps needed to reach the highest possible fitness become less again. For example considering only one agent the time the system needs to get to the highest value grows until a rule set size of 8 and sinks thereafter. It has to be noticed that for one agent with 6 rules it takes less time to get to the maximal fitness of 1000 as for an agent with a rule set of size 8.

5.3.2 The Incremental Parity Problem

The incremental parity problem of size m (m-IPP) is defined for binary strings of length m. The correct classification of a string is its parity, that is 0 if the number of '1's in the string is even and 1 otherwise. If a request is classified wrong the system gets 0 reward. For a proper classification the reward is the integer value corresponding to the request string plus one. To classify the request 11110 correctly gives a reward of 31 and thus is more valuable for the system than the correct classification of 000110 which only gives a reward of 7.

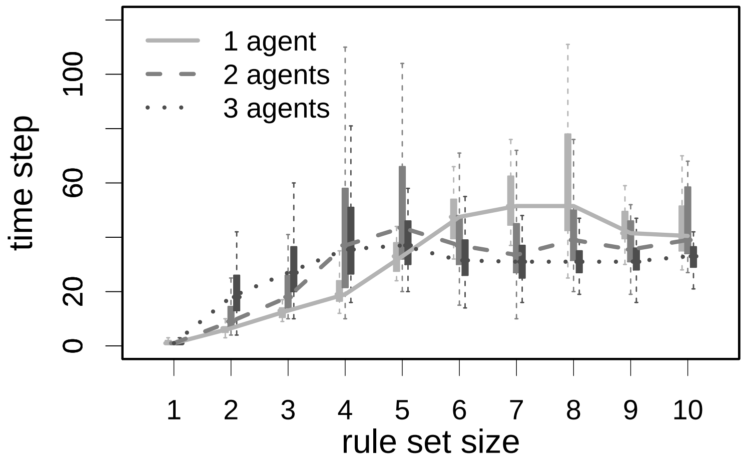


Figure 5.5: Time step at which the best fitness was reached on the 9-IMP problem

The reason for introducing a second test problem is twofold. In the multiplexer problem every correct classified request gives the same reward. In the IPP the different requests lead to different reward values. Thus, it can be tested if the system can concentrate the limited rules on the most valuable requests. The second and more important reason is that in order to solve the IPP optimally many more rules are required than for the IMP. Whereas to solve the m-IMP optimally less than m rules are needed, to solve the m-IPP at least $2^(m-1)$ rules must be used. This is because the IPP does not allow for generalisation, i.e., the use of '#' in the condition part of the rules. Thus the IPP is more practical to use for our experiments when we study how a large group of agents distributes the knowledge.

k	highest fitness	k	highest fitness	k	highest fitness	k	highest fitness
1	9.250	5	12.531	9	14.844	13	16.188
2	10.156	6	13.219	10	15.281	14	16.344
3	11.000	7	13.844	11	15.656	15	16.438
4	11.812	8	14.375	12	15.938	16	16.500

Table 5.6: Highest possible fitness one agent can get on the 5-IPP problem for specific rule set sizes k

The incremental parity problem with strings of length $m = 5$ (5-IPP) is the default problem used in the following experiments. The maximal possible reward an agent can get with a rule set of size k at the 5-IPP problem is given in Table 5.6. Note, to reach the maximal reward of 16.5 at least 16 rules are needed.

In Table 5.7 two example rule sets generated by a system with only one agent are given. The size of the left rule set was restricted to two and the right rule set has a size of 4. The both most valuable rules [`11111` $\rightarrow$ `1`] and [`11100` $\rightarrow$ `1`] can be found in both rule

2 rules on 5-IPP	4 rules on 5-IPP
	`11001` → `1`
`11100` → `1`	`11111` → `1`
`11111` → `1`	`11010` → `1`
`#####` → `0`	`11100` → `1`
	`#####` → `0`
fitness: 10.15625	fitness: 11.8125

Table 5.7: Examples of rules sets generated by systems with one agent; both rule sets have the highest possible fitness for the given number of rules

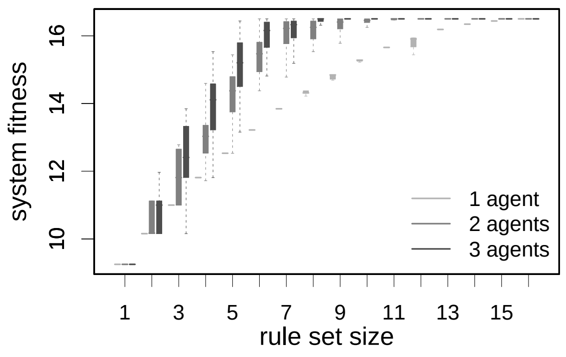
Agent A	Agent B
`1#0##` → `B`	`1#1##` → `A`
`11111` → `1`	`#1#10` → `1`
`0#1##` → `B`	`#1#01` → `1`
`#0#10` → `1`	`#0#11` → `1`
`#1#00` → `1`	`#0#00` → `1`
`#####` → `0`	`#####` → `0`
Fitness: 15.375	Fitness: 14.1875

Table 5.8: Example of rules sets evolved by two agents with rule sets of size five on the 5-IPP problem

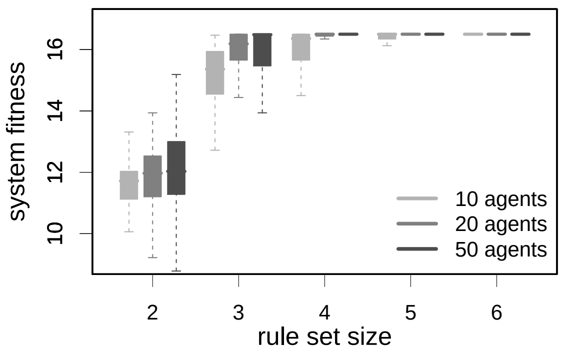
sets. They give a reward of 32 (respectively 29) for the correct classification of '`11111`' (respectively '`11100`'). The rule set with a size of 4 also matches the third and the fourth most valuable requests. These results show that the Pittsburgh-style classifier system evolves rule sets which get the maximal possible reward for a fixed number of rules.

An example of evolved cooperating rule sets generated by two agents on the 5-IPP problem is presented in Table 5.8. The evolved division of the problem is more complex than for example the one given in Table 5.3 where the decision which agent classifies what requests was only determined by one bit. The shown solution has even a higher fitness (14.78125) than one agent with 8 rules could get (14.375). This shows that there is no fixed distinction between "rules for cooperation" and "rules for classification".

In Figure 5.6(a) the system fitness of systems with 1, 2, and 3 agents depending on the allowed rule set size is given. Regarding only the results of systems using only one agent it can be noticed that in most runs these systems are able to evolve rule sets which have the highest possible fitness. This means the used Pittsburgh style classifier system with a fixed rule set size works like intended. It can get the highest possible fitness not only on



(a)



(b)

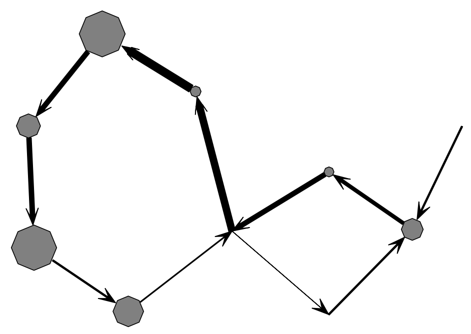
Figure 5.6: System fitness on 5-IPP for one, two, and three agents (a), and 10, 20, and 50 agents (b)

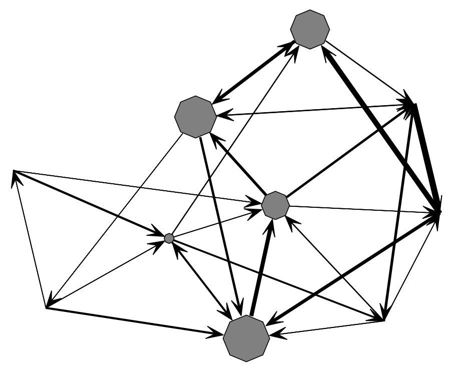
problems like the multiplexer where every classification gives the same reward, but also on problems where there is an unequal reward distribution. Furthermore Figure 5.6(a) shows that the more agents are used, the better the performance of the system is. The same as for the multiplexer problem, also for the incremental parity problem the agents start to cooperate and this eventually leads to a higher system fitness.

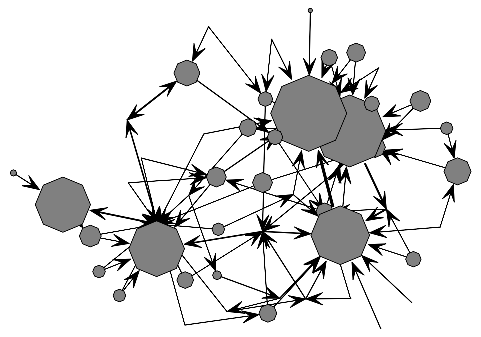
To investigate if this still holds when using a large number of agents, we simulated systems of 10, 20 and 50 agents. The resulting system fitness for different restrictions in rule set sizes on the 5-IPP problem are shown in Figure 5.6(b). Again it can be observed that the more agents in the system, the higher the resulting system fitness. When using a large number of agents the systems can reach the maximal fitness even for small rule set sizes. When restricting the rule set size to 5 most of the time all systems are able to reach the maximal fitness. For systems of 20 or 50 agents this is still the case for rule set size 4. Even with a very small rule set size of 3 the systems are able to reach a quite high fitness, 50 agents actually can reach the maximal fitness in this case sometimes.

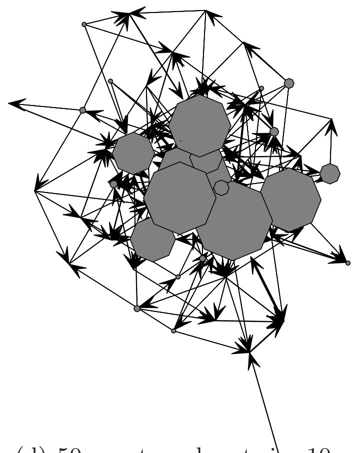
5.4 Evolved Communication Patterns

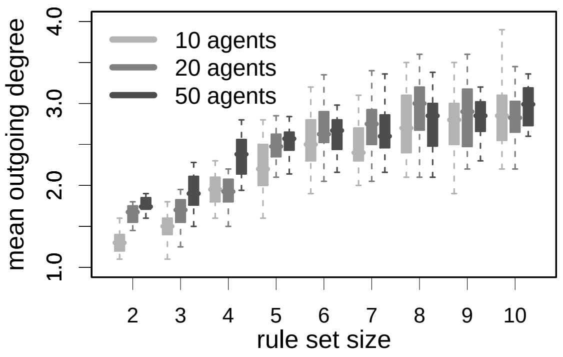
To investigate the evolved communication patterns between the agents, i.e., how they cooperate and distribute requests, the paths of the requests have been recorded and are represented by a graph called *communication graph*. In this graph agents are depicted as circles. The size of the circle corresponds to the number of requests which have been classified by the corresponding agent. Between two agents A and B a directed link is drawn, if A has sent a request to B. The width of a link corresponds to the number of requests which have been send between the agents that are connected by the link.


(a) 10 agents, rule set size 2

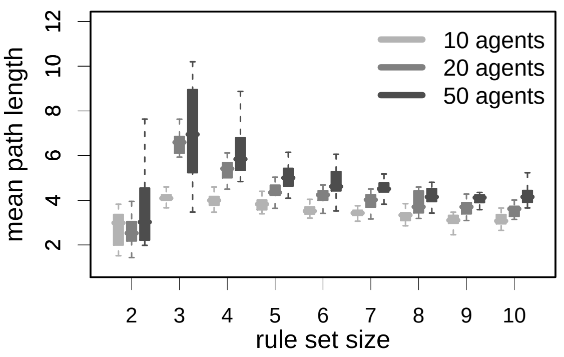

(b) 10 agents, rule set size 10


(c) 50 agents, rule set size 2


(d) 50 agents, rule set size 10

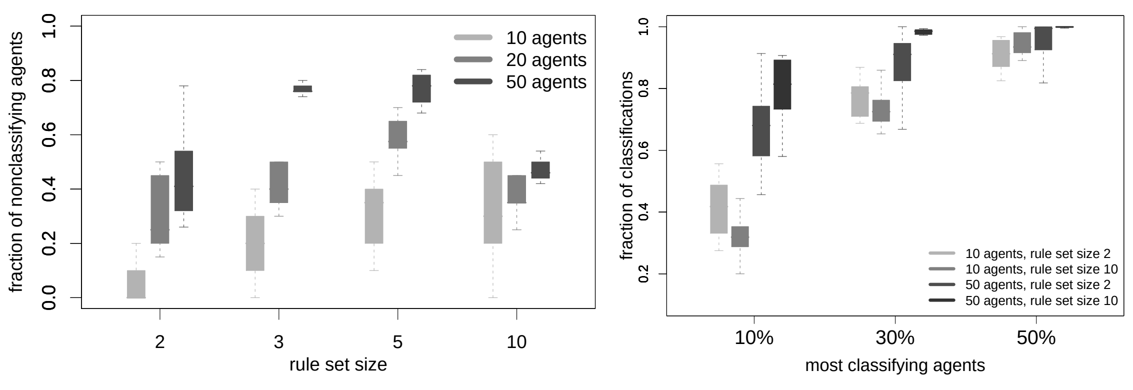


(e) mean outgoing degree



(f) mean path length

Figure 5.7: Examples of evolved communication graphs (a-d); Size of agents relates to the number of classifications done; mean outgoing degree (e) and mean path length (f) for 10, 20, and 50 agents as a function of the rule set sizes



(a) (b)

Figure 5.8: Fractions of non-classifying agents for systems of 10, 20, and 50 agents depending on the rule set size (a); fraction of classifications done by different fractions of agents performing most of the classifications (b)

The number of agents a specific agents sends requests to is called the *outgoing degree* of the agent. The number of agents a specific agents directly gets requests from is called *incoming degree*. Regarding the effort for classification it is interesting how many agents a request passes until it is classified, i.e., how far a request is send through the network. This value is called *path length*. The mean outgoing degrees of the agents in systems of 10, 20, and 50 agents using different rule set sizes are depicted in Figure 5.7(e). The outcome of these experiments regarding the mean path length can be seen in Figure 5.7(f).

In Figure 5.7 typical examples of resulting communication graphs are given. In Figure 5.7(a) an evolved network using 10 agents with rule sets constrained to a size of 2 is given. First it can be noticed that every agent sends requests to other agents. There can be found even two agents which send their requests to two other agents. Since the rule set size is 2 this implies that at least these agents have no “own” knowledge, since they use all their rules for delegating requests. The mean outgoing degree for networks with 10 agents and rule set size 2 is between 1 and 1.5 and in mean a request passes 3 agents until its classification.

When using larger rule sets the number of outgoing links increases. As an example consider Figure 5.7(b), which shows the communication graph of a system with 10 agents and a rule set size restricted to 10. In mean every agent sends requests to three other agents and it takes about 4 hops to classify a request. Since it needs 16 rules to reach the maximal fitness, in principle two agents with rule sets of size 10 could solve the problem. But the evolved systems delegate more requests as needed because there is no force in the system towards less communication.

In systems with 50 agents and a small rule set size of 2 agents can be found which get requests from a high number of agents. For instance in the network given Figure 5.7(c) there is one agent which gets requests from 25 agents. The second most incoming degree is 15, but the mean incoming degree in this system is around 1.8. The mean outgoing degree between 1.5 and 2 shows again that most of the agents do not classify at all. The path length is slightly higher than when using less agents. This suggests that only a few agents have “real knowledge” and the other agents use this knowledge by delegating all their requests. When using a larger rule sets size of 10 the outcome looks a little different (Figure 5.7(d)). The highest incoming degrees are near 10 and the mean incoming degree is around 3.4. The outgoing degree is approximately 3 and the path length a request takes through the system is around 4.

These results signify that the larger the rule set, the less agents requests have to pass until classification. The reason is, the more rules the agents have, the less need for cooperation because the agents can evolve more classification rules for their own. The outgoing degree, i.e., the number of agents an agent sends requests to, is also growing with the size of the rule sets, whereas in Figure 5.7(e) it seems there is a saturation around 3.

The depicted communication graphs suggest an unbalanced distribution of the classifications done by the agents. Some agents classify a lot of requests and others do not classify at all. To investigate this observation in more detail, the fraction of non-classifying agents, i.e., agents which only delegate requests, was measured. The results for 10, 20, and 50 agents are given in Figure 5.8(a). Following observations can be made. The more agents used in the system, the higher the fraction of agents that delegate all their requests to other agents. For instance, consider the case of a rule set size of 5. Using 10 agents leads to about 30% of non-classifying agents, whereas from 50 agents even 80% exhibit no “own” knowledge.

Additionally, the agents were ranked according to the number of classifications they did. The fraction of all classifications that have been done by the highest 10% agents of this ranking (respectively, 30%, 50%) is shown Figure 5.8(b). The plotted results suggest that a small fraction of the agents did most of the classifications of the system. For example the top 10% of agents in terms of number of classifications in systems of 50 agents and rule set size of 10 did about 80% of all classifications.

The unbalanced knowledge distribution can be explained with the fact that it seems to be more easy to evolve rules for delegation than rules for classification. For example, assume agent A already has a high fitness, i.e., it can classify most requests correctly and Agent B has a low fitness. If B evolves a rule that delegates some requests to A this will instantly make B perform better. To evolve such a rule has a high probability because only a mutation of the action part of any of B’s rules, changing the propagated action to ’`A`’, is needed. The more general the new delegation rule, i.e., the more wild cards ’`#`’

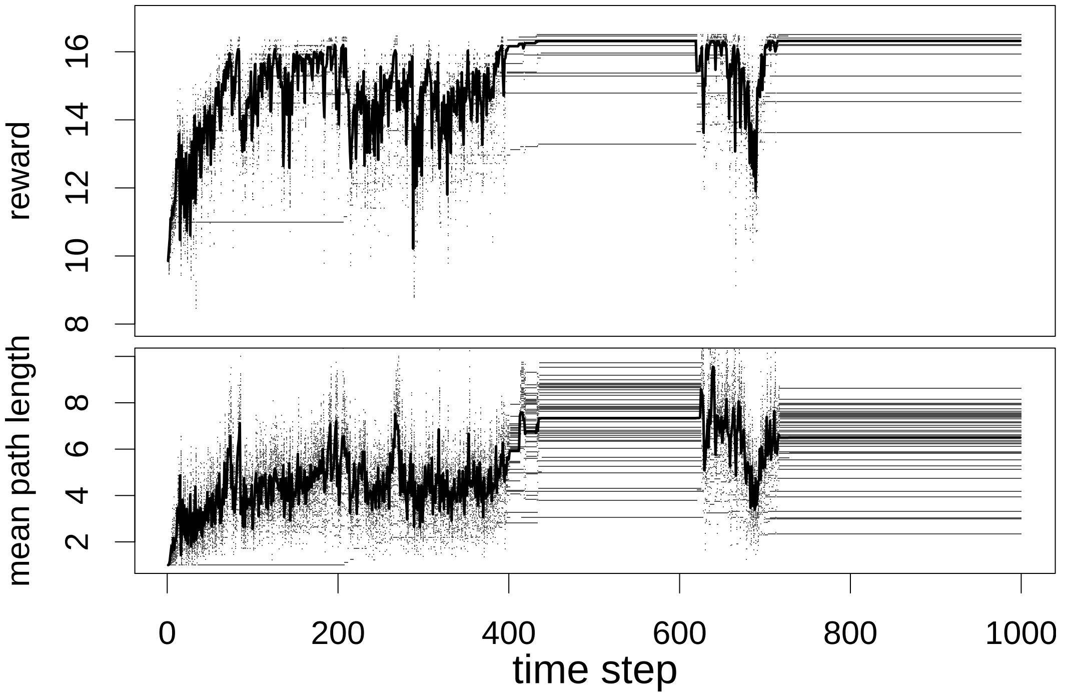


Figure 5.9: A typical run of a system with 50 agents; stable system phases are intercept by phases of reorganization

are in the condition, the more requests are delegated to A and therefore the higher B's fitness. That means that any mutation changing a position in the condition of the rule to '#' will lead to a higher fitness immediately. In sum, compared to the evolution of a correct classification rule, it is much more easy to generate a correct delegation rule. This is because of two reasons. First, as shown, the evolution of a delegation rule has a high probability and B can not do wrong if A has a high fitness. Second, the gain of reward for the delegation rule is much higher than for a correct classification rule since with only one rule B can earn a reward at least as high as A. After B has evolved the new rule other agents with low fitness only need to evolve delegation rules to A or B to get better. This is the reason for the observed communication graphs with the unbalanced distribution.

In Figure 5.9 a typical evolution of the fitness of the agents (top - thick line marks the system fitness) and the mean path length (bottom) of a system with 50 agents over time is given. As can be seen, after around 450 time steps all observed values stay at constant values for about 150 time steps. After this phase, at a point near time step 600, a reorganization of the whole system occurs. Starting from 16.315 the fitness of the system fluctuates for approximately 100 time steps between 11.90 and 16.323 and finally becomes constant again at a value of 16.3193. The average of the mean path lengths decreases from 7.33 to 6.5 in the reorganization phase.

These phases are typical and have been observed in most simulation runs. This behavior can be explained by the fact, that although the actual best rule sets of all agents do not change, there is still an evolution of new rule sets within the population pools of the agents. If one agent evolves a new best rule set this can imply that the fitness of other agents changes. This can start a cascade of changes within the system.

This leads to the interesting question if the system can get into a suboptimal state in terms of system fitness from which it can hardly escape. If 'agent A gets better' implies that 'agent B gets worse' and when B gets better A gets worse, this could lead to an unlimited reorganization of the system. Investigation regarding these questions must be

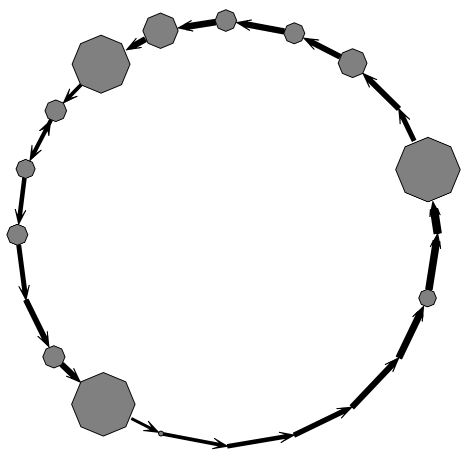
In the system as proposed and investigated here only the agent that gets the request from the environment is rewarded and all other agents involved in the classification of the request are not. Rewarding all involved agents could prevent an unlimited reorganization. But

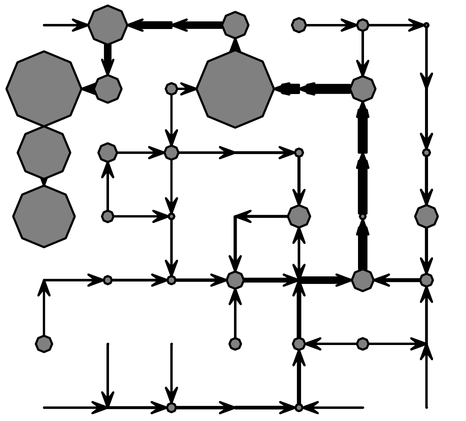
5.4.1 Communication Topologies

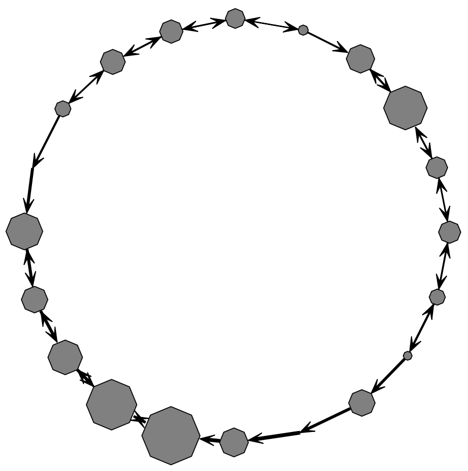
So far we assumed that each agent can send requests to every other agent directly, or in other words the communication network of the agents was assumed to be fully connected. In real Organic Computing systems the components will probably be spatially distributed and an underlying communication topology will be present. In such a topology the components have only a limited number of neighboring components which they can send requests to and receive requests from. For example in sensor networks the transmission range of the wireless communication is limited and sensors can only communicate with their close neighbors directly.

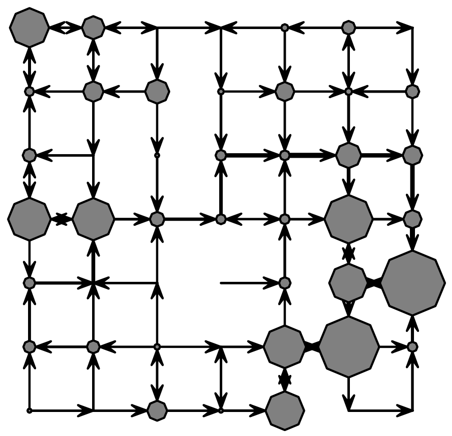
In this subsection we restrict the direct communication possibilities of the agents by introducing a communication topology and investigate its influence. In addition to the case of a fully connected communication network two other topologies, namely the ring and the grid topology, are studied. In the ring topology each agent can send requests to exactly two other agents, forming a single bidirectional pathway for requests - a ring. In the grid topology the agents are arranged in a grid and each agent has up to four neighbors depending on its position in the grid (agents at the corners have two and agents at the borders have only three neighbors).

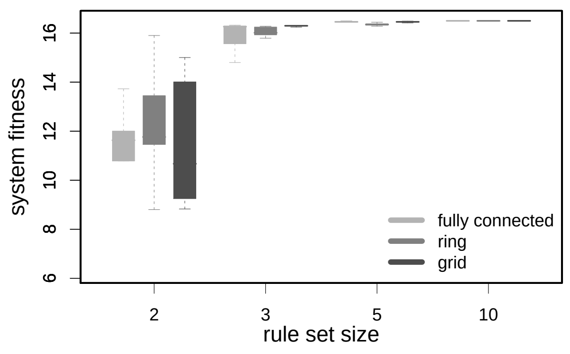
Examples of resulting communication graphs on the ring and grid topology are given in Figure 5.10. Shown are two ring networks build by 20 agents and two grid networks build by 49 agents. The rule set size used to generate the two top (resp. bottom) graphs was 2 (resp. 10). It can be noticed that the same effect as in the fully connected case occurs. Some agents (depicted by very small or no circles) do very little or even no classifications and delegate all their requests to neighboring agents. Also there are some agents which do


(a) ring, 20 agents, rule set size 2

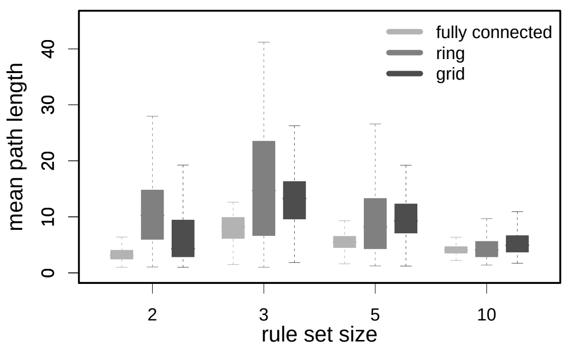

(b) grid, 49 agents, rule set size 2


(c) ring, 20 agents, rule set size 10


(d) grid, 49 agents, rule set size 10



(e)



(f)

Figure 5.10: Evolved communication graphs on ring and grid topologies (a-d); System fitness (e) and mean path length (f) using 50 (resp. 49) fully, ring or grid connected agents as a function of the rule set size

many of classifications for the system (depicted as large circles). Furthermore some highly used connections where lots of requests are passed over (depicted by thicker arrows) can be recognized. Using of a high rule set size (10) in comparison to low size (2) leads to more connections and a little weaker effect of unbalanced knowledge distribution. These observations do not differ much from the case of a fully connected topology.

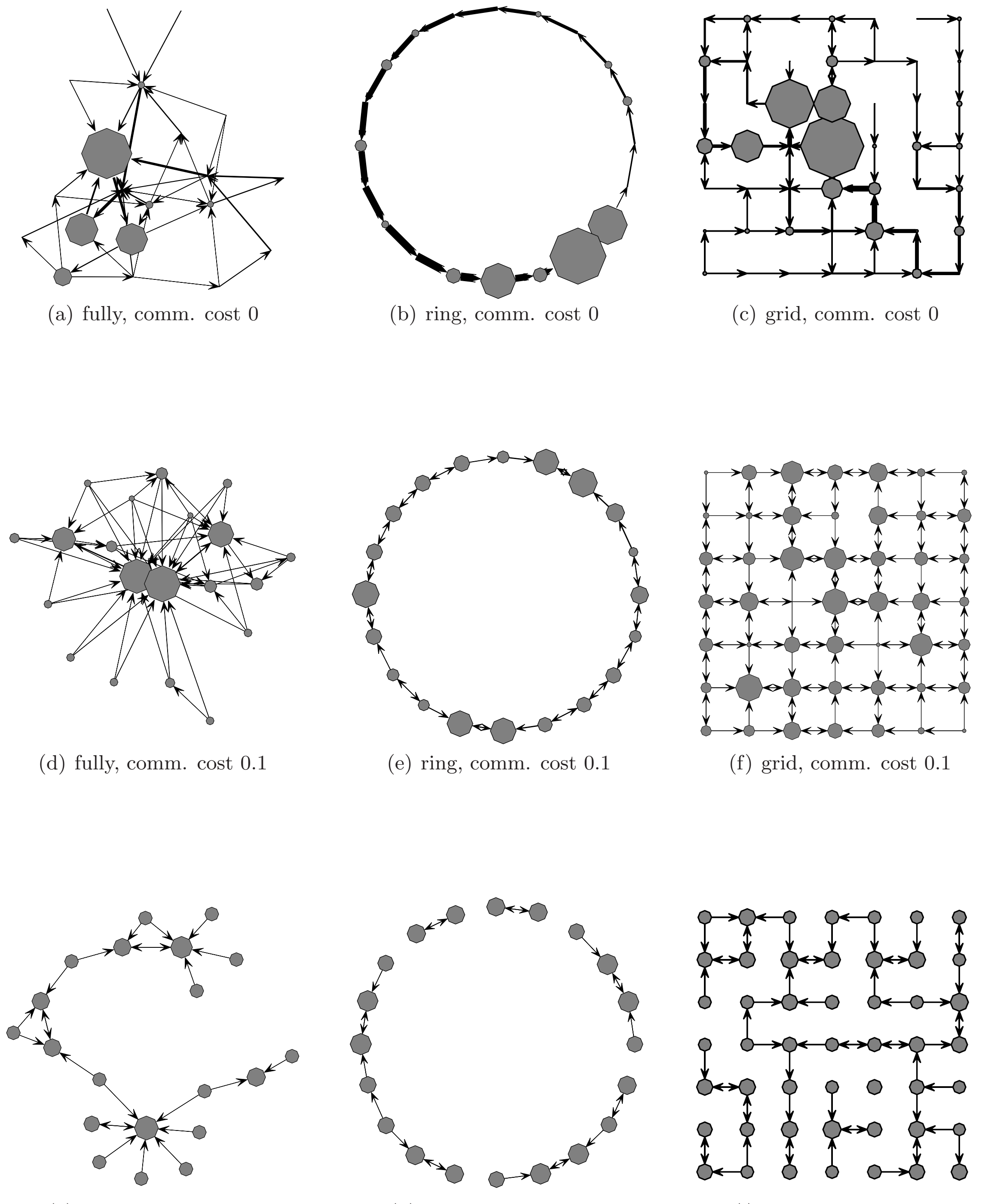
Figure 5.10(e) shows the system fitness, i.e., the mean reward over all agents for all possible requests, of systems using 50 (respectively 49) agents in fully, ring, and grid topology. No big difference between the three topologies can be noticed. This shows that the influence of the communication topologies on the system fitness is only marginal. For rule set size 2 the average fitness was 10.8 (respectively 12.3, 11.3) for the fully connected topology (respectively the ring topology, grid topology). For rule set sizes ≥ 5 no significant difference could be observed. Hence, in this case the different communication topologies do not significantly influence the ability of the systems to reach the maximal fitness.

The topology has a strong influence on the mean length of the paths along which requests are send through the system. Figure 5.10(f) shows the mean path length for systems of 50 fully connected agents, 50 agents in a ring topology, and 49 agents that are connected by a grid topology. It can be seen that the mean path length for the ring topology shows the highest values and the largest variance at small rule set sizes. In the test runs some requests passed 40 or more agents along the ring topology before they were classified. For larger rule set sizes the difference between systems with a ring topology and with a grid topology to systems of fully connected agents becomes smaller but is still observable.

To sum up, it can be stated that the restriction of the agents' communication possibilities through the introduction of a ring or a grid topology does not observably influence the possibility of the systems to reach the maximal fitness, but can increase the mean path length dramatically. This can lead to an increased communication overhead when applying the evolved rule sets on a real system.

5.4.2 Communication Costs

Two properties of the evolved communication patterns are very likely to be unwanted in real systems. First is the fact that the system can evolve unexpected long path lengths, i.e., requests may take long paths through the system until they are classified. This might be a flaw because communication always leads to energy consumption and usually for small components like sensors in sensor networks energy is a crucial resource and the reason why communication must be reduced to a minimum. Second, the distribution of knowledge is very unbalanced, as only a few agents classify most requests and the remaining agents only delegate requests to these agents. This becomes disadvantageous in situations were the system has to be robust with respect to (temporary) failures of the agents.



(a) fully, comm. cost 0

(b) ring, comm. cost 0

(c) grid, comm. cost 0

(d) fully, comm. cost 0.1

(e) ring, comm. cost 0.1

(f) grid, comm. cost 0.1

(g) fully, comm. cost 5

(h) ring, comm. cost 5

(i) grid, comm. cost 5

Figure 5.11: Influence of communication costs on the communication graph; communication cost 0 (a,b,c), 0.1 (d,e,f) and 5 (g,h,i) on the different topologies

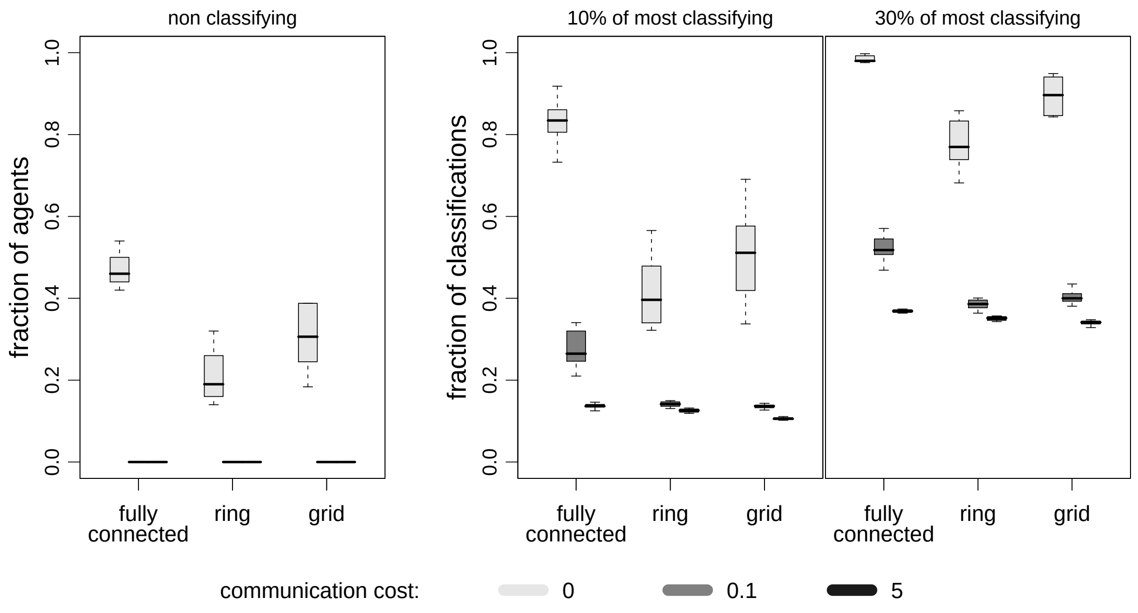
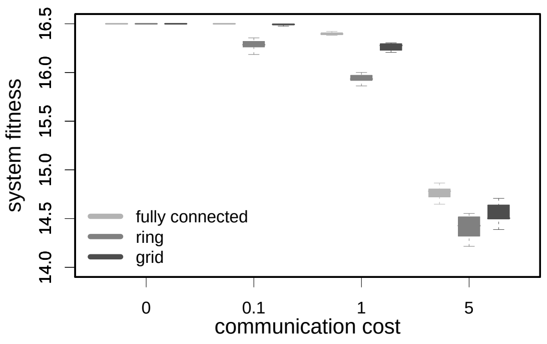


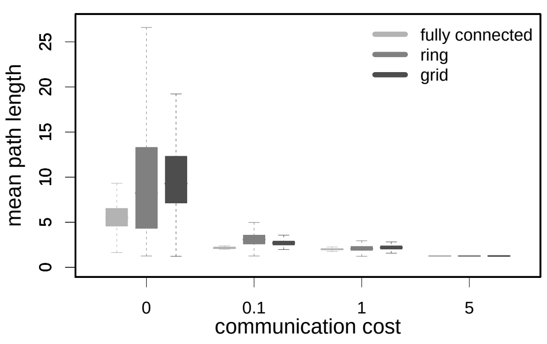
Figure 5.12: Fraction of agents which do not classify any request depending on the communication cost are given (left); fraction of the performed classifications done by the upper 10% (resp. 30%) of the most classifying agents (right)

In the following a force towards less communication is induced in the system through the introduction of communication cost in the training phase. Communication costs are realized by decreasing the reward the system gets for a classification by a fixed amount for every additional communication operation that was used. For instance, the classification of the request `'10000'` in the 5-IPP usually gives a reward of 16. If the request passes three agents until classification and the communication costs are 5 the resulting reward is $16 - 2 * 5 = 6$.

In Figure 5.11 example communication graphs evolved under the influence of different communication costs for the three test topologies are given. Without communication costs (top row) there is an unequal distribution of the number of classifications done by the agents. As already observed "hot spots" evolve, i.e., agents which do most of the classifications of the systems. The introduction of small communication cost of 0.1 has a strong influence on the distribution of knowledge in the system (Figures 5.11(d),(e) and (f)). Note that this cost is 10 times smaller than the reward for the least valuable classification, which is reward 1 for the request '`00000`'. Obviously, there are more agents classifying requests and less non-classifying agents. When using high communication cost of 5 there is overall less communication in the system and there even evolve agents which do not communicate at all.



(a) system fitness



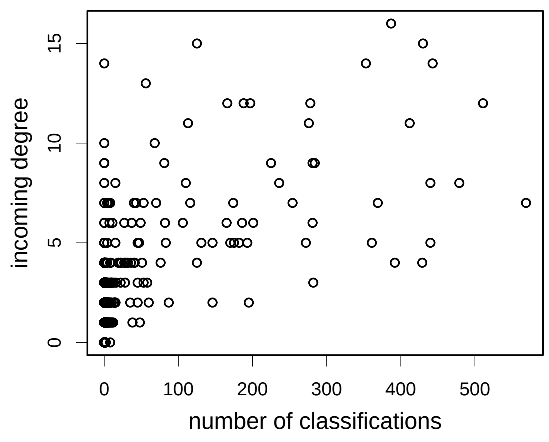
(b) mean path length

Figure 5.13: Influence of communication costs on the system fitness (a) and the mean path length (b) for the different topologies

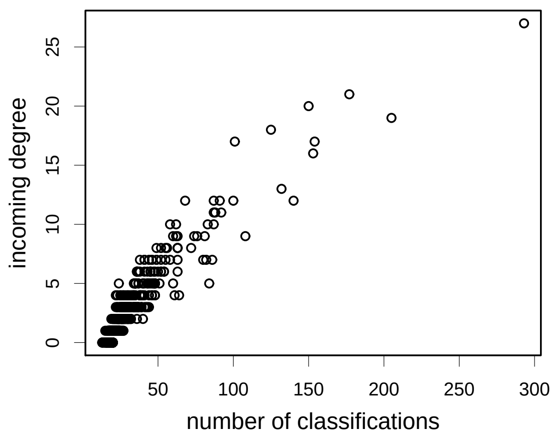
Figure 5.12 quantifies these findings in more detail. The shown results are generated on systems of 50 agents (respectively 49 for the grid topology) and using a rule set size of 5. For the three test topologies the fractions of non-classifying agents are given in the left part of the figure. As can be seen without communication costs roughly 50% of the agents in the fully connected topology (resp. ≈ 20% in the ring and ≈ 30% in the grid topology) do not do a single classification and delegate all received requests to other agents. The introduction of communication cost changes the outcome strongly and leads to systems without non-classifying agents at all, i.e., every agent classifies at least one request it gets.

The right part of Figure 5.12 shows the contribution of the top 10% (resp. 30%) of the agents, ranked dependent on the number of individually performed classifications, to all classifications of the system. Following observations can be made. Using no communication cost leads to the stated unbalanced knowledge distribution, where 10% (resp. 30%) of the agents perform ≈ 50% (resp. 70%) of the classifications of the whole system (in case of the fully connected topology these values are even higher). Introducing communication cost has a strong influence, as the fraction of classifications done by the most classifying agents rapidly decreases. For example, using communication cost 0.1 in the fully connected topology lowers the fraction of classifications done by the best 10% of the agents from ≈ 80% to ≈ 30%. This supports the interpretation that the distribution of classification knowledge over the agents becomes more balanced when using communication cost in the evolution process.

In Section 5.4.1 it is shown that the communication topology has only a slight influence on the system fitness but a strong influence on the mean path length. Its influence on the evolved distribution of knowledge can be seen in Figure 5.12. When no communication cost are applied the fractions of non-classifying agents and classifications done by 10% (resp. 30%) of the most classifying agents are highest if the agents are fully connected and



(a) no communication cost



(b) communication cost 0.1

Figure 5.14: Correlation of the incoming degree and the number of classifications without (a) and with (b) communication cost of 0.1; dots represent agents in fully connected systems of 50 agents and rule set size 10

lowest for the ring topology. That is, the knowledge is more evenly distributed in the grid and in the ring compared to the fully connected topology.

Under the influence of increasing communication costs the mean path length decreases rapidly as shown in Figure 5.13(b). For instance, when communication is for free systems of 50 agents connected as a ring sometimes even evolve mean path lengths larger than 20 hops. On the other hand, even with low communication costs the mean path lengths in the experiments were not higher than 5. High communication costs lead to mean path lengths of 1, what means that there is hardly any communication.

The influence of the communication costs on the system fitness can be seen in Figure 5.13(a). High communication costs reduce the system fitness significantly. But with small communication costs systems that are fully connected or connected as a grid can get a maximal fitness in most of the test runs. Only the agents connected in a ring topology suffer slightly from communication cost 0.1. This is because at a certain point of the evolution of the system the path lengths in the ring are long for some requests. If the communication costs for the classification of these request would be larger than the expected reward, rules delegating these requests are not evolved.

Figure 5.14 gives the correlation between the incoming degree of an agent, i.e., the number of agents it gets requests from, and the number of classifications done by the agent. Compared to Figure 5.14(a) where no communication costs are applied Figure 5.14(b) shows a strong correlation between the number of classifications done and the incoming degree of the agents. This means that in systems evolved under the influence of communication costs agents that get requests from many other agents also have much classification knowledge.

5.5 Summary

In this chapter we studied a first approach to generate suitable rule sets for solving classification problems on systems of autonomous, memory constrained components. It was shown that a multi agent system that uses interacting Pittsburgh-style classifier systems can evolve appropiate rule sets. The system evolves specialists for parts of the classification problem and cooperation between them. In this way the components overcome their restricted memory size and are able to solve the entire problem. It was shown that the communication topology between the components strongly influences the average number of components that a request has to pass until it is classified. It was also shown that the introduction of communication costs into the fitness function leads to a more even distribution of knowledge between the components and reduces the communication overhead without influencing the classification performance very much.

If the system is used to generate rule sets to solve classification tasks on real hardware systems, communication cost in the training phase can thus lead to a better knowledge distribution and small communication cost. That is, in this way the system will be more robust against the loss of single components and longer reliable in case of limited energy resources.

6 Conclusions

Organic Computing is a new field of computer science that has the vision to make future technical systems more life-like in order to address the challenging requirements raised by their increasing complexity. Complex living systems are often self-organizing and show emergent behavior that makes them robust, adaptive, and reliable. The aim of Organic Computing is to identify and adapt the underlying principles of self-organization and emergence found in natural systems, in order to design technical systems that exhibit the same properties and in order to be able to develop methods to control the resulting technical systems. This thesis covered all these aspects: We utilized emergent effects for solving sorting and clustering problems in a decentralized and robust way. We investigated how to control emergent effects by using control swarms or by modifying the environment of the system. We applied self-organization principles found in social insects to the task allocation in Organic Computing Systems. And we used evolutionary methods to evolve systems of specialized cooperating components.

First we investigated two typical self-organizing systems that exploit emergent effects. Emergent Sorting Networks sort random sequences of objects of different types while these objects are traversing the networks. We proposed an algorithm called DPClust for decentralized clustering of packets in networks that is executed by the routers. The purpose of both systems, the Emergent Sorting Networks and the Decentralized Clustering, is to create order, that is, to sort objects and to cluster data vectors. As this is achieved in an emergent fashion, i.e., through the interaction of the systems' entities these systems outline the utilization of emergence in a technical context. As shown the systems show typical properties of self-organizing system: they scale well with the number of entities (agents, packets, routers) and are robust and adaptive.

An important question when dealing with self-organizing systems is how to control (unwanted) emergent effects. A new approach called *swarm controlled emergence* was introduced to deal with this problem. This approach uses a swarm of control components introduced in addition to the normal components of a system. A proof of concept was given by successfully applying the approach to control the emergent clustering effect in a well known model of the clustering behavior of ants. An interesting observation was made when a medium number of a certain type of control agents was introduced into the system. In this case an increased strength of the clustering effect was observed. This shows that it is not a trivial task to design control swarms. A second investigated form of unwanted emergence can occur in form of congestion effects in systems of ant like moving agents. In order to control these effects we modified the environment and have shown experimentally that this can significantly reduce the congestion. Both methods for controlling emergent effects, i.e., the introduction of control components and the modification of the environment do not need to modify the controlled system, i.e., they can be applied to technical systems that are already in use.

In Organic Computing Systems consisting of a large number of active components there will be the necessity for support and system care. We introduced so called *Organic Support Systems*, a distributed self-organizing system of so called helper components to take care of these accruing tasks. The helpers exhibit reconfigurable hardware in order to be able to adapt to the actual needs of the supported systems by specializing for the required types of support tasks. Several aspects were treated. First, we applied models of task allocation in social insects to implement a self-organizing, adaptive, and scalable mechanism for the allocation of the support tasks to the helper components. Second, we used a model of task partitioning in ants to successfully design a system that can allocate the helper components to the different subtasks of the service tasks is a self-organized and robust way and showed that the system can deal well with a changing environment. Third, for the case the supported system components and the helper components are connected via a network we proposed to use a task allocation method based on the DPClust algorithm. This part of the thesis has shown that self-organization methods for task allocation found in social insects can be successfully applied to technical applications resulting in robust, adaptive, and scalable systems.

In the last part of the thesis we studied an approach to generate suitable rule sets for solving classification problems in Organic Computing systems consisting of autonomous, memory constrained components. It was shown that a multi agent system that uses interacting Pittsburgh-style classifier systems can evolve appropriate rule sets. The system evolves specialists for parts of the classification problem and cooperation between them. In this way the components overcome their restricted memory size and are able to solve the entire problem. It was shown that the communication topology between the components strongly influences the average number of components that a request has to pass until it gets classified. It was also shown that the introduction of communication costs into the fitness function leads to a better distribution of knowledge between the components and reduces the communication overhead without influencing the classification performance very much. This is an important finding if the system is to be applied on a real hardware system. This part of the thesis has shown evolutionary processes can create self-organizing systems of cooperating components that evolve interesting communication patterns on system level.

The thesis has shown that self-organization principles from nature can help to design scalable technical systems that consist of a large number of autonomous interacting components. Utilizing emergent effects such system are robust against the loss of single components and adaptive to changing environments. Moreover the thesis has shown that methods can be developed that enable to control emergent behavior in such self-organizing systems.

Bibliography

Abraham A and Ramos V (2003). Web Usage Mining Using Artificial Ant Colony Clustering and Genetic Programming. In *Proceedings of the Congress on Evolutionary Computation (CEC03)*, 1384–1391. IEEE Press.

Agassounon W and Martinoli A (2002). Efficiency and Robustness of Threshold-Based Distributed Allocation Algorithms in Multi-Agent Systems. In *Proceedings of the First Internatial Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-02)*, 1090–1097. ACM Press.

Alpaydin E (2004). *Introduction to Machine Learning*. MIT Press.

Anderson C (2002). Self-organization in relation to several similar concepts: Are the boundaries to self-organization indistinct? In *Biological Bulletin*, (202): 247–255.

Anderson C and Ratnieks FLW (1999a). Task Partitioning in Insect Societies (I): Effect of Colony Size on Queueing Delay and Colony Ergonomic Efficiency. In *American Naturalist*, 154(5): 521–535.

Anderson C and Ratnieks FLW (1999b). Task Partitioning in Insect Societies (II): Use of Queueing Delay Information in Recruitment. In *American Naturalist*, 154(5): 536–548.

Arabie P, Hubert LJ, and de Soete G (1996). *Clustering and Classification*. World Scientific.

Azahar MA, Sunar MS, Daman D, and Bade A (2008). Survey on Real-Time Crowds Simulation. In *Proceedings of the 3rd international conference on Technologies for E-Learning and Digital Entertainment (Edutainment08)*, volume 5093 of *Lecture Notes in Computer Science*, 573–580. Springer.

Azzag H, Monmarché N, Slimane M, Guinot C, and Venturini G (2003). AntTree: A new model for clustering with artificial ants. In *Advances in Artificial Life - Proceedings of the 7th European Conference on Artificial Life (ECAL)*, volume 2801 of *Lecture Notes in Artificial Intelligence*, 564–571. Springer.

Bacardit J (2004). *Pittsburgh Genetics-Based Machine Learning in the Data Mining era: Representations, generalization, and run-time.* Ph.D. thesis, Ramon Llull University, Barcelona, Catalonia, Spain.

Bacardit J and Butz MV (2005). Data Mining in Learning Classifier Systems: Comparing XCS with GAssist. In *Learning Classifier Systems, International Workshops, IWLCS 2003-2005, Revised Selected Papers*, volume 4399 of *Lecture Notes in Artificial Intelligence*, 282–290. Springer.

Bacardit J and Garrell JM (2003). Incremental Learning for Pittsburgh Approach Classifier Systems. In *Proceedings of the Segundo Congreso Espanol de Metaheuristicas, Algoritmos Evolutivos y Bioinspirados*, 303–311.

Bakos JD, Cathey CL, and Michalski A (2006). Predictive Load Balancing for Interconnected FPGAs. In *Proceedings of the 2006 International Conference on Field Programmable Logic and Applications (FPL), Madrid, Spain*, 1–4. IEEE Press.

Balch T (2000). Hierarchical Social Entropy: An Information Theoretic Measure of Robot Team Diversity. In *Autonomous Robots*, 8(3): 209–237.

Banzhaf W (2009). Self-organizing Systems. In RA Meyers (editor), *Encyclopedia of Complexity and Systems Science*, 8040–8050. Springer.

Batouche M, Meshoul S, and Abbassene A (2006). On Solving Edge Detection by Emergence. In *Advances in Applied Artificial Intelligence, Proceedings of the 19th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE 2006)*, volume 4031 of *Lecture Notes in Computer Science*, 800–808. Springer.

Bedau MA (1997). Weak emergence. In *Philosophical Perspectives*, 11: 375–399.

Bedau MA and Humphreys P (editors) (2008). *Emergence: Contemporary readings in philosophy and science.* MIT Press.

Beringer J and Hüllermeier E (2006). Online clustering of parallel data streams. In *Data Knowl. Eng.*, 58(2): 180–204.

Bernauer A, Fritz D, and Rosenstiel W (2008). Evaluation of the Learning Classifier System XCS for SoC run-time control. In *GI Jahrestagung (2)*, volume 134 of *Lecture Notes in Informatics*, 761–768. Springer, Gesellschaft für Informatik.

Bernon C, Gleizes MP, Peyruqueou S, and Picard G (2002). ADELFE: A Methodology for Adaptive Multi-agent Systems Engineering. In *Engineering Societies in the*

Agents World III, Third International Workshop (ESAW 2002), Revised Papers, volume 2577 of *Lecture Notes in Computer Science*, 156–169. Springer.

Beshers SN and Fewell JH (2001). Models of division of labor in social insects. In *Annual Review of Entomology*, 46: 413–440.

Bonabeau E, Dorigo M, and Theraulaz G (1999). *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press.

Bonabeau E, Theraulaz G, and Deneubourg JL (1996). Quantitative study of the fixed threshold model for the regulation of division of labour in insect societies. In *Proceedings of the Royal Society London B*, 263: 1565–1569.

Bonabeau E, Theraulaz G, and Deneubourg JL (1998). Fixed response thresholds and the regulation of division of labor in insect societies. In *Bulletin of Mathematical Biology*, (60): 753–807.

Branke J, Mnif M, Müller-Schloer C, Prothmann H, Richter U, Rochner F, and Schmeck H (2006). Organic Computing - Addressing Complexity by Controlled Self-Organization. In *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 06)*, 185–191. IEEE Press.

Branke J and Schmeck H (2008). Evolutionary Design of Emergent Behavior. In RP Würtz (editor), *Organic Computing*, 123–140. Springer.

Brown AB and Hellerstein JL (2005). Reducing the cost of IT operations: is automation always the answer? In *Proceedings of the 10th conference on Hot Topics in Operating Systems (HOTOS'05)*, 12–12. USENIX Association.

Brueckner SA (2000). *Return From the Ant—Synthetic Ecosystems for Manufacturing Control*. Ph.D. thesis, Humbold University, Berlin.

Brun Y, Serugendo GDM, Gacek C, Giese HM, Kienle H, Litoiu M, Müller HA, PezzÃĺ M, and Shaw M (2009). *Engineering Self-Adaptive Systems through Feedback Loops*, volume 5525 of *Lecture Notes in Computer Science*, 48–70. Springer.

Brutschy A (2007). *Selbstorganisierte Rekonfiuration von Computingsystemen mittels naturinspirierter Entscheidungsverfahren*. Master's thesis, Universität Leipzig.

Brutschy A, Scheidler A, Merkle D, and Middendorf M (2008). Learning from House-Hunting Ants: Collective Decision-Making in Organic Computing Systems. In *Proceedings of the Sixth International Conference on Ant Colony Optimization and*

Swarm Intelligence (ANTS08), volume 5217 of *Lecture Notes in Computer Science*, 96–107. Springer.

Buchtala O and Sick B (2007). Functional Knowledge Exchange Within an Intelligent Distributed System. In *Proceedings of the 20th InternationalConference on Architecture of Computing Systems (ARCS2007)*, volume 4415 of *Lecture Notes in Computer Science*, 126–141. Springer.

Buck J (1988). Synchronous Rhythmic Flashing of Fireflies. II. In *The Quarterly Review of Biology*, 63(3): 265–289.

Bull L (1999). On Evolving Social Systems: Communication, Speciation and Symbiogenesis. In *Computational & Mathematical Organization Theory*, 5(3): 281–302.

Bull L (2001). On Coevolutionary Genetic Algorithms. In *Soft Computing*, 5(3): 201–207.

Cakar E, Hähner J, and Müller-Schloer C (2008). Creating collaboration patterns in multi-agent systems with generic observer/controller architectures. In *Autonomics '08: Proceedings of the 2nd International Conference on Autonomic Computing and Communication Systems*, 6. ICST.

Cakar E, Mnif M, Müller-Schloer C, Richter U, and Schmeck H (2007). Towards a quantitative notion of self-organisation. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC2007)*, 4222–4229. IEEE Press.

Caprari G, Colot A, Siegwart R, Halloy J, and Deneubourg JL (2004). Building Mixed Societies of Animals and Robots. In *IEEE Robotics & Automation Magazine*, 12: 58–65.

Channa A, Rajpoot N, and Rajpoot K (2006). Texture Segmentation using Ant Tree Clustering. In *Proceedings IEEE International Conference on Engineering of Intelligent Systems (ICEIS'2006)*, 1–6. IEEE Press.

Cheng BHC, Giese H, Inverardi P, Magee J, and Lemos RD (editors) (2009). *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*. Springer.

Christensen A, O'Grady R, and Dorigo M (2009). From Fireflies to Fault Tolerant Swarms of Robots. In *IEEE Transactions on Evolutionary Computation*, 13(4): 754–766.

Chu CT, Kim SK, Lin YA, Yu Y, Bradski GR, Ng AY, and Olukotun K (2007). Map-Reduce for Machine Learning on Multicore. In *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems*, 281–288. MIT Press.

Cicirello VA and Smith SF (2003). Distributed Coordination of Resources via Wasp-Like Agents. In *Proceedings of the First NASA GSFC/JPL Workshop on Radical Agent Concepts (WRAC)*, 71–80.

Cicirello VA and Smith SF (2004). Wasp-like agents for distributed factory coordination. In *Journal of Autonomous Agents and Multi-Agent Systems*, 8(3): 237–266.

Compton K and Hauck S (2002). Reconfigurable computing: a survey of systems and software. In *ACM Computing Surveys*, 34(2): 171–210.

Cooke KL and Grossman Z (1982). Discrete Delay, Distributed Delay and Stability Switches. In *Journal of Mathematical Analysis and Applications*, 86: 592–627.

Corning PA (2002). The re-emergence of "emergence": A venerable concept in search of a theory. In *Complexity*, 7(6): 18–30.

Correia L (2006). Self-organised systems: fundamental properties. In *Revista de CiÃłncias da ComputaÃğÃčo*, 1(1): 1–10.

Couzin I, Krause J, Franks NR, and Levin SA (2005). Effective leadership and decision-making in animal groups on the move. In *Nature*, 433: 513–516.

Cox M and Blanchard G (2000). Gaseous templates in ant nests. In *Journal of Theoretic Biology*, 204: 223–238.

Darley V (1994). Emergent Phenomena and Complexity. In *Artificial Life IV. Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, 411–416. MIT Press.

Datta S, Bhaduri K, Giannella C, Wolff R, and Kargupta H (2006a). Distributed Data Mining in Peer-to-Peer Networks. In *Internet Computing*, 10(4): 18–26.

Datta S, Giannella C, and Kargupta H (2006b). K-Means Clustering over a Large, Dynamic Network. In *Proceedings of 2006 SIAM Conference on Data Mining*.

Datta S, Giannella C, and Kargupta H (2009). Approximate Distributed K-Means Clustering over a Peer-to-Peer Network. In *IEEE Transactions on Knowledge and Data Engineering*, 21(10): 1372–1388.

Davis L, Fu C, and Wilson SW (2002). An Incremental Multiplexer Problem and Its Uses in Classifier System Research. In *IWLCS 01: Revised Papers from the 4th International Workshop on Advances in Learning Classifier Systems*, number 2321 in Lecture Notes in Computer Science, 23–31. Springer.

Dawson D (2003). Improving Performance in Size-Constrained Extended Classifier Systems. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003)*, volume 2724 of *Lecture Notes in Computer Science*, 1870–1881. Springer.

Day WH and Edelsbrunner H (1984). Efficient algorithms for agglomerative hierarchical clustering methods. In *Journal of Classification*, 1(1): 7–24.

De Jong KA (2006). *Evolutionary Computation.* MIT Press.

De Jong KA and Spears WM (1991). Learning Concept Classification Rules Using Genetic Algorithms. In *Proceedings of the Twelfth International Conference on Artificial Intelligence (IJCAI91)*, 651–657. Morgan Kaufmann Publishers Inc.

De Jong KA, Spears WM, and Gordon DF (1993). Using Genetic Algorithms for Concept Learning. In *Machine Learning*, 13(2-3): 161–188.

de Oca MAM, Garrido L, and Aguirre JL (2005a). Effects of Inter-agent Communication in Ant-Based Clustering Algorithms: A Case Study on Communication Policies in Swarm Systems. In *Proceedings of the 4th Mexican International Conference on Artificial Intelligence (MICAI2005)*, volume 3789 of *Lecture Notes in Computer Science*, 254–263. Springer.

de Oca MAM, Garrido L, and Aguirre JL (2005b). An hybridization of an ant-based clustering algorithm with growing neural gas networks for classification tasks. In *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC)*, 9–13. ACM Press.

De Wolf T and Holvoet T (2004). Emergence Versus Self-Organisation: Different Concepts but Promising When Combined. In *International Workshop on Engineering Self-Organising Systems (ESOA 2004), Revised and Invited Papers*, volume 3464 of *Lecture Notes in Computer Science*, 1–15. Springer.

De Wolf T and Holvoet T (2006). Design Patterns for Decentralised Coordination in Self-organising Emergent Systems. In *4th International Workshop on Engineering Self-Organising Systems (ESOA 2006), Revised and Invited Papers*, volume 4335 of *Lecture Notes in Computer Science*, 28–49. Springer.

Degueta J, Demazeaua Y, and Magninb L (2006). Elements about the Emergence Issue: A Survey of Emergence Definitions. In *Complexus*, 3: 24–31.

Deneubourg JL, Goss S, Franks N, Sendova-Franks A, Detrain C, and Chrétien L (1990). The dynamics of collective sorting robot-like ants and ant-like robots. In *Proceedings of the first international conference on simulation of adaptive behavior on From animals to animats*, 356–363. MIT Press.

Deutsch A and Dormann S (2005). *Cellular automaton modeling of biological pattern formation. Characterization, applications, and analysis.* Birkhäuser.

Dhillon I and Modha D (2000). A Data-Clustering Algorithm On Distributed Memory Multiprocessors. In *Large-Scale Parallel Data Mining*, volume 1759 of *Lecture Notes in Computer Science*, 245–260.

Di Caro GA (2004). *Ant Colony Optimization and its application to adaptive routing in telecommunication networks.* Ph.D. thesis, UniversitÃl' Libre de Bruxelles (ULB), Brussels, Belgium.

Dorigo M, Maniezzo V, and Colorni A (1996). Ant System: Optimization by a Colony of Cooperating Agents. In *IEEE Transactions on Systems, Man, and CyberneticsÂŰ-Part B*, 26(1): 29ÂŰ–41.

Dorigo M and Stuetzle T (2004). *Ant Colony Optimization.* MIT Press.

Edmonds B (2004). Using the Experimental Method to Produce Reliable Self-Organised Systems. In *Engineering Self-Organising Systems*, volume 3464 of *Lecture Notes in Computer Science*, 84–99. Springer.

Edmonds B and Bryson JJ (2004). The Insufficiency of Formal Design Methods - The Necessity of an Experimental Approach for the Understanding and Control of Complex MAS. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 04)*, 938–945. IEEE Press.

Eisenhardt M, Müller W, and Henrich A (2003). Classifying Documents by Distributed P2P Clustering. In *GI Jahrestagung (2)*, volume 35 of *Lecture Notes in Informatics*, 286–291. GI.

Eugster PT, Guerraoui R, Kermarrec AM, and Massoulie L (2004). Epidemic information dissemination in distributed systems. In *Computer*, 37(5): 60–67.

Everitt BS, Landau S, and Leese M (2001). *Cluster Analysis.* Arnold Publishers.

Fehervari I and Elmenreich W (2009). Evolutionary Methods in Self-organizing System Design. In *Proceedings of the 2009 International Conference on Genetic and Evolutionary Methods (GEM 2009)*, 10–15. CSREA Press.

Ferrandi F, Lanzi PL, Palermo G, Pilato C, Sciuto D, and Tumeo A (2007). An Evolutionary Approach to Area-Time Optimization of FPGA designs. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS2007)*, 145–152. IEEE Press.

Ferreira PR, de Oliveira D, and Bazzan ALC (2005). A Swarm Based Approach to Adapt the Structural Dimension of Agents' Organizations. In *Journal of the Brazilian Computer Society*, (1): 63–73.

Freedman HI and Kuang Y (1991). Stability switches in linear scalar neutral delay equations. In *Funkcialaj Ekvacioj*, 34: 187–209.

Fromm J (2005a). Ten Questions about Emergence. arXiv:nlin/0509049v1.

Fromm J (2005b). Types and Forms of Emergence. arXiv:nlin/0506028v1.

Fromm J (2006). On Engineering and Emergence. arXiv:nlin/0601002v1.

Gaber MM, Zaslavsky A, and Krishnaswamy S (2005). Mining data streams: a review. In *SIGMOD Record*, 34(2): 18–26.

Gan G, Ma C, and Wu J (2007). *Data Clustering: Theory, Algorithms, and Applications*. SIAM.

Gardelli L, Viroli M, and Omicini A (2007). Design Patterns for Self-Organizing Multiagent Systems. In T De Wolf, F Saffre, and R Anthony (editors), *2nd International Workshop on Engineering Emergence in Decentralised Autonomic System (EEDAS) 2007*, 62–71. CMS Press, University of Greenwich, London, UK, ICAC 2007, Jacksonville, Florida, USA.

Gershenson C (2007). *Design and Control of Self-organizing Systems*. Ph.D. thesis, Vrije Universiteit Brussel, Brussels, Belgium.

Giardina I (2008). Collective behavior in animal groups: theoretical models and empirical studies. In *HFSP Journal*, 2(4): 205–219.

Gleizes MP, Camps V, Georgé JP, and Capera D (2007). Engineering Systems Which Generate Emergent Functionalities. In *International Workshop on Engineering Environment-Mediated Multi-Agent Systems (EEMMAS2007), Selected Revised and Invited Papers*, volume 5049 of *Lecture Notes in Computer Science*, 58–75. Springer.

Goldberg DE and Deb K (1991). A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In GJE Rawlins (editor), *Foundations of Genetic Algorithms*, 69–93. Morgan Kaufmann Publishers Inc.

Goldingay H and Mourik Jv (2008). The Influence of Memory in a Threshold Model for Distributed Task Assignment. In *Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO08)*, 117–126. IEEE Computer Society.

Grünbaum D (1998). Schooling as a strategy for taxis in a noisy environment. In *Evolutionary Ecology*, 12: 503–522.

Gu Y and Hall L (2006). Kernel based fuzzy ant clustering with partition validity. In *Proceedings of the IEEE International Conference on Fuzzy Systems*, 263–267.

Gutowitz H (1995). Complexity-Seeking Ants. In *Advances in Artificial Life: Proceedings of the third European Conference on Artificial Life (ECAL95)*, volume 929 of *Lecture Notes in Artificial Intelligence*, 704–720. Springer.

Haboush W and Shrimpton DH (2005). Fixed Response-Threshold Model for Task Allocation in Sensor Networks. In *Proceedings of the London Communications Symposium (LCS2005)*.

Handl J, Knowles J, and Dorigo M (2003a). On the Performance of Ant-Based Clustering. In A Abraham, M Köppen, and K Franke (editors), *Design and application of hybrid intelligent systems*, volume 104 of *Frontiers in Artificial intelligence and Applications*, 204–213. IOS Press.

Handl J, Knowles J, and Dorigo M (2003b). Strategies for the increased robustness of ant-based clustering. In *Postproceedings of the First International Workshop on Engineering Self-Organising Applications (ESOA 2003)*, volume 2977 of *Lecture Notes in Computer Science*, 90–104.

Handl J, Knowles J, and Dorigo M (2006). Ant-Based Clustering and Topographic Mapping. In *Artificial Life*, 12(1): 35–61.

Handl J and Meyer B (2002). Improved Ant-based Clustering and Sorting in a Document Retrieval Interface. In *Proceedings of the 7th International Conference on Parallel Problem Solving from Nature (PPSN VII)*, volume 2439 of *Lecture Notes in Computer Science*, 913–923.

Handl J and Meyer B (2007). Ant-based and swarm-based clustering. In *Swarm Intelligence*, 1(2): 95–113.

Hardin G (1968). The Tragedy of the Commons. In *Science*, 162(3859): 1243–1248.

Hauert S, Winkler L, Zufferey JC, and Floreano D (2008). Ant-based Swarming with Positionless Micro Air Vehicles for Communication Relay. In *Swarm Intelligence*, 2(2-4): 167–188.

Holland JH (1992). *Adaptation in Natural and Artificial Systems*. MIT Press.

Holzer M, Knerr B, and Rupp M (2007). Design Space Exploration for Real-Time Reconfigurable Computing. In *Proceedings of Asilomar Conference on Signals, Systems, and Computers*, 1981–1985. IEEE Press.

Holzer R, de Meer H, and Bettstetter C (2008). On Autonomy and Emergence in Self-Organizing Systems. In KA Hummel and JPG Sterbenz (editors), *IWSOS*, volume 5343 of *Lecture Notes in Computer Science*, 157–169. Springer.

Hua M, Lau MK, Pei J, and Wu K (2009). Continuous K-Means Monitoring with Low Reporting Cost in Sensor Networks. In *IEEE Transactions on Knowledge and Data Engineering*, 21(12): 1679–1691.

IBM Corporation (2006). An architectural blueprint for autonomic computing. White Paper 4th Ed.

Igel C and Sendhoff B (2008). Genesis of Organic Computing Systems: Coupling Evolution and Learning. In RP Würtz (editor), *Organic Computing*, 141–166. Springer.

Jain AK, Murty MN, and Flynn PJ (1999). Data Clustering: a Review. In *ACM Computing Surveys*, 31(3): 264–323.

Janson S, Merkle D, and Middendorf M (2008). A decentralization approach for swarm intelligence algorithms in networks applied to multi swarm PSO. In *International Journal of Intelligent Computing and Cybernetics*, 1(1): 25–45.

Janson S, Middendorf M, and Beekman M (2005). Honeybee Swarms: How do Scouts Guide a Swarm of Uninformed Bees? In *Animal Behaviour*, 70(2): 349–358.

Jelasity M, Montresor A, and Babaoglu O (2009). T-Man: Gossip-based fast overlay topology construction. In *Comput. Netw.*, 53(13): 2321–2339.

Jensen D and Lesser V (2002). Social Pathologies of Adaptive Agents. In MBH Guesgen (editor), *Safe Learning Agents: Papers from the 2002 AAAI Spring Symposium*. AAAI Press.

Johnson S (2001). *Emergence: The Connected Lives of Ants, Brains, Cities, and Software*. Scribner.

Jones C and Mataric M (2003). Adaptive division of labor in large-scale minimalist multi-robot systems. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*, volume 2, 1969–1974.

Kanade PM and Hall LO (2003). Fuzzy Ants as a Clustering Concept. In *Proceedings 22nd International Conference of the North American Fuzzy Information Processing Society (NAFIPS)*, 227–232.

Kargupta H and Sivakumar K (2004). Existential pleasures of distributed data mining. In *Data Mining: Next Generation Challenges and Future Directions*, 1–25.

Kaufman L and Rousseuw PJ (1990). *Finding Groups in Data: An Introduction to ClusterAnalysis*. Wiley, New York.

Kaufmann P and Platzner M (2007). Toward Self-adaptive Embedded Systems: Multi-objective Hardware Evolution. In *Proceedings of the 20th International Conference on Architecture of Computing Systems (ARCS 2007)*, volume 4415 of *Lecture Notes in Computer Science*, 199–208. Springer.

Kephart JO and Chess DM (2003). The Vision of Autonomic Computing. In *Computer*, 36(1): 41–50.

Kindratenko V, Brunner R, and Myers A (2007). Dynamic load-balancing on multi-FPGA systems: a case study. In *Proceedings of the 3rd Annual Reconfigurable Systems Summer Institute (RSSI07)*.

Kittithreerapronchai O and Anderson C (2003). Do ants paint trucks better than chickens? Market versus response thresholds for distributed dynamic scheduling. In *Proceedings of the 2003 IEEE Congress on Evolutionary Computation*.

Klein M, Metzler R, and Bar-Yam Y (2005). Handling emergent resource use oscillations. In *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 35(3): 327–336.

Klein M, Sayama H, Faratin P, and Bar-Yam Y (2003). The Dynamics of Collaborative Design: Insights from Complex Systems and Negotiation Research. In *Concurrent Engineering: R&A*, 11(3): 201–209.

Komann M, Ediger P, Fey D, and Hoffmann R (2009). On the Effectiveness of Evolution Compared to Time-Consuming Full Search of Optimal 6-State Automata.

In *Proceedings of the 12th European Conference on Genetic Programming (EuroGP09)*, Lecture Notes In Computer Science, 280–291. Springer.

Komann M and Fey D (2009). Evaluating the evolvability of emergent agents with different numbers of states. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation (GECCO09)*, 1777–1778. ACM Press.

König A, Lakshmanana SK, and Tawdrossa PM (2006). Towards organic sensing systems - Dynamically reconfigurable mixed-signal electronics for adaptive sensing in organic computing systems. In *Invited and selected papers of the 2nd International Conference on Brain-inspired Information Technology*, volume 1291 of *International Congress Series*, 38–45. Elsevier.

Krieger M, Billeter J, and L K (2000). Ant-like task allocation and recruitment in cooperative robots. In *Nature*, 406: 992–995.

Krieger M and Billeter JB (2000). The call of duty: Selforganised task allocation in a population of up to twelve mobile robots. In *Robotics and Autonomous Systems*, 30: 65–84.

Krüger M, von der Malsburg C, and Würtz RP (2008). Self-organized Evaluation of Dynamic Hand Gestures for Sign Language Recognition. In RP Würtz (editor), *Organic Computing*, 321–342. Springer.

Labella T, Dorigo M, and Deneubourg JL (2006). Division of labour in a group of robots inspired by ants' foraging behaviour. In *ACM Transactions on Autonomous and Adaptive Systems*, 1: 4–25.

Labroche N, Monmarché N, and Venturini G (2002). A new clustering algorithm based on the chemical recognition system of ants. In *Proceedings of the 15th Eureopean Conference on Artificial Intelligence (ECAI2002)*, 345–349. IOS Press.

Labroche N, Monmarché N, and Venturini G (2003a). AntClust: Ant Clustering and Web Usage Mining. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003)*, volume 2723 of *Lecture Notes in Computer Science*, 25–36. Springer.

Labroche N, Monmarché N, and Venturini G (2003b). Visual clustering with artificial ants colonies. In *Proceedings of the 7th International Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES 2003)*, volume 2773 of *Lecture Notes in Computer Science*, 332–338. Springer.

Lambertsen G and Nishio N (2004). Dynamic Clustering Techniques in Sensor Networks. In *Proceedings of the 7th JSSST SIGSYS Workshop on Systems for Programming and Applications (SPA2004)*.

Leidenfrost R and Elmenreich W (2009). Firefly clock synchronization in an 802.15.4 wireless network. In *EURASIP Journal on Embedded Systems*.

Licata I and Sakaji A (editors) (2008). *Physics of Emergence and Organization*. World Scietific.

Lu Y, Marconi T, Bertels K, and Gaydadjiev G (2009). Online Task Scheduling for the FPGA-Based Partially Reconfigurable Systems. In J Becker, R Woods, PM Athanas, and F Morgan (editors), *ARC*, volume 5453 of *Lecture Notes in Computer Science*, 216–230. Springer.

Luhmann N (1997). *Die Gesellschaft der Gesellschaft*. Frankfurt/M.

Lumer E and Faieta B (1994). Diversity and adaptation in populations of clustering ants. In D Cliff, P Husbands, J Meyer, and S W (editors), *Proceedings of the Third International Conference on Simulation of Adaptive Behavior (SAB)*, 501–508. MIT Press.

Magg S and Boekhorst RT (2006). Interaction and Emergent Behaviour in Heterogeneous Groups of Artificial Agents. In *Proceedings of the 7th German Workshop on Artificial Life (GWAL-7)*. IOS Press.

Magg S and te Boekhorst R (2007). Pattern Formation in Homogeneous and Heterogeneous Swarms: Differences Between Versatile and Specialized Agents. In *Proceedings of the 2007 IEEE Symposium on Artificial Life (CI-ALife 2007)*.

Mamei M, Menezes R, Tolksdorf R, and Zambonelli F (2006). Case studies for self-organization in computer science. In *J. Syst. Archit.*, 52(8): 443–460.

Marzo D, Gleizes MP, and Karageorgos A (2006). Self-organisation and emergence in mas: An overview. In *Informatica*, 30: 45–54.

Matake N, Hiroyasu T, Miki M, and Senda T (2007). Multiobjective clustering with automatic k-determination for large-scale data. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation (GECCO07)*, 861–868. ACM Press.

Merkle D, Middendorf M, and Scheidler A (2004). Decentralized Packet Clustering in Networks. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*. IEEE Computer Society.

Merkle D, Middendorf M, and Scheidler A (2005a). Decentralized Packet Clustering in Router-based Networks. In *International Journal of Foundations of Computer Science*, 16(2): 321–341.

Merkle D, Middendorf M, and Scheidler A (2005b). Dynamic Decentralized Packet Clustering in Networks. In F Rothlauf, J Branke, S Cagnoni, DW Corne, R Drechsler, Y Jin, P Machado, E Marchiori, J Romero, GD Smith, and G Squillero (editors), *EvoWorkshops*, volume 3449 of *Lecture Notes in Computer Science*, 574–583. Springer.

Merkle D, Middendorf M, and Scheidler A (2006a). Self-Organized Task Allocation for Computing Systems with Reconfigurable Components. In *Proceedings of the 9th International Workshop on Nature Inspired Distributed Computing (NIDISC'06)*. IEEE.

Merkle D, Middendorf M, and Scheidler A (2006b). Using Decentralized Clustering for Task Allocation in Networks with Reconfigurable Helper Units. In *Proc. International Workshop on Self-Organizing Systems 2006 (IWSOS 2006)*, number 4124 in Lecture Notes in Computer Science, 137–147. Springer, Berlin.

Merkle D, Middendorf M, and Scheidler A (2007). Swarm Controlled Emergence - Designing an Anti-Clustering Ant System. In *Proceedings of IEEE Swarm Intelligence Symposium (SIS 2007)*. Honolulu, Hawaii.

Merkle D, Middendorf M, and Scheidler A (2008). Self-Organized Task Allocation for Service Tasks in Computing Systems with Reconfigurable Components. In *Journal of Mathematical Modelling and Algorithms*, 7(2): 237–254.

Minati G and Pessa E (2006). *Collective Beings (Contemporary Systems Thinking)*. Springer.

Mirollo RE and Strogatz SH (1990). Synchronization of pulse-coupled biological oscillators. In *SIAM Journal on Applied Mathematics*, 50(6): 1645–1662.

Mnif M and Müller-Schloer C (2006). Quantitative Emergence. In *Proceedings of the 2006 IEEE Mountain Workshop on Adaptive and Learning Systems (SMCals 2006)*, 78–84. IEEE.

Mnif M, Richter U, Branke J, Schmeck H, and Müller-Schloer C (2007). Measurement and Control of Self-organised Behaviour in Robot Swarms. In P Lukowicz, L Thiele, and G Tröster (editors), *Proceedings of the 20th InternationalConference on Architecture of Computing Systems (ARCS2007)*, volume 4415 of *Lecture Notes in Computer Science*, 209–223. Springer.

Mogul JC (2006). Emergent (mis)behavior vs. complex software systems. In *SIGOPS Operating Systems Review*, 40(4): 293–304.

Monmarché N, Slimane M, and Venturini G (1999). AntClass: discovery of clusters in numeric data by an hybridization of an ant colony with the kmeans algorithm, Technical Report 213. Technical report, Laboratoire d'Informatique de l'Université de Tours.

Mühl G, Werner M, Jaeger MA, Herrmann K, and Parzyjegla H (2007). On the Definitions of Self-Managing and Self-Organizing Systems. In *KiVS 2007 Workshop: Selbstorganisierende, Adaptive, Kontextsensitive verteilte Systeme (SAKS 2007)*, 291–301. VDE Verlag.

Müller-Schloer C (2004). Organic computing: on the feasibility of controlled emergence. In *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2004*, 2–5. ACM Press.

Müller-Schloer C and Sick B (2006). Emergence in Organic Computing Systems: Discussion of a Controversial Concept. In *Proceedings of the 3rd International Conference on Automic and Trusted Computing*, volume 4158 of *Lecture Notes in Computer Science*, 1–16. Springer.

Nagel K and Schreckenberg M (1992). A cellular automaton model for freeway traffic. In *Journal de Physique I France*, 2(12): 2221–2229.

Nicolas G and Sillans D (1989). Immediate and latent effects of carbon dioxide on insects. In *Annual Review of Entomology*, 34: 87–116.

Nouyan S, Ghizzioli R, Birattari M, and Dorigo M (2005). An insect-based algorithm for the dynamic task allocation problem. In *Künstliche Intelligenz*, 4: 25–31.

Ohayon EL, Kwan HC, Burnham WM, Suffczynski P, and Kalitzin S (2004). Emergent complex patterns in autonomous distributed systems: mechanisms for attention recovery and relation to models of clinical epilepsy. In *SMC (2)*, 2066–2072.

Panait L and Luke S (2005). Cooperative Multi-Agent Learning: The State of the Art. In *Autonomous Agents and Multi-Agent Systems*, 11(3): 387–434.

Parunak HVD and Brueckner SA (2004). *Methodologies and Software Engineering for Agent Systems*, volume 11 of *Multiagent Systems, Artificial Societies, And Simulated Organizations*, chapter Engineering Swarming Systems, 341–376. Springer.

Peterson G, Mayer C, and Kubler T (2008). Ant clustering with locally weighted ant perception and diversified memory. In *Swarm Intelligence*, 2(1): 43–68.

Pini G, Brutschy A, Birattari M, and Dorigo M (2009). Interference Reduction through Task Partitioning in a Robotic Swarm - Or: ”Don’t you Step on My Blue Suede Shoes!”. In *Proceedings of the 6th International Conference on Informatics in Control, Automation and Robotics, Volume Robotics and Automation (ICINCO 2009)*, 52–59. INSTICC Press.

Potter MA and Jong KAD (2000). Cooperative Coevolution: An Architecture for Evolving Coadapted Subcomponents. In *Evolutionary Computation*, 8(1): 1–29.

Potter MA, Jong KAD, and Grefenstette JJ (1995). A Coevolutionary Approach to Learning Sequential Decision Rules. In *Proceedings of the 6th International Conference on Genetic Algorithms*, 366–372. Morgan Kaufmann Publishers Inc.

Price R and Tino P (2004). Evaluation of Adaptive Nature Inspired Task Allocation Against Alternate Decentralised Multiagent Strategies. In *Parallel Problem Solving from Nature - PPSN VIII, 8th International Conference, Birmingham, UK, September 18-22, 2004, Proceedings*, volume 3242 of *Lecture Notes in Computer Science*, 982–990. Springer.

Prothmann H, Branke J, Schmeck H, Tomforde S, Rochner F, Hähner J, and Müller-Schloer C (2009). Organic traffic light control for urban road networks. In *International Journal of Autonomous and Adaptive Communications Systems*, 2(3): 203–225.

Prothmann H, Rochner F, Tomforde S, Branke J, Müller-Schloer C, and Schmeck H (2008). Organic Control of Traffic Lights. In *Proceedings of the 5th International Conference on Autonomic and Trusted Computing (ATC-08)*, volume 5060 of *Lecture Notes in Computer Science*, 219–233. Springer.

Ramakrishnan KK and Yang H (1994). The Ethernet Capture Effect: Analysis and Solution. In *Proceedings of the 19th Conference on Local Computer Networks*, 228–240.

Ramos V and Merelo JJ (2002). Self-Organized Stigmergic Document Maps: Environment as a Mechanism for Context Learning. In *Proceedings of the 1st Spanish Conference on Evolutionary and Bio-Inspired Algorithms (AEBA2002)*, 284–293.

Ribock O, Richter U, and Schmeck H (2008). Using Organic Computing to Control Bunching Effects. In *Proceedings of the 21th International Conference on Architecture of Computing Systems (ARCS 2008)*, volume 4934 of *Lecture Notes in Computer Science*, 232–244. Springer.

Richert W, Kleinjohann B, and Kleinjohann L (2005). Learning Action Sequences through Imitation in Behavior Based Architectures. In *Systems Aspects in Organic and Pervasive Computing – ARCS 2005*, volume 3432 of *Lecture Notes in Computer Science*, 93–107. Springer.

Richter U, Mnif M, Branke J, Müller-Schloer C, and Schmeck H (2006). Towards a generic observer/controller architecture for Organic Computing. In *GI Jahrestagung (1)*, volume 93 of *Lecture Notes in Informatics*, 112–119. GI.

Richter U, Prothmann H, and Schmeck H (2008). Improving XCS Performance by Distribution. In *Proceedings of the 7th International Conference on Simulated Evolution And Learning (SEAL2008)*, volume 5361 of *Lecture Notes in Computer Science*, 111–120. Springer.

Rijsbergen CJv (1979). *Information retrieval*. Butterworths, London, 2 edition.

Rivest RL (1987). Learning Decision Lists. In *Machine Learning*, 2(3): 229–246.

Rochner F and Müller-Schloer C (2005). Emergence in technical systems. In *it - Information Technology*, 47(4): 195–200.

Runkler TA (2005). Ant colony optimization of clustering models. In *International Journal of Intelligent Systems*, 20(12): 1233–1251.

Samaey G, Holvoet T, and De Wolf T (2008). Using Equation-Free Macroscopic Analysis for Studying Self-Organising Emergent Solutions. In *Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO2008)*, 425–434. IEEE Press.

Sapin E and Bull L (2007). Searching for Glider Guns in Cellular Automata: Exploring Evolutionary and Other Techniques. In *Proceedings of the 8th International Conference on Artificial Evolution (EA 2007), Revised Selected Papers*, volume 4926 of *Lecture Notes in Computer Science*, 255–265. Springer.

Scheidler A (2005). *Modellierung und Simulation von Verteilungsmustern bei Ameisen*. Master's thesis, University of Leipzig, Germany.

Scheidler A, Blum C, Merkle D, and Middendorf M (2008a). Emergent Sorting in Networks of Router Agents. In M Dorigo, M Birattari, C Blum, M Clerc, T Stützle, and AFT Winfield (editors), *ANTS Conference*, volume 5217 of *Lecture Notes in Computer Science*, 299–306. Springer.

Scheidler A, Merkle D, and Middendorf M (2006). Emergent Sorting Patterns and Individual Differences of Randomly Moving Ant Like Agents. In *Proceedings of the 7th German Workshop on Artificial Life (GWAL-7)*, 105–115. IOS Press.

Scheidler A, Merkle D, and Middendorf M (2007). Stability and Performance of Ant Queue Inspired Task Division Methods. In *Proceedings of the European Conference on Complex Systems 2007 (ECCS 2007)*. Dresden.

Scheidler A, Merkle D, and Middendorf M (2008b). Congestion Control in Ant Like Moving Agent Systems. In *Proceedings of the 2nd IFIP Conference on Biologically Inspired Collaborative Computing (BICC08)*, volume 268 of *IFIP Advances in Information and Communication Technology*, 246–256.

Scheidler A, Merkle D, and Middendorf M (2008c). Stability and Performance of Ant Queue Inspired Task Partitioning Methods. In *Theory in Biosciences*, Volume 127 (2): 149–161.

Scheidler A and Middendorf M (2009a). Evolved cooperation and emergent communication structures in learning classifier based organic computing systems. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO09)*, 2633–2640. ACM Press.

Scheidler A and Middendorf M (2009b). Invited Talk: Communication Patterns in Decentralized Learning Classifier Systems. In *Proceedings of Third International ICST Conference on Autonomic Computing and Communication Systems (Autonomics 2009)*, Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (LNICST).

Schockaert S, Cock MD, Cornelis C, and Kerre EE (2004). Fuzzy Ant Based Clustering. In *Proceedings of the 4th International Workshop on Ant Colony Optimization and Swarm Intelligence (ANTS2004)*, volume 3172 of *Lecture Notes in Computer Science*, 342–349. Springer.

Schockaert S, Cock MD, Cornelis C, and Kerre EE (2007). Clustering web search results using fuzzy ants. In *International Journal on Intelligent Systems*, 22(5): 455–474.

Schöler T and Müller-Schloer C (2005). First Steps Towards Organic Computing Systems: Monitoring an Adaptive Protocol Stack with a Fuzzy Classifier System. In *Proceedings of the 2nd conference on Computing frontiers (CF'05)*, 10–20. ACM Press.

Schöler T and Müller-Schloer C (2006). An Observer/Controller Architecture for Adaptive Reconfigurable Stacks. In *Proceedings of the 18th InternationalConference on Architecture of Computing Systems (ARCS2005)*, volume 3432 of *Lecture Notes in Computer Science*, 139–153. Springer.

Sendova-Franks AB and Lent JV (2002). Random walk models of worker sorting in ant colonies. In *Journal of Theoretical Biology*, 217: 255–274.

Shah R, Krishnaswamy S, and Gaber MM (2005). Resource-aware very fast K-Means for ubiquitous data stream mining. In *In Proceedings of 2nd International Workshop on Knowledge Discovery in Data Streams*.

Sigaud O and Wilson SW (2007). Learning classifier systems: a survey. In *Soft Computing*, 11(11): 1065–1078.

Sudeikat J, Braubach L, Pokahr A, Renz W, and Lamersdorf W (2009). Systematically Engineering Self-Organizing Systems: The SodekoVS Approach. In *ECEASST*, 17: 1–12.

Sudeikat J and Renz W (2008). Toward Systemic MAS Development: Enforcing Decentralized SelfâĂŞOrganization by Composition and Refinement of Archetype Dynamics. In *Proceedings of Engineering Environment-Mediated Multiagent Systems (EEMMAS07)*, volume 5049 of *Lecture Notes In Artificial Intelligence*, 39–57. Springer.

Sumpter D, Krause J, James R, Couzin I, and Ward A (2008). Consensus Decision Making by Fish. In *Current Biology*, 18(22): 1773–1777.

Sutton RS and Barto AG (1998). *Reinforcement Learning I: Introduction*. MIT Press.

Symons J (2008). Computational Models of Emergent Properties. In *Minds and Machines*, 18(4).

Theraulaz G, Bonabeau E, , and Deneubourg J (1998). Response threshold reinforcement and division of labour in insect societies. In *Proceedings of the Royal Society London B*.

Theraulaz G, Bonabeau E, Nicolis S, Solé R, Fourcassié V, Blanco S, Fournier R, Joly JL, Fernández P, Grimal A, Dalle P, and Deneubourg JL (2002). Spatial patterns in ant colonies. In *Proceedings of the National Academy of Science*, 99: 9645–9649.

Theraulaz G, Goss S, Gervet J, and Deneubourg JL (1991). Task differentiation in polistes wasp colonies: A model for self-organizing groups of robots. In *From Animals* *to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, 346–355.

Tomforde S, Steffen M, Hähner J, and Müller-Schloer C (2009). Towards an Organic Network Control System. In *Proceedings of the 6th International Conference on Autonomic and Trusted Computing (ATC2009)*, volume 5586 of *Lecture Notes in Computer Science*, 2–16. Springer.

Tozier WA, Chesher MR, and Devgan TS (2006). The Brueckner Network: An Immobile Sorting Swarm. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2006)*, 91–92. ACM press.

Trianni V (2008). *Evolutionary Swarm Robotics: Evolving Self-Organising Behaviours in Groups of Autonomous Robots*, volume 108 of *Studies in Computational Intelligence*. Springer.

Tyrrell A, Auer G, and Bettstetter C (2007). Biologically Inspired Synchronization for Wireless Networks. In F Dressler and I Carreras (editors), *Advances in Biologically Inspired Information Systems: Models, Methods, and Tools*, volume 69 of *Studies in Computational Intelligence*, 47–62. Springer.

Vizine A, Castro L, Hruschka E, and Gudwin R (2005). Towards improving clustering ants: An adaptive ant clustering algorithm. In *Informatica*, 29: 143–154.

Waibel M, Floreano D, Magnenat S, and Keller L (2006). Division of labour and colony efficiency in social insects: effects of interactions between genetic architecture, colony kin structure and rate of perturbations. In *Proceedings of the Royal Society B*, 273: 1815–1823.

Wakamiya N, Hyodo K, and Murata M (2008). Reaction-Diffusion Based Topology Self-Organization for Periodic Data Gathering in Wireless Sensor Networks. In *Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2008)*, 351–360. IEEE Press.

Wilson SW (1994). ZCS: A Zeroth Level Classifier System. In *Evolutionary Computation*, 2(1): 1–18.

Wilson SW (1995). Classifier Fitness Based on Accuracy. In *Evolutionary Computation*, 3(2): 149–175.

Wolf TD and Holvoet T (2005). Towards a Methodology for Engineering Self-Organising Emergent Systems. In *Self-Organization and Autonomic Informatics (I),* *Proceedings of the SOAS 2005 Conference*, volume 135 of *Frontiers in Artificial Intelligence and Applications*, 18–34. IOS Press.

Wyatt TD (2003). *Pheromones and Animal Behaviour: Communication by Smell and Taste*. Cambridge University Press.

Yamins D (2005). Towards a theory of "local to global" in distributed multi-agent systems (I). In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, 183–190. ACM.

Zapf M and Weise T (2007). Offline Emergence Engineering For Agent Societies. In *Proceedings of the Fifth European Workshop on Multi-Agent Systems (EUMAS 07)*.

Author's Publications

SCHEIDLER A AND MIDDENDORF M (2009). Evolved cooperation and emergent communication structures in learning classifier based organic computing systems. In F Rothlauf (editor), *GECCO (Companion)*, 2633–2640. ACM Press.

SCHEIDLER A AND MIDDENDORF M (2009). Invited Talk: Communication Patterns in Decentralized Learning Classifier Systems. In *Proceedings of Third International ICST Conference on Autonomic Computing and Communication Systems (Autonomics 2009)*, Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (LNICST).

DIWOLD K, SCHEIDLER A, AND MIDDENDORF M (2009). The Effect of Spatial Organisation in Response Threshold Models for Social Insects. In *Proceedings of the European Conference on Complex Systems (ECCS 2009)*. University of Warwick, UK.

HERNÁNDEZ H, BLUM C, MIDDENDORF M, RAMSCH K, AND SCHEIDLER A (2009). Self-synchronized duty-cycling for mobile sensor networks with energy harvesting capabilities: A swarm intelligence study. In *Proceedings the IEEE Swarm Intelligence Symposium (SIS 2009)*, 153–159. IEEE press.

SCHEIDLER A, BLUM C, MERKLE D, AND MIDDENDORF M (2008). Emergent Sorting in Networks of Router Agents. In M Dorigo, M Birattari, C Blum, M Clerc, T Stützle, and AFT Winfield (editors), *ANTS Conference*, volume 5217 of *Lecture Notes in Computer Science*, 299–306. Springer.

SCHEIDLER A, MERKLE D, AND MIDDENDORF M (2008). Congestion Control in Ant Like Moving Agent Systems. In *Proceedings of the 2nd IFIP Conference on Biologically Inspired Collaborative Computing (BICC08)*, volume 268 of *IFIP Advances in Information and Communication Technology*, 246–256.

SCHEIDLER A, MERKLE D, AND MIDDENDORF M (2008). Stability and Performance of Ant Queue Inspired Task Partitioning Methods. In *Theory in Biosciences*, Volume 127 (2): 149–161.

BRUTSCHY A, SCHEIDLER A, MERKLE D, AND MIDDENDORF M (2008). Learning from House-Hunting Ants: Collective Decision-Making in Organic Computing Systems. In

Proceedings of the Sixth International Conference on Ant Colony Optimization and Swarm Intelligence (ANTS08), volume 5217 of *Lecture Notes in Computer Science*, 96–107. Springer.

Merkle D, Middendorf M, and Scheidler A (2008). *Organic Computing*, chapter Self-Adaptive Worker-Helper Systems with Self-Organized Task Allocation, 221–240. Springer.

Merkle D, Middendorf M, and Scheidler A (2008). Self-Organized Task Allocation for Service Tasks in Computing Systems with Reconfigurable Components. In *Journal of Mathematical Modelling and Algorithms*, 7(2): 237–254.

Merkle D, Middendorf M, and Scheidler A (2007). Swarm Controlled Emergence - Designing an Anti-Clustering Ant System. In *Proceedings of IEEE Swarm Intelligence Symposium (SIS 2007)*. Honolulu, Hawaii.

Scheidler A, Merkle D, and Middendorf M (2007). Stability and Performance of Ant Queue Inspired Task Division Methods. In *Proceedings of the European Conference on Complex Systems 2007 (ECCS 2007)*. Dresden.

Scheidler A, Merkle D, and Middendorf M (2006). Emergent Sorting Patterns and Individual Differences of Randomly Moving Ant Like Agents. In *Proceedings of the 7th German Workshop on Artificial Life (GWAL-7)*, 105–115. IOS Press.

Merkle D, Middendorf M, and Scheidler A (2006). Modelling Ant Brood Tending Patterns with Cellular Automata. In *Journal of Cellular Automata*, 2(2): 332–336.

Merkle D, Middendorf M, and Scheidler A (2006). Self-Organized Task Allocation for Computing Systems with Reconfigurable Components. In *Proceedings of the 9th International Workshop on Nature Inspired Distributed Computing (NIDISC'06)*. IEEE.

Merkle D, Middendorf M, and Scheidler A (2006). Using Decentralized Clustering for Task Allocation in Networks with Reconfigurable Helper Units. In *Proc. International Workshop on Self-Organizing Systems 2006 (IWSOS 2006)*, number 4124 in Lecture Notes in Computer Science, 137–147. Springer, Berlin.

Merkle D, Middendorf M, and Scheidler A (2005). Decentralized Packet Clustering in Router-based Networks. In *International Journal of Foundations of Computer Science*, 16(2): 321–341.

Merkle D, Middendorf M, and Scheidler A (2005). Dynamic Decentralized Packet Clustering in Networks. In F Rothlauf, J Branke, S Cagnoni, DW Corne, R Drechsler, Y Jin, P Machado, E Marchiori, J Romero, GD Smith, and G Squillero (editors), *EvoWorkshops*, volume 3449 of *Lecture Notes in Computer Science*, 574–583. Springer.

Merkle D, Middendorf M, and Scheidler A (2005). Modelling Ant Brood Tending Behavior with Cellular Automata. In VS Sunderam, GD van Albada, PMA Sloot, and J Dongarra (editors), *International Conference on Computational Science (2)*, volume 3515 of *Lecture Notes in Computer Science*, 412–419. Springer.

Merkle D, Middendorf M, and Scheidler A (2004). Decentralized Packet Clustering in Networks. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*. IEEE Computer Society.

Scheidler A (2005). A Comparison between Two Interval-based Time Ontologies. Technical report, Onto-Med Report Nr. 10. Research Group Ontologies in Medicine (Onto-Med), University of Leipzig.

Scheidler A (2005). *Modellierung und Simulation von Verteilungsmustern bei Ameisen.* Master's thesis, University of Leipzig, Germany.

Submitted: Scheidler A, Merkle D, and Middendorf M (2009). Swarm Controlled Emergence. to IEEE Transactions on Systems, Man, and Cybernetics Part B

Submitted: Merkle D, Middendorf M, and Scheidler A (2009). Task Allocation in Organic Computing Systems: Networks with Reconfigurable Helper Units. to International Journal of Autonomous and Adaptive Communications Systems.

Submitted: Aderhold A, Diwold K, Scheidler A, and Middendorf M (2009). Artificial Bee Colony Optimization: A new Selection Scheme and its Performance. to NICSO 2010.

Leipzig, 4. Dezember 2009

Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Dissertation selbständig und ohne unzulässige fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angeführten Quellen und Hilfsmittel benutzt und sämtliche Textstellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen wurden, und alle Angaben, die auf mündlichen Auskünften beruhen, als solche kenntlich gemacht. Ebenfalls sind alle von anderen Personen bereitgestellten Materialen oder erbrachten Dienstleistungen als solche gekennzeichnet.

Alexander Scheidler

Curriculum Vitae

Zur Person

Name:	Alexander Scheidler
Anschrift:	In der Ecke 14, 99947 Hörselberg-Hainich OT Wolfsbehringen
Geboren:	20. Dezember 1977 in Eisenach

Wissenschaftlicher Werdegang

1996	Abitur am Spezialschulteil des Albert-Schweitzer-Gymnasiums Erfurt
1997 - 2005	Diplomstudium Informatik mit Nebenfach Philosophie an der Universität Leipzig Spezialisierung: Angewandte Informatik Diplomarbeit: *Modellierung und Simulation von Verteilungsmustern bei Ameisen*; Betreuer: Prof. Dr. M. Middendorf
2005 - 2009	Wissenschaftlicher Mitarbeiter im Projekt "Organisation and Control of Self-Organising Systems in Technical Compounds " (DFG SPP 1183)

Leipzig, den 04. Dezember 2009