THE USE OF ORGANIZATIONAL SELF-DESIGN TO COORDINATE MULTIAGENT SYSTEMS

by

Sachin Kamboj

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer and Information Sciences

Fall 2009

THE USE OF ORGANIZATIONAL SELF-DESIGN TO COORDINATE MULTIAGENT SYSTEMS

by

Sachin Kamboj

Approved: ____
B. David Saunders, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved: ____
George H. Watson, Ph.D.
Interim Dean of the College of Arts and Sciences

Approved: ____
Debra Hess Norris, M.S.
Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____

Keith S. Decker, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____

Errol Lloyd, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____

Martin Swany, PhD.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____

Victor Lesser, Ph.D.
Member of dissertation committee

To my parents, Malti Kamboj and Shyam Sunder Kamboj

ACKNOWLEDGEMENTS

If I have been able to see further, it was only because I stood on the shoulders of giants. (Sir Issac Newton and others)

If it takes a village to raise a child, it probably took a small town to raise me to the point where I could write this dissertation. Indeed thanking everyone who helped me along the way would fill a dissertation-sized document in itself, so I will limit myself to the principal actors — my teachers, my family and my friends.

First of all, I would like to thank my advisor Keith Decker, without whom this dissertation would not have been remotely possible. Indeed Keith was like a central guiding father figure during my years at the University of Delaware (UD) and he helped me in many big and small ways. I particularly enjoyed and benefited from our discussions about my research and I hope your example, intellect and insight made me not only a better researcher but also a better presenter and teacher.

Next, I would like to thank my dissertation committee for their comments and insights. Particularly, Victor Lesser for taking such a fervent interest in the work of such a lowly researcher as myself. I, especially, benefited from your comments and questions on my proposal and I'd like to thank you for forcing me to challenge my own thinking and to look at the broader picture. I would also like to thank Errol Lloyd for answering my numerous queries on algorithm design and Martin Swany for our discussions on applying my research to the field of grid computing. Indeed, I was fortunate to have the both of you on my committee. I would also like to thank Michela Taufer for being like a "non-official" committee member and helping me with the application of my research to volunteer computing.

I would be amiss not to thank my former advisor Vijay Shanker for taking a freshly minted graduate student under his tutelage and for teaching me the ropes of research. Truth be told, you first taught be how to do research and for this I will forever be grateful.

I would also like to thank all the faculty at the University of Delaware, particularly Paul Amer, David Saunders, Kathy McCoy, Sandra Carberry and Lori Pollock. Thanks, not only for the knowledge that you imparted but also for being such excellent guardians and role models.

I also owe my gratitude to Joan Burnside, Robin Morgan and David Mills for hiring me as a research assistant and for supporting me financially while I worked on my dissertation.

Moving on to my family, I am particularly grateful to my parents for being a constant source of encouragement in all my endeavors. Thanks, not only for your love, patience and support but also for all the values that you tried to instill in me. Indeed it was your support that saw me through the many ups and downs of life. To my sister, Suruchi, thanks for showing me, by example, the importance of hard work and endurance and for the need to strive for perfection in all that we set to achieve.

Finally, I would not have survived a second without the support of all of my friends. To all of them, I say thanks for being there and supporting me despite the worst of my temperament. You provided me with a home away from home for which I shall forever be grateful.

Since my friends form such a big part of my life, I think they deserve at least a couple of paragraphs more, so here goes...

First, I would like to thank my lab-mates, James Atlas and Li Jin. I really enjoyed your company and our discussions on a whole slew of topics. Next, I'd like to thank Trilce Estrada for sharing the data she painstakingly collected for the

Dicking@Home project. I am also particularly grateful to Roger Craig and David and Emily Klein for befriending me when I first got here and for initiating me into the American way of life. The Computer and Information Sciences department at UD has been a really friendly place for a graduate student like myself and I would like to thank Manabu Torii, Keith Trnka, Ben Breech, Wei Chen, Umakanth Papulla, David Shepard, Joe Stelmach, Alvaro Gonzalez, Emily Gibson, Oana Tudor and the others for all the time they shared with me.

Next, I would like to thank the spirited set of characters that form the "UD Paltan" group. I could just say thank you for the all the chit-chat, movies, animated debates, hikes, camping trips, pot-luck dinners and the numerous other moments that we have shared over the years. However, I think each of you has helped me become a better person in a unique way and you have the right to at least a sentence for all the ways in which you have enriched my life. So (in alphabetical order by first names, for obvious reasons), thanks to Asmita (for keeping us sane and grounded and for looking after us like a mother hen), Gajju (for his advice and help in all worldly matters), Prabha (for the cooking and keeping me in line), Rahul (for his witty comments and sense of humor), Trisha (for reminding me of the value of humility and for forcing me to challenge all my preconceived notions) and Vamsi (for our shared love of technology and computers). To the former members of this group (Anoop, Divya, Hansneet, Keyur and Ryan), you are sorely missed.

Last but not least, I would commit a grave injustice to my undergrad friends if I did not acknowledge them here. So a special thanks to Aditya, Ashish, Jitesh, Pravin, Ritesh, Tushar and Vishwanath. I have so many fond memories of the four years that we spent together and I can surely say those were the best days of my life.

Thank You!

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Multiagent systems are increasingly being used to solve a wide variety of problems in a range of applications such as distributed sensing, information retrieval, workflow and business process management, air traffic control and spacecraft control, amongst others. Each of these systems has to be designed at two levels: the micro-architecture level, which involves the design of the individual agents and the macro-architecture level which involves the design of the agents' organizational structure. In this research, we are primarily concerned with the agents' macro-architecture.

At the macro-architecture level, the multiagent designer is concerned with issues such as the number of agents needed to solve the problem, the assignment of tasks to the agents and the coordination mechanisms being used. The design of the agents' macro-architecture is complicated by the fact that there is no best way to organize and all ways of organizing are not equally effective. Instead the optimal organizational structure depends on the problem at hand and the environmental conditions under which the problem needs to be solved. In some cases, the environmental conditions may not be known a priori, at design time, in which case the multi-agent designer does not know how to develop an optimal organizational structure. In other cases, the environmental conditions may change requiring a re-design of the agents' macro-architecture. These are only a few of the many hurdles confronting the macro-architecture designer.

In our research, we simplify the macro-architectural design by passing on some of the macro-architectural design responsibilities to the agents themselves.

That is, instead of manually designing the macro-architecture of a multiagent system at design time, we allow the agents to come up with their own organizational structure at run time. This approach is known as Organizational Self Design (OSD) and it allows the agents to adapt their organizational structure to changing environmental conditions and differences in the problems being solved.

Most of the current work on OSD has focused on task-oriented domains. In our research, we extend OSD to apply to worth-oriented domains, the hardest class of problems. Our research focuses on developing algorithms and mechanisms that allow (a) the generation of agents as an artifact of the system; and (b) the generation of different organizational structures that make different quality/cost tradeoffs based on the organizational design constraints specified and the performance criteria being optimized. Such tradeoffs are not possible in task-oriented and state-oriented domains.

Chapter 1

INTRODUCTION

In ancient times alchemists believed implicitly in a philosopher's stone which would provide the key to the universe and, in effect, solve all of the problems of mankind. The quest for coordination is in many respects the twentieth century equivalent of the medieval search for the philosopher's stone. If only we can find the right formula for coordination, we can reconcile the irreconcilable, harmonize competing and wholly divergent interests, overcome irrationalities in our government structures, and make hard policy choices to which no one will dissent.
(Harold Seidman: Politics, Position, and Power)

Multiagent systems are increasingly being used to solve a wide variety of problems in a range of applications such as distributed sensing, information retrieval, workflow and business process management, air traffic control and spacecraft control, amongst others. These systems have to be designed at two levels: the micro-architecture level, which involves the design of the individual agents and the macro-architecture level, which involves the design of the organizational and social aspects of the system. In our research, we are primarily interested in the macro-architectural, organizational design of the multiagent system.

At the organizational level, the multiagent designer is primarily concerned with issues such as:

- The number of agents needed to solve the problem
- The breakup of the problem into subtasks
- The allocation of the subtasks to the individual agents

- The distribution of the available resources amongst the agent population
- The coordination mechanisms being used to manage the interdependencies between the agent activities.

These issues can be resolved by, firstly, imposing an organizational structure on the agents and then by instantiating the chosen organizational structure with actual agents. The organizational structure consists of roles that the agents play and the manner in which they interact with other agents in the system. The instantiation consists of selecting the number of agents needed in the system and the assignment of roles and resources to the individual agents.

The organizational structure employed directly influences the effectiveness of the organization in solving the problem at hand, the resources needed by the agents and the cost of coordinating the activities of the individual agents. Hence, the organizational design is a very important part of the multiagent system design. However, there are few good rules and formal mechanisms for designing effective organizations for computational agents that are general enough for a wide range of agent systems. For example, consider the question of the number of agents needed in the system. If too few agents are available, the system will be overloaded and will not be able to perform optimally. If too many agents are used, resources may be wasted and contention for the limited resources amongst the agents will increase. However, few researchers have directly addressed this question.

The macro-architectural design is further complicated by the fact that there is no best way to organize and all ways of organizing are not equally effective [Carley & Gasser, 1999; So & Durfee, 1996][1]. Instead, the optimal organizational structure depends both on the problem at hand and the environmental conditions under

[1] The theory that there is no "one best way" to organize is sometimes referred to, in the organizational literature, as *Contingency Theory*, a term coined by Lawrence and Lorsch [Lawrence & Lorsch, 1967]. See [Scott, 1998] for more details.

which the problem needs to be solved. In some cases, the environmental conditions may not be known *a priori*, at design time, in which case the multi-agent designer does not know how to come up with the suitable organizational structure. In other cases, the environmental conditions may change requiring a redesign of the agents' macro-architecture. Hence, it is not obvious that a static design-time approach to an organizational structure is feasible in a significant number of cases. At the opposite end of the spectrum, systems may be designed to create a new, bespoke organizational design for every problem instance. The most popular example of such a one-off task allocation approach is the Contract Net approach [Smith & Davis, 1978; Smith, 1980, 1988] (See Section 2.2.5.1). Such an approach brings with it a different set of inefficiencies and belies the fact that while many real environments have dynamic components, there are also commonalities in the structure of problem instances that can be taken advantage of through proper organizational structuring. Hence, an alternative approach is needed for such situations in which the environment is dynamically, albeit slowly, changing. We will call such environments *semi-dynamic*.

In our research, we focus on the organizational design of such a subset of multiagent systems, that is, ones in which the environment is semi-dynamic. We show that most current approaches to organizational design either model the organization at design time, assuming a static environment, or generate a new organization on the fly, at run time, for each new instance of the problem and that such approaches are inefficient and fail to correctly model the dynamics of a slowly changing environment. Furthermore, we show that most of the existing approaches to organizational design (of multiagent systems in semi-dynamic environments) are not general enough to handle problem domains that are worth-oriented [Rosenschein & Zlotkin, 1994].

Towards these ends, we propose a dynamic run-time approach to the macro-architectural design, in which the agents use organizational-self design (OSD)

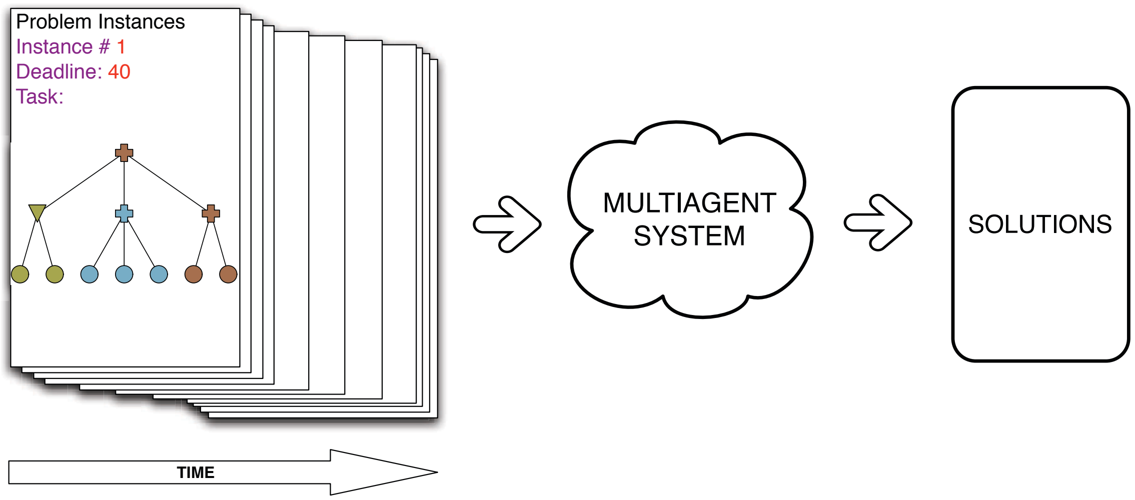


Figure 1.1: Figure showing our basic overall approach

[Gasser & Ishida, 1991; Ishida *et al.* , 1992; So & Durfee, 1993] to come up with their own organizational structure. In our approach (shown in Figure 1.1), problem solving requests arrive at the organization continuously at varying rates and with varying deadlines. To gain utility, the agents in the organization need to solve the problems by their given deadlines. In our model, we start off with a simple organization consisting of a single agent which is solely responsible for the problem solving activities of the entire organization. As new problem solving requests arrive, the agent checks to see whether it can complete the request by the given deadline. If not, the agent spawns off a new agent which is responsible for some subpart of the main problem, thus parallelizing the solution to the problem. Each agent in the organization is individually responsible for its subpart of the problem – hence the agent can be thought to fulfill the role of solving that subpart of the problem and achieving the subgoal characterized by that subpart, coordinated by some pre-arranged mechanism. If an agent cannot meet a deadline, it is individually responsible for spawning off a new agent and delegating a part of its responsibility to the new agent. If an agent is free for an extended period of time, it may decide to combine with another agent to save computational resources. Hence, we propose two organizational primitives that are responsible for generating the organizational structure: agent spawning and agent composition. These organizational primitives are further described in Chapter 3.

The organization of the rest of this chapter is as follows. We begin by enumerating our intellectual contributions in Section 1.1. We follow this up by describing why organization is hard in Section 1.2. Finally, we end with a description of the scope of our work in Section 1.3.

1.1 Intellectual Contributions

We divide our intellectual contributions into two parts: (a) Contributions from a multiagent perspective; and (b) Contributions from a Volunteer/Grid/Cloud Computing perspective. We primarily focus of the former and only briefly mention the latter.

From a multiagent perspective, our intellectual contributions are outlined below:

- We generate the agents as an artifact of the system. That is, given a set of problems to be solved, and a utility function that allows us to balance agent costs and performance, our multiagent framework generates the requisite number of agents needed to solve the problem. This is a significant departure from most traditional multiagent organizational techniques that assume a preexisting set of agents and then try to organize them.

  This is important because, whereas there are many problems (such as, personal assistant agents) in which there is a natural fit between the problems being solved and the agents in the system, there are many problems in which the fact that agents are used is secondary and irrelevant to the problems being solved. That is, the user is primarily interested in solving a set of problems and is not the least bit concerned with whether multiple agents or some other technology is used to solve the problems, as long as the technology lives up to his requirements. Also if there is no perfect mapping between the problems and the agents and if the environment is semi-dynamic, the user has no way of knowing, a priori, how many agents to use or in what configuration. Using too few agents will result in the agents being overloaded; using too many agents will result in wasted resources — either way, the system will be inefficient. Examples of such applications include distributed information gathering, bioinformatics applications, web service choreography and the efficient dynamic use of grid and cloud computing, etc. Furthermore, in such systems, extra resources can often be purchased as needed - hence, in such systems it makes sense to generate agents at run-time as an artifact of the system.

  In our framework, the user is not concerned with specifying the number of agents needed in the system or with specifying the organization of these agents. Instead all the user does is give a representation of the problems and the system generates the agents and organization needed to solve the problems. Also note that since more agents are equivalent to more resources (especially processor resources), our system allows resources to be traded for the number of problems being solved in a given time and the quality of the solutions.

  Hopefully, this will help simplify the use of a multi-agent framework for a broad range of cooperative, distributed problem-solving (CDPS) applications.

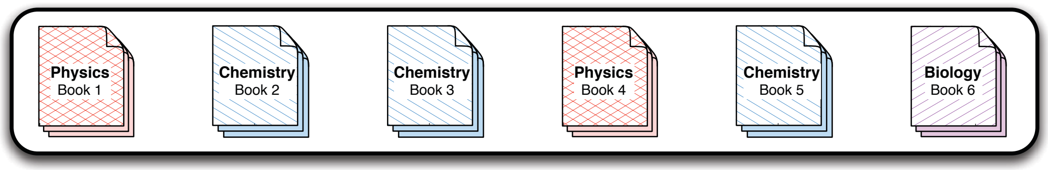
- The OSD approach is not new and has been proposed and used by a number of researchers over the last twenty years. However, none of the researchers have proposed applying it to worth-oriented domains — the most complex class of problem representations.

What are worth-oriented domains and how do they differ from other domains? [Rosenschein & Zlotkin, 1994] suggested that problem representations [2] may be modeled as belonging to one of three kinds of domains — Task-Oriented Domains, State-Oriented Domains and Worth-Oriented Domains. According to Rosenschein and Zlotkin:

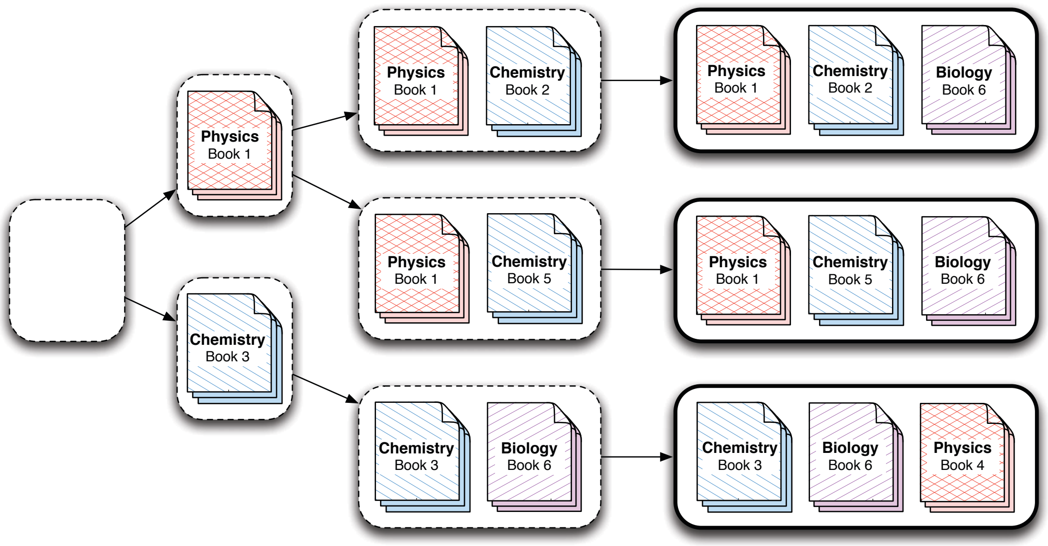
> In task oriented domains, a goal specifies a set of tasks that the agent is required to carry out. In state oriented domains, a goal specifies a set of states that the agent wishes to reach. In contrast, in worth-oriented domains the goal definition is subsumed by a worth function over all possible final states. Those states with the highest value of worth might be thought of as those that satisfy the full goal, while other with lower worth values, only partially specify the goals. ([Rosenschein & Zlotkin, 1994])

To better understand the difference between these three domains, consider the example of a "knowledge-seeking agent" that wishes to learn about the natural sciences by reading a series of books (See Figure 1.2). If this problem were to be represented in a task-oriented domain, the agent's goal would be to read as many books as possible on Physics, Chemistry and Biology. In a state-oriented domain, on the other hand, the agent would recognize that goal state is one in which the agent has acquired some requisite knowledge in each of the three subject areas — Physics, Chemistry and Biology. Hence, the initial state would be one in which our agent hasn't read any books and the final states would be ones in which an agent has read at-least one book in each of the three subject areas. Every book that the agent reads would lead to a transition from one state to another. Modeling this problem in the worth-oriented domain, the agent would recognize that not all books are written equal and that some books are better than others. Hence, this agent would assign a *utility* to each

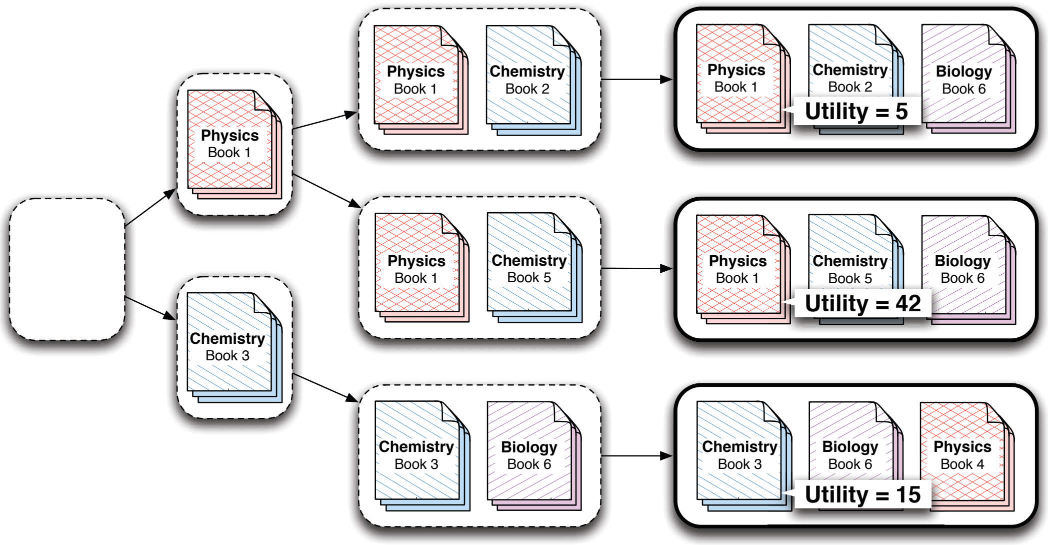
[2] There is usually a direct mapping from the problems that the agents are trying to solve to goals that the agents must achieve.



(a) Task Oriented Domain



(b) State Oriented Domain



(c) Worth Oriented Domain

Figure 1.2: Figure showing the three types of problem domains. The goal states are shown using bold rounded rectangles. Not all possible states are shown.

of the goal states — the goal state with the three worst books on each subject would have the lowest utility while a state in which every book on all three subjects has been read would have the highest utility. We can easily envision a situation in which reading a book would have a cost associated with it and in such situations the utility calculations would be based on a function of the benefit (quality of each book) and cost of reading each book.

As should be obvious from this example, the same problem may belong to different domains depending on how it has been framed, modeled and represented. Moving from a problem representation in a task-oriented domain to a problem representation in a worth-oriented domain allows better modeling of the quality/cost tradeoffs between the various solutions to the problem.

We used TÆMS (Task Analysis, Environment Modeling and Simulation) [Decker, 1995, 1996; Lesser *et al.* , 2002; Horling *et al.* , 1999] as our problem representation language, since it is a representation language specifically designed for worth-oriented domains[3]. TÆMS is a highly expressive general purpose task modeling language that allows the modeling of a broad range of problems belonging to the worth oriented domain.

Extending OSD to worth oriented domains allowed us to generate different organizational structures that make different quality/cost tradeoffs based on the organizational design constraints specified and the performance criteria being optimized. Different organizations use a different number of agents, have different resource requirements and generate different run-time characteristics (such as the quality of the results produced). For example, if there are two alternative ways of achieving a goal, representing the goal in TÆMS allows our OSD approach to select alternative organizations that achieve the maximum quality possible based on different time and cost constraints. Such tradeoffs

[3] See Section 3.1 for more details on TÆMS and for a justification of its use.

are not possible in task-oriented and state-oriented domains. Another positive side-effect of using TÆMS is that it allowed us to model uncertainties in the execution of tasks and to diminish the effects of these uncertainties on the performance of the organization.

Our research in this area offers key insights to the design of organizations for worth-oriented domains.

- We not only used TÆMS as our problem representation language, but we also extended TÆMS in two ways:

  1. We used TÆMS to represent the organizational structure of the agents. Specifically, we added organizational nodes to TÆMS that allow each agent to represent its local view of the organization — the roles that it is enacting and its relationship to other agents in the organization. All reorganization (agent spawning and composition) is defined in terms of rewriting the local TÆMS task structure of the agents. (See Section 3.4.1 for details.)

  2. We added *iteration nodes* to TÆMS . Iteration nodes allow the number of iterations of a sub-task to depend on the results of the execution of another method in the task structure. For example, the number of buffers required in a grid data transfer application might depend on the size and criticality of the data being transferred. The goal of allocating a buffer can then be represented using an *iteration node* — the number of iterations of this goal would depend on the results of executing a method that determines the size of the data being transferred. (See Section 3.3.4 for details.)

- One of the key advantages of using a one-off task allocation/organization scheme, like the Contract Net Protocol, is that it is more robust to agent and task failures[4]. This is because a new organization is generated from scratch for every problem instance, which basically allows the multiagent system to work around agent and task failures. This essentially limits the effect of any failure to a single problem instance and reduces the likelihood of a failure propagating to other problem instances. However, organizations that exist for longer than a single problem instance need some explicit way of detecting and overcoming failures. Hence, this problem is extremely important in the context of our OSD approach.

Most of the existing OSD approaches fail to consider the robustness of the generated organizations. Similarly, there has been limited research that explicitly looks into the robustness of CDPS applications that use TÆMS as their underlying framework.

We rectified these shortcomings in the current research on OSD and CDPS by studying the mechanisms through which organizations can be made more robust against agent and task failures. In this thesis, we implemented and evaluated the two commonly used approaches for adding robustness to multiagent systems — the citizens approach [Dellarocas & Klein, 2000] and the survivalist approach [Marin *et al.* , 2001; Briot *et al.* , 2006]. (See Section 3.4.7 for more details.)

We also studied the interplay between the organizational structure, the probability of failure and the desired level of robustness. One way of achieving a higher level of robustness in the survivalist approach, given a large numbers of agent failures, would be to relax the task deadlines. However, such a relaxation would result in the OSD approach using fewer agents in order

[4] This is not to say failures have no affect on the performance of the contract net protocol. This statement is about the comparative effect of failures. See [Dellarocas & Klein, 2000] for a more thorough investigation of the effect of failures on the performance of the contract net protocol.

to conserve resources, which in turn would have a detrimental effect on the robustness. Hence, we explored the robustness properties of task structures and the ways in which the organizational design can be modified to take such properties into account. Specifically we showed that the problem of computing the robustness properties of a task structure is NP-hard.

- Finally, we believe that task and resource allocation in multiagent systems is still an open problem. Through our work on OSD, we have offered key insights and developed new protocols to handle task and resource allocation for a broad range of problem.

  Unfortunately, coming up with an optimal[5] task allocation strategy (using, for example, Partially Observable Markov Decision Processes (POMDPs) [Sutton & Barto, 1998; Cassandra, 1998]) might not be feasible for a soft real-time algorithm that is expected to work at runtime (See Section 1.2). Hence, in this thesis, we have looked at three task allocation heuristics that can be used with organizational self-design and can generate satisficing solution to the task allocation problem. (See Section 3.4.5.1 for details.) We investigate how the variation in task allocation strategy affects the performance of the organization and the tradeoffs being made when using different strategies.

  We have also evaluated the tradeoffs between specialization and generalization [March & Simon, 1993]. Specialization involves breaking up a task into smaller subtasks that are assigned to the agents. The agents then become specialists in executing these tasks. Generalization, on the other hand, involves creating clones of agents that can handle different instances of a larger task. These agents are generalists, in that they are capable of achieving a larger set of tasks. We show that the choice between specialization and generalization

[5] For some definition of optimal. Such a definition of optimal will necessarily depend on the criteria being optimized.

depends on the task structure and environmental factors like the task arrival rate and the deadline of the tasks. (See Section 3.4.5.2 for details.)

We now very briefly mention our contributions from a grid/cloud/volunteer computing and web services perspective. In Grid Computing, one of the key issues is how to come up with Virtual Organizations (VO) [Foster *et al.* , 2001] that make efficient use of the grid resources. Similarly, Web Service Choreography deals with composing complex web services from simpler ones. We believe that our OSD approach can be applied (with some modification) to both of these key problems. Therefore, even though our focus is on multiagent systems, we are hoping that our work on OSD might provide key insights and may be applied to both grid computing and web services in the future.

In Chapter 5, we describe how our OSD research can be applied to the problem of determining a suitable scheduling policy for a volunteer computing system. A scheduling policy is simply an assignment of jobs (tasks) to volunteers (agents) so as to maximize the throughput and minimize the turnaround time of the volunteer computing system. We use simulations to show that our OSD approach performed better than the naive scheduling policy used in a current volunteer computing system, although implementing our approach would require significant architectural changes to that system.

1.2 Why is the problem hard

Designing and building an organization is an extremely complex problem. Firstly, the computational complexity of the problem makes it intractable for all but the simplest of problems and organizations. In his thesis, Bryan Horling [Horling, 2006], proves that instantiating a valid organization from a set of templates, at compile-time, is an *NEXP-Complete* problem[6]. And finding an optimal organization is considerably harder than finding a valid instantiation. Similar results are presented by Nair et. al. in [Nair *et al.* , 2003].

Building an organization at run-time has its own set of complexities and requirements. It might be acceptable to have an algorithm that takes a long time to build an organization, if the organization is being built *before* the agents are ever deployed in the field[7]. However, in the case of a run-time algorithm, any time spent making organizational decisions is time that is diverted from the domain-specific problem-solving that the agent should be doing. Ideally, the time spent making organizational decisions should be less than the time wasted due to an inefficient organization. Unfortunately, the problem of deciding when it is appropriate to keep an existing, inefficient organizational structure and when it makes sense to reorganize is an open research problem. Furthermore, the time taken to make organizational decisions should be considerably less than the time required for domain-specific problem solving. Hence, it is important for the run-time algorithms that we are interested in to be extremely efficient.

Not only is the problem of organizing computationally hard, but the problem of determining whether a satisficing organization can exist is itself extremely hard. Part of the problem is specifying what is needed of the organization and what the desired optimizing criteria are. Often the specified criteria are at odds with each other (for example, maximizing quality and minimizing cost) and it is not obvious how these conflicts can be resolved. But the other part of the problem — knowing whether the specified criteria and constraints are too strict to be met is a

[6] An *NEXP-Complete* problem is a problem in which the solutions to the problem can be verified in exponential time. Hence, an *NEXP-Complete* problem is considerably harder than an *NP-Complete* problem, which are usually used in Computer Science to describe the class of intractable problems.

[7] For example, consider a program like Openoffice that takes forever to compile but works efficiently once it has been compiled

considerably harder problem. Hence, *not only is the problem of coming up with an organization hard, but the problem of knowing whether an organization can exist is itself hard.*

Finally, the OSD problem has not been studied in any considerable depth. Hence, there is a lack of any understanding on what the issues are and how efficient algorithms can be designed to handle this problem. This complicates the design of algorithms for OSD.

1.3 Scope

In this thesis, we will **not** be looking at:

1. Scheduling, except where there is an interplay between scheduling and generating an organizational structure. We believe that scheduling is a separate open problem in its own right deserving a number of theses. Indeed, both Alan Garvey [Garvey & Lesser, 1996] and Tom Wagner [Wagner & Lesser, 2000] have looked at scheduling for TÆMS task structures in some detail. More recently, the entire *COORDINATORS* project has been studying the scheduling of TÆMS -based task structures for use in military missions planning [Sims *et al.* , 2006; Zimmerman *et al.* , 2007; Maheswaran *et al.* , 2008; Atlas, 2009].

2. Coming up with an accurate model of the environment. A model of the environment will be able to predict things like the kinds of tasks that the organization will have to deal with, the arrival time of tasks, the variation in the available resources over time, the failure rate of the agents, the failure rates of tasks, etc. Whereas, such a model of the environment would be of tremendous help in generating stable organizational structures, we believe that (1) real-world environments are highly unpredictable and extremely hard to model and (2) if such an accurate model exists, it will dispense with the need to generate organizational structures at run-time.

3. Generating organizations in which not all the agents are cooperative and benevolent. In our work, we are assuming that the agents are an artifact of the system, are a result of trying to solve a set of similar problems using a multiagent system and are generated and destroyed based on the needs of the system/organization. Hence, it follows that the primary aim of the agents is to increase the “utility” of the organization even if it is at their own expense.

4. Generating organizations for open environments. Open environments are those in which the agents are self-interested and heterogeneous and may join or leave the environment at any time. Again, since we are assuming that the agents are an artifact of the system, we will not be looking at such environments.

5. Trying to learn a per-agent subjective model of the task characteristics. In particular, we assume that the quality, cost and duration distributions as well as the resource requirements of the executable methods are known, a priori, before the agents try to execute a task structure. Also we assume that this knowledge is shared by all the agents in the organization.

1.4 Outline of the rest of this thesis

The organization of the rest of this thesis is as follows: Chapter 2 covers some background material on organizations and also describes existing literature relevant to our research. We follow this up with a detailed description of our approach to organizational self-design in Chapter 3. Chapter 4 then describes some experiments that we have conducted to evaluate our approach. Subsequently, in Chapter 5, we describe how our approach can be applied to real-world volunteer, grid and cloud computing systems. Finally, we conclude and describe our future work in Chapter 6.

Chapter 2

BACKGROUND

Every phase of evolution commences by being in a state of unstable force and proceeds through organization to equilibrium. Equilibrium having been achieved, no further development is possible without once more oversetting the stability and passing through a phase of contending forces.
(Kabbalah)

This chapter consists of two main parts. The first part of this chapter introduces the term "Organization" and describes its constituent parts. The second part of this chapter is a review of the related literature relevant to our work.

2.1 Organization of the Agents

Once a problem has been analyzed and an appropriate task structure generated, the multiagent designer needs to decide on the number of agents needed to solve the problem and the way in which those agents will be organized. By organized, we mean (1) what will be the duties and responsibilities of each agent, (2) how will the available resources be distributed amongst the available agents and (3) how will the agents be coordinated, that is, how will they manage the interdependencies between their activities.

Organizing a multiagent system typically involves coming up with a suitable *organizational structure* [Fox, 1981; Horling *et al.* , 2001] and instantiating that

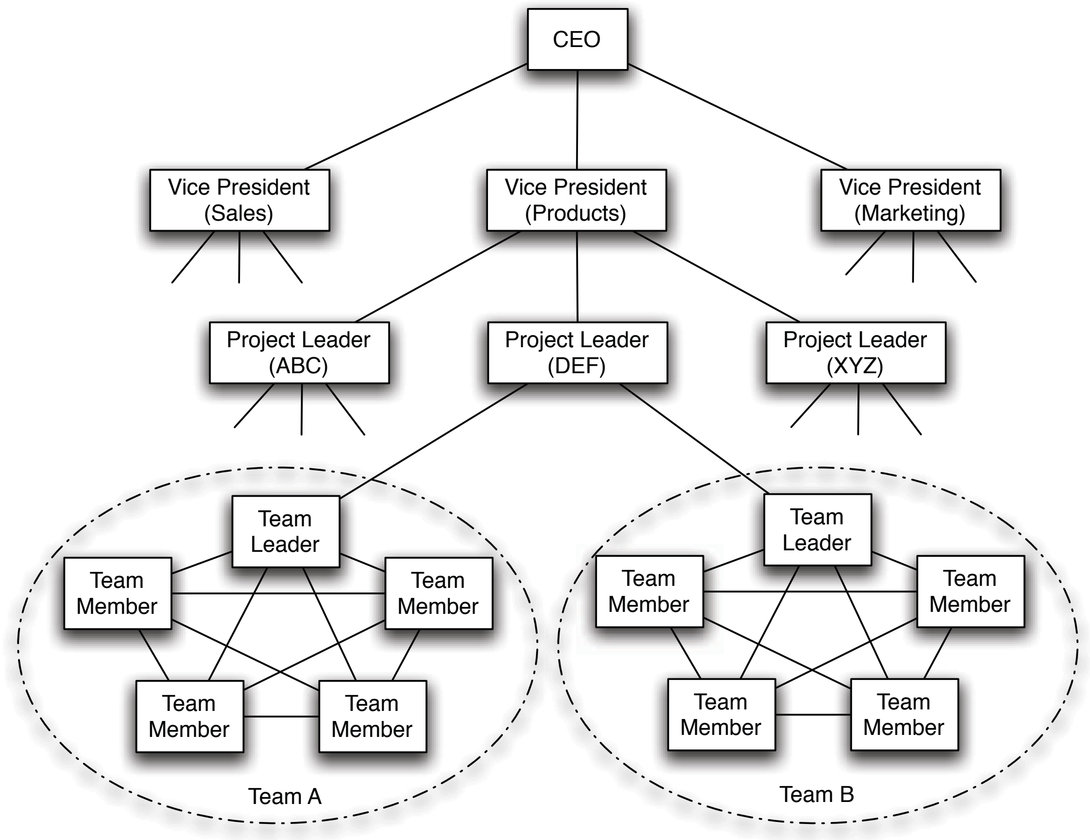


Figure 2.1: A semi-hierarchical organization for a traditional (human) corporation. The boxes represent the roles and the lines show the relationships between the roles.

structure with actual agents. See Figure 2.1 for an example of a traditional (human) organizational structure. An organizational structure consists of two major components:

1. *The roles:* The roles are used to answer the question of *who does what* in an organization and may be thought of as the parts played by the agents enacting the roles in the solution to the problem. The roles correspond to *positions* in the organizational hierarchy of human organizations. For example, a team leader is responsible for managing a team, a programmer is responsible

for writing code, while a CEO is responsible for the operation of the whole organization.

Hence, roles reflect the long-term commitments made by the agents enacting the roles to a certain course of action and typically include a description of (1) the job/task responsibility of the agents enacting the roles, (2) the skill-set needed for performing the role and (3) the resources available/allocated for execution of the role. More broadly, the roles include information about the deontic aspects of the organization — the obligations, permissions and prohibitions of the agents enacting these roles. Note that one or more agents may take on a single role and a single agent can participate in more than one role.

2. *The relationships between the roles:* The relationships between the roles are used to answer the question of *who talks to whom* and can be thought of as communication links between the roles. These relationships reflect the inter-dependencies between the tasks being performed by agents enacting different roles and are used to effectively coordinate the activities of these agents. The relationships correspond to authority relations in a hierarchical human organizations. For example, the CEO has authority over the vice presidents, who have authority over the project leaders.

   An explicit enumeration of relationships between the roles is needed for efficient coordination because it *constrains the amount of communication needed between the agents enacting the roles.* An agent only needs to communicate with other agents that are enacting roles that have direct relationships with a role that it is enacting. For example, a team member only needs to communicate with other team members for performing its job, whereas a team leader has to communicate with both the team members and the project leaders.

The organizational structure directly influences the effectiveness of the multiagent system in solving the problem at hand, the resources needed by the agents and the cost of coordinating the activities of the individual agents. Hence, the design of the organizational structure is a very important part of the whole multiagent system design. Unfortunately, there is no best way to organize and all ways of organizing are not equally effective [Carley & Gasser, 1999; Lawrence & Lorsch, 1967; Scott, 1998]. We will discuss a list of factors that affect the choice of an organizational structure in the next subsection. For a survey of various organizational paradigms, see [Horling & Lesser, 2005b].

Besides an organizational structure, organizations also consist of *rules, regulations, policies, procedures and conventions* that are used for effective coordination between the agents. These procedures are used to codify a coordination protocol between the agents. For example, in a human organization, members of a software team need to complete a set of documents (timeline charts, requirement documents, use cases, feature specs, UML diagrams, QA plans, etc) that are used to ensure that jobs are done properly in a coordinated fashion.

Similarly, in organizations made up of software agents, a set of coordination mechanisms is needed to ensure that the dependencies between the agent roles are properly managed. Examples of coordination mechanisms that can be used in such organizations are Generalized Partial Global Planning (GPGP) [Decker, 1995; Decker & Li, 2000; Lesser *et al.* , 2004; Chen & Decker, 2005], which involves the use of commitments for managing scheduling dependencies between tasks and Negotiation [Rosenschein & Zlotkin, 1994; Jennings *et al.* , 2001; Koifman *et al.* , 2004; Kraus, 2001], which involves the resolution of conflicts using various negotiation strategies.

2.1.1 Choosing an Organizational Structure

The main factors influencing the choice of an organizational structure are:

1. **The task structure:** It makes intuitive sense that the organizational structure should depend on the underlying task structure of the problem being solved. Indeed in human organizations, the kind of roles, skills and resources needed by an organization that deals with, say, manufacturing outdoor equipment is very different from the kind of roles, skills and resources needed by an organization whose primary purpose is to impart an education to college students. Table 2.1 shows some of the differences in the roles, skills and resources of different organizations.

Table 2.1: Table showing the difference in the roles, resources and skills required in two different organizations

Organization	**Roles**	**Resources**	**Skills**
Manufacturing Organization	Managers, Product Designers, Factory-shop Workers	Corporate Offices, Factories, Machines, Production Lines, Distributors, etc.	MBA's, tool-operating skills, etc.
Educational Organization	President, Deans, Professors, Lecturers	Classrooms, Libraries, Research Labs, etc.	Ph.D.s

Similarly, in multiagent organizations, the task structure directly influences the choice of organizational structures in the following ways:

- At a high level, the number of nodes in the task structure determines the number of possible roles in the organizational structure. This is because nodes in the task structure represent goals and equating nodes to roles is equivalent to assigning responsibility for achieving goals to the agents enacting the roles.[1] Hence, a "bigger" task structure with a

[1] Note that equating roles and goals does not preclude the possibility of having more than one agent responsible for a role and having an agent responsible for multiple roles.

greater number of nodes is likely to have a larger number of roles than a "smaller" task structure with few nodes.

- The number of non-local effects (enablements, disablements, facilitations, etc) and the number of sibling relationships[2] will directly influence the number of coordination relationships between the roles. Intuitively, a task structure with a large number of such relationships will be more difficult to coordinate as the decisions taken by an agent enacting a particular role directly influences a large number of other agents.

- The number of alternative ways of achieving a task directly affects the robustness of an organization with respect to that task. We will describe robustness in some detail later.

2. **The criteria being optimized:** The criteria being optimized defines what it means to have an "optimal" organization. *The goals of an organization determine what all the organization wishes to achieve, whereas the criteria determine how well the organization achieves its goals.* For example, consider two personal computer manufacturers. Since they are computer manufactures, both of these organizations have the same goal — to manufacture computers. However, they may be optimizing different criteria in trying to meet their goal — one of these organizations may be trying to manufacture the cheapest possible computers, while the other organization might be trying to manufacture the fastest, best designed computers. Now if the computers manufactured by the first organization are not the cheapest, then the first organization is not optimal. Similarly, if the computers manufactured by the second organization are not the best deigned, the second organization is not optimal. Hence, the criteria being optimized complement the goals of the organization.

[2] Nodes in a TÆMS task structure that have a common parent are said to have a sibling relationship with each other.

For most publicly traded corporations, the primary criteria being optimized is the shareholder value of the corporation, which is itself determined by a number of secondary criteria such as the profits earned, the market capitalization, etc. Similarly for organizations composed of software agents, the primary criteria being optimized might be an organizational utility function, which might be itself be dependent on various secondary criteria such as the quality of the produced solution or the time, cost and resources needed to produce the solution. Expressing the criteria being optimized as a utility function allows us to combine various different criteria (such as quality, cost, duration, available resources) in arbitrary ways.

Often, it is extremely difficult to come up with an optimal organization. Furthermore, it is often hard to come up with an accurate utility function for combining different and often conflicting criteria. Hence, the criteria are often specified in satisficing terms instead of optimizing terms. For example, the criteria might be specified as "The solution should have a minimum quality of 10 and a maximum cost of 100".

The criteria being optimized determines the kind of organizational structure in two ways:

(a) It affects the number of agents needed in the organization. The number of agents needed is directly contingent on the assignment of roles to the agents. Indeed, the number of agents assigned to each role is determined by (1) the time constraints by which the goal of a role needs to be achieved, (2) the time it takes for a single agent to achieve the goal and (3) the amount of resources available for completion of the goal. If the time it takes for an agent to achieve a goal is greater than than the deadline by which the goal needs to be completed, more than one agent needs to be assigned to the goal. The deadline or time constraints on

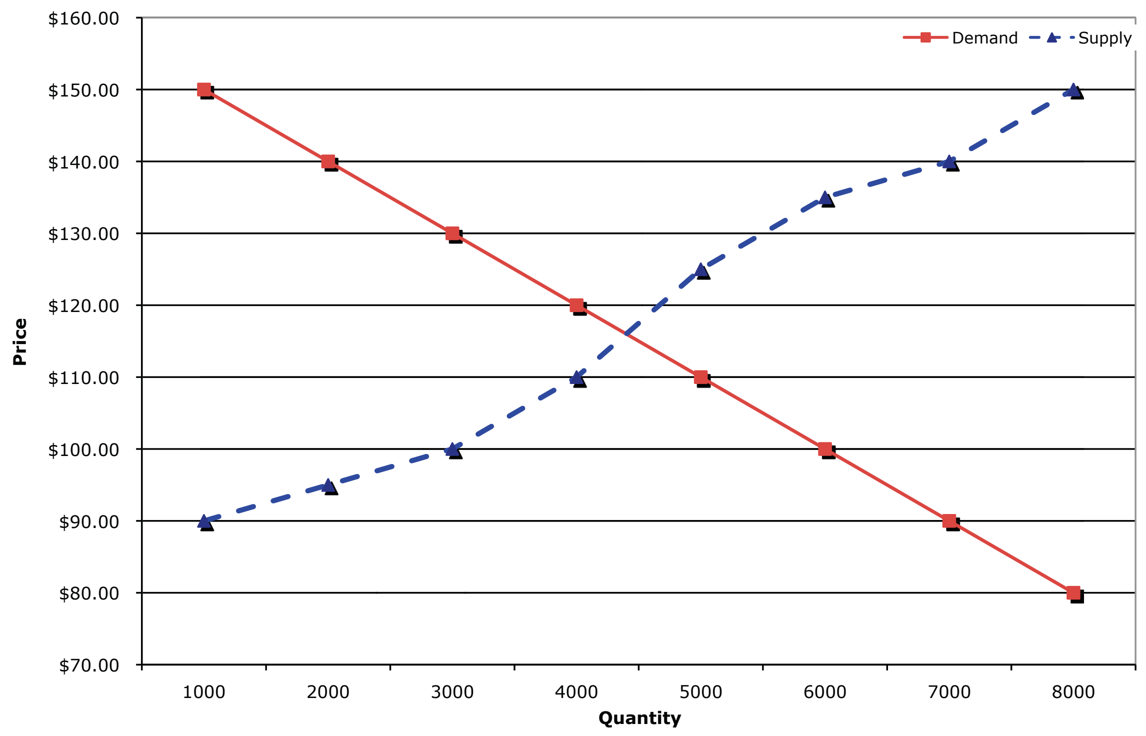


Figure 2.2: A Simple Supply-Demand graph for sleeping bags. Note that the independent variable, price, is shown on the y-axis and the dependent variable, quantity is shown on the x-axis.

a goal are usually directly or indirectly determined by the criteria being optimized by the organization.

To see how, consider an outdoor equipment manufacturing organization that wishes to maximize its profits. Also, for simplicity, let us assume that this organization only manufactures a single product, namely sleeping bags. Now the maximum profit that this organization can make is directly dependent on the number of sleeping bags sold. Also the quantity of sleeping bags manufactured must equal the maximum number of sleeping bags that the organization can possibly sell. Hence, the organization needs to compute a supply-demand curve for its sleeping bag, similar to the one shown in Figure 2.2[3]. In this figure, the demand curve (shown using the solid red line) shows the inverse relation between the price and the demand for sleeping bags — as the price increases, the demand drops (This is known as the law of demand). Similarly, the supply curve (shown using the blue dashed line) shows the positive relation between price and supply. As the price increases, the willingness and ability of manufacturers to produce goods increases. The optimal quantity and price is determined by the intersection of these curves[4]. From the figure, it is obvious that the corporation should manufacture 4400 sleeping bags per month. If a single factory-shop worker can manufacture 500 sleeping bags per month, the 9 workers are needed in an optimal organization.

Another way of looking at it is if too few agents are available, the system will be overloaded and will not be able to perform optimally. If too many agents are used, resources may be wasted and contention for the limited resources amongst the agents will increase.

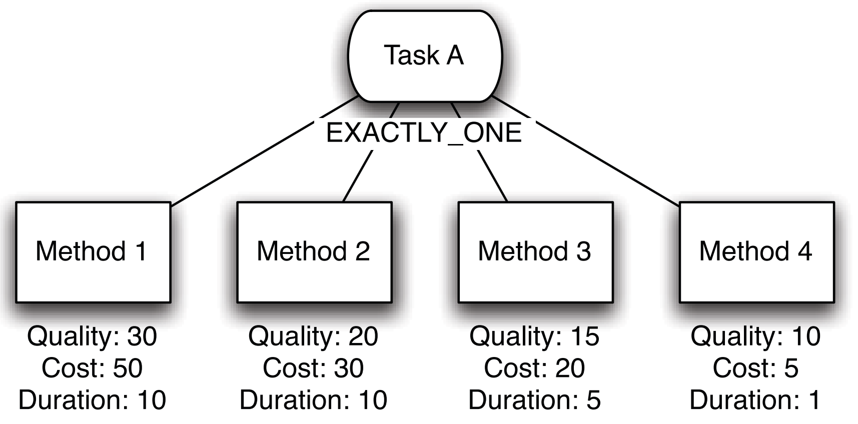
(b) It affects the selection of tasks. The desired criteria that is being optimized primarily affects the sets of tasks that can be scheduled. This in turn affects the kind of organizational structure that can be produced as it doesn't make sense to equate roles and assign agents to tasks that can never be scheduled.

To see how, consider the two hypothetical TÆMS task structure[5] shown in Figure 2.3. If a minimum quality of 15 is desired and a maximum cost

[3] By convention, supply-demand curves are shown with the independent-variable, the price, on the vertical y-axis and the dependent variable, the quantity on the horizontal x-axis.

[4] This is a very simple example that ignores a lot of market dynamics.

[5] TÆMS is simply a hierarchical task representation language that we use for our problem description. See Section 3.1 for a description and formal definition.



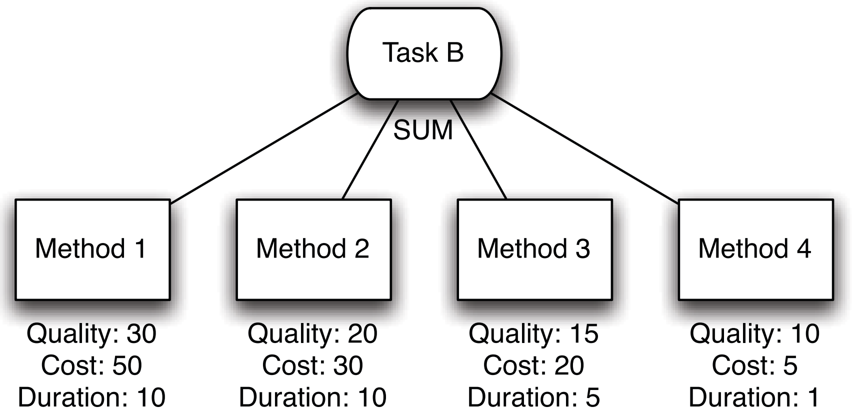


Figure 2.3: Two TÆMS task structures with different CAFs.

of 30 is permissible, then it only makes sense to execute either methods 2 or 3 in order to achieve Task A as both methods 1 and 4 will violate one of the specified criteria. If, however, the same criteria was specified for Task B with a *SUM* CAF[6], then either of the sets {Method 2}, {Method 3} or {Method 3, Method 4} could be scheduled. Note that in the former case, there should be no roles for methods 1 and 4 in the organization, while in the latter case, only method 1 should not have an equivalent role.

Hence, the criteria being optimized can be used to prune the possible set of roles in the organizational structure. This in turn reduces the cost of scheduling and coordination because non-satisfiable tasks have already been removed from the task structure, reducing the number of nodes in the task structure.

3. **The available resources:** Similar to the criteria being optimized, the available resources also constrain both the number of agents and the kind of roles in the organizational structure. The number of agents are constrained because agents need resources (for example processors, memory, disk space, network bandwidth, etc.) to operate — the available resources will hence determine the number of agents that can exist without contending with each other for the scarce resources. The kind of roles are similarly constrained. If some tasks require resources that are unavailable, such tasks cannot be scheduled and it does not make sense to assign roles to such tasks.

   Note that there are two differences between the way the available resources and the criteria being optimized affect the number of agents and the kind of roles in the organizational structure. Firstly, the available resources are

[6] A CAF describes how the quality of a task is computed from the qualities of its subtasks. See Section 3.1 for a formal definition.

hard constraints that cannot be violated (for example, if there are only 5 processors available, there is no way in which an organization that requires 10 processors will work). The criteria being optimized, on the other hand, often impose soft constraints that should be respected as much as possible[7]. Secondly, the available resources are usually constraints that are imposed by the environment rather than by the organization designer or the user of the system. Hence, these constraints are often at odds with what a designer or user wants. These conflicts need to be detected and resolved in the organization as much as possible.

4. **The desired robustness:** Robustness may be defined as the ability of a multiagent system to recover from failures and exceptions. An exception may be defined as a departure from an "ideal" system behavior [Dellarocas & Klein, 2000]. Recovery would then involve the execution of some corrective measures to reinstate the ideal system behavior.

   There are two aspects to robustness in multiagent systems:

   (a) **Robustness in the face of agent failure:** A multiagent system should be able to survive random agent failures, that is, the failure of any single agent or group of agents should not bring the system to its knees. At its best, the system should be able to function without any performance degradation in the face of failures. At its worst, the system should degrade gracefully in proportion to the number of failures.

   (b) **Robustness in the face of task failure:** A task is said to fail when the agent achieves no quality, or quality below a desired threshold, on attempting the task. Tasks may fail for a multitude of reasons such as

[7] For hard-real time systems, some of the criteria being optimized might actually impose hard constraints that cannot be violated.

the lack of resources, the lack of desired preconditions or enablements, lack of satisfiable solutions, etc. An agent should be able to detect when a task has failed (which requires monitoring capabilities), diagnose the cause of failure and resolve the issues that led to the failure.

Both of these types of failures can affect the choice of organizational structure. First let us consider agent failure. There are two approaches commonly used to achieve robustness in such cases:

(a) the *Survivalist Approach* [Marin *et al.* , 2001], which involves replicating domain agents in order to allow the replicas to take over should the original agents fail; and

(b) the *Citizen Approach* [Dellarocas & Klein, 2000], which involves the use of special monitoring agents (called *Sentinel Agents*) in order to detect agent failure and dynamically startup new agents in lieu of the failed ones.

Both of these approaches involve the use extra agents to provide robustness — they differ in the kinds of extra agents used and the method of recovery. These extra "recovery" agents also need to be organized and they will affect the resultant organizational structure. Hence, the original organizational structure needs to be augmented to provide for robustness. For example, consider the citizens approach. The monitoring agents, in this approach, will fulfill monitoring roles and need relationship/communication links between the roles being monitored and the monitoring roles. Also the number of extra agents needed will depend on factors such as the failure rate of the agents, the number of original (domain-specific) roles, the number of roles that can be monitored by a single agent, etc. and any change to any of these factors will result in a change in the organizational structure. Furthermore, the monitoring agents might themselves fail and, hence, "second-level" monitoring agents are needed to monitor the monitors. Hence, the monitoring agents may themselves form their own organizational hierarchy.

Returning to task failures, note that task failures only affect the organizational structure in subtle ways. Often, task failures represent coordination failures and since all the coordination is constrained and determined by the organizational structure, a coordination failure is usually an indication of a sub-optimal organizational structure. For example consider two subtasks B and C of a main task A that are performed by two different agents, Agents 1 and 2. If Task B enables Task C and Task B is the responsibility of Agent 1 and Task C is the responsibility of Agent 2, it is not sufficient for Agent 1 to know the deadline on Task A. It also has to know something about the amount of time needed by Agent 2 to complete Task C and needs to decrease the deadline on Task B by this amount. Task A might fail due to a failure to communicate this information and coordinate the two subtasks. Hence, the agents should carefully monitor task failure and should initiate an organizational change if the number of task failures increases above a certain threshold.

2.2 Literature Review

Organizations have been studied extensively in the management sciences (for example see [Burton & Obel, 1984; Lawrence & Lorsch, 1967; March & Simon, 1993; Robbins, 1989; Scott, 1998]), partly because of their importance and ubiquity and partly because we, as individuals, spend so much of our time in them. It was not long before researchers in the areas of the distributed systems, in general, and multiagent systems, in particular, realized the natural fit between organizations and the systems that they were trying to build. Indeed, Mark Fox offered the following definition of a distributed system:

A distributed system is defined as a particular organization — task decomposition and control regime — resulting from the distribution of a set of tasks over a logically or physically disjoint processing elements.
(Mark Fox in [Fox, 1981])

Whereas the above definition was offered before the emergence of multiagent systems as a field, a lot of multiagent researchers have used organizations and organizational modeling — both as a means for understanding the dynamics of existing human organizations and for building new computer systems based on multiagent systems. Since we are primarily interested in building multiagent systems, we will only discuss research that deals with the latter area here. For a representative sample of research in the area of organizational modeling using multiagent systems see [Carley, 2002b; Chang & Harrington, 2006; Forno & Merlone, 2002; Lant, 1994; Vriend, 2006] and journals like *Handbook of Computational Economics* [Carley & Wallace, n.d.] and *Computational and Mathematical Organization Theory* [Amman *et al.* , 1996].

The organization of the rest of this section is as follows: we will first describe an organizational taxonomy that can be used to classify the various kinds of organizations. We will then discuss some of the research literature relevant to the design of static[8] multiagent organizations in Section 2.2.2. Such organizations have a static organizational structure that does not change at run-time. Then we will discuss most of the work related to adaptive, run-time multiagent organizational design in Section 2.2.3. This includes the subset of work related to Organizational Self-Design (OSD), which most of our research is based on. Finally, we will briefly

[8] By static, we do not mean that the organization does not change at all during the entire duration of its existence. Indeed in most organizations, the agents that make up the organization do change over time, as agents fail or leave the organization and as new agents join the organization. Instead the term static here simply means that the organizational structure remains static and does not change unless explicitly changed by an entity (usually the multiagent designer) that is external to the agent and its environment.

discuss some of the work on *self-organization*, primarily because we have often been asked for the connection between our work and the significant magnitude of work done on self-organization.

2.2.1 An Organizational Taxonomy

In the multiagent literature there is no universally accepted definition of what organizations are, and hence, there is no universally accepted approach towards their design. Instead most researchers come up with their own definition, depending on the characteristics of organizations that they are interested in.

It is, however, possible to classify and characterize the various approaches to organization according to the following criteria [Coutinho *et al.* , 2007; van Elst *et al.* , 2003; Horling & Lesser, 2005b; Vazquez-Salceda *et al.* , 2005]:

- *The meaning of term organization:* Coutinho et. al. [dos Reis Coutinho *et al.* , 2005] differentiate between two different meanings of an organization. According to the first meaning, an organization is an entity with a distinct identity and purpose [Robbins, 1989; Zambonelli *et al.* , 2003]. This meaning might be referred to as the *enterprise view* of an organization and is directly borrowed from the management sciences. According to the second view, an organization is basically *a set of structures or patterns of joint activity that serve to constrain the behavior of the constituent agents* [Singh & Huhns, 2005].

  Note that the distinction between these two meanings is not rigid and clear-cut. Instead most usages of the term organization transcends both of these meanings in the literature. The key distinction between these two meanings is on the emphasis. The first view emphasizes the organization and its goals over those of any individual agent while the second view stresses the constituent agents themselves and uses an organization as a means of constraining their behavior.

- *The type of multiagent system being organized, i.e. whether the multiagent system is open or closed:* A closed multiagent system is one in which (a) all the agents are designed under centralized control (usually by a single person or human organization) and (b) the set of agent types is predefined by the entity that sets up and controls the system [Krupansky, 2005]. Examples of such systems includes most of the cooperative distributed problem solving (CDPS) systems [Decker *et al.* , 1989; Horling & Lesser, 2005a].

  An open multiagent system, on the other hand, is one in which *agents, that are owned by a variety of stake-holders, continuously enter and leave the system* [Huynh, 2006]. In such systems, the agents may be designed by multiple non-interacting entities (persons or organizations) and the set of agents types may not be predetermined before the system is put into effect. An example of such a system would be an electronic market place [Dellarocas & Klein, 1999; Guttman *et al.* , 1998; Wellman, 1997].

  In general open multiagent systems are much more difficult to design than closed ones. This is because the design of the agents is not under centralized control, and hence, there is no way, in general, to guarantee the behavior of any particular agent or even the system as a whole. Some of the issues that have to be dealt with include stability [Lee *et al.* , 1998], security, trust [Huynh, 2006], reputation [Huynh, 2006], social norms [Shoham & Tennenholtz, 1995], etc.

- *The type of agents present in the system, i.e. whether cooperative or competitive:* The constituent agents of an organization may either be benevolent (cooperative) or self-interested (competitive). [Lesser, 1999] describes the difference between the two as follows:

  > Cooperative agents work toward achieving some common goals,

whereas self-interested agents have distinct goals but may still interact to advance their own goals. In the latter case, self-interested agents may, by exchanging favors or currency, coordinate with other agents in order to get those agents to perform activities that assist in the achievement of their own objectives. For example, in a manufacturing setting where agents are responsible for scheduling different aspects of the manufacturing process, agents in the same manufacturing company would behave in a cooperative way while agents representing two separate companies where one company was outsourcing part of its manufacturing process to the other company would behave in a self-interested way.

- *The duration of existence of the organization, i.e. whether short term or long term:* Short term organizations are typically goal-directed, i.e. they are formed with a specific purpose in mind and cease to exist once the purpose for their formation is fulfilled. Examples of short term organizations include coalitions [Klusch & Gerber, 2002; Shehory & Kraus, 1998] and teams [Tambe *et al.* , 1999; Pynadath & Tambe, 2002; Tambe, 1997].

  Most organizations are, however, long-term [So & Durfee, 1993] and are expected to outlive the existence of any of their constituent agents. As argued by Carley, [Carley & Gasser, 1999] *organizations are reasonably long term in duration and have knowledge, culture, memories, history, and capabilities distinct from (that of) any single agent.*.

- *The explicitness of the organizational structure:* Most traditional multiagent systems either focused on the design of individual agents or on the design of specific coordination mechanisms. Such systems often did not have an explicit organizational design or structure — an implicit organization often emerged from the interactions of individual agents and their coordination protocols. For example, most of the systems based on the contract net protocol [Smith & Davis, 1978; Smith, 1980] had an implicit organizational structure that was generated anew for each problem instance. Implicit organizational design is particularly efficient when each problem instance is different from every other problem instance.

Systems that have an explicit organizational design usually have an explicit representation of the goals, the organizational structure and the rules, regulations, policies, procedures and conventions that make up the organization. Some of the advantages of having an explicit organizational structure are: (1) It allows the modeling and evaluation of alternative task and role assignments and their effects on the performance of the organization. (2) It reduces the cost of coordination when problem instances have common task structures or environments by exploiting the commonalities between the task structures to pre-coordinate the activities of the agents.

- *Run-time adaptability of the organizational structure:* Some organizations have a static organization structure that does not change during the lifetime of the organization. Designing such an organization involves modeling the problems being solved and the environmental conditions under which they have to be solved *before* the organization and its agents are ever deployed in the field. The obvious disadvantage of using such a static organizational structure is that any change in the task-structure and/or environmental conditions (for example, available resources) requires a redesign of the organization.

  To overcome this disadvantage researchers have been looking at ways to allow the agents to dynamically alter their organizational structures at run-time. This is especially true for situations in which the problems being solved and/or the environmental conditions are likely to change significantly during the lifetime of the organization. Such an adaptive run-time approach to organization design requires the agents to have a means of representing their organizational knowledge and reasoning about it. Such approaches form the core of our research and we will be discussing the existing literature in this area in Section

2.2.3.

- *The structural form/type of the organizational structure:* If the roles of an organization form the nodes of a graph and the relationships between the nodes form the edges of the same, then this criteria attempts to classify the agents based on the type/structure of the resultant graph. For example, if the graph representing an organization forms a rooted tree with the root of the tree being the role of the decision maker, then this organization can be considered to be a hierarchy. Similarly if all the nodes of the graph are inter-connected with each other, the organization has a flat structure.

  This categorization corresponds to the *Structural Dimension* of modeling organizations due to [Coutinho *et al.* , 2007].

Both Horling and Lesser [Horling & Lesser, 2005b] and Coutinho et. al [Coutinho *et al.* , 2007] present alternative schemes for classifying organizations. The former scheme is based on what the authors refer to as the organizational paradigms, which attempt to categorize organizations primarily based on both the structural form of the organization and the functional decomposition of their constituent agents. The latter scheme looks at organization as having four dimensions — (1) the *structural dimension*, which deals with roles and relationships between the roles; (2) the *functional dimension*, which deals with goal/task decomposition; (3) the *dialogical dimension* that deals with how information is communicated and includes the ontology, communication language, and knowledge representational aspects of the organization; and (4) *deontic dimension*, which deals with permissions, obligations, norms, rules, etc. We don't focus much on the dialogical dimension as this dimension is more relevant to open organizations. See Section 3.4.3 for a description of the deontic dimension.

2.2.2 Static Compile-time Organizational Design

In this section we will discuss three representative research works (out of the many) that deal with static compile-time organizational design — AGR, OMNI and ODML. The first, AGR, is a qualitative approach to designing static organizations that emphasizes the concept of roles and relationships as described in this chapter. The second, OMNI, is another qualitative approach to designing static organizations that focuses on open multiagent systems. Finally, ODML is a quantitative approach to organizational design that attempts to come up with near-optimal organizations given a set of tasks and environmental conditions. These three approaches are very briefly described below:

- **AGR: Agents, Groups and Roles** [Ferber *et al.* , 2005, 2004]. The AGR model is one of the simplest models of a organization. In this model, organizations are represented using three organizational constructs: Agents, Groups and Roles. Basically the agents enact one or more roles. One or more roles are aggregated together to form groups and groups may be further aggregated to form other groups. The groups are simply a way of partitioning an organization so as to constrain the amount of communication that takes place within an organization since communication only takes place between the agents that are part of the same group.

  Hence, AGR uses the concepts of Agents, Roles and Groups to define organizational structures that match our notion of an organization as described above.

- **OMNI: Organizational Model for Normative Institutions** [Dignum *et al.* , 2005; Vazquez-Salceda *et al.* , 2005]. OMNI is an approach to organizational design that is best suited to the design of open systems, especially ones in which the organizational designer is not the same as the agent designer. It can, however, be used for closed systems as well.

In OMNI, organizations are viewed as having three dimensions:

1. the *normative dimension* defines the rules or norms of the organization;
2. the *organizational dimension* defines the roles and the relationships (interaction patterns) between the roles; and
3. the *ontological dimension* describes a controlled vocabulary that contains objects and the relationships between them. This vocabulary is used by the agents when communicating with each other.

Furthermore, each dimension is modeled at three levels of abstraction: the abstract level, the concrete level and the implementation level. The idea behind the three levels is that as the designer moves from a more abstract level to a more concrete level, he moves from a broad representation of the goals of the organization to an implementation of the organization. In the interests of brevity, we will not go into all the nitty-gritty details of all the three levels but refer the reader to the referenced papers.

- **ODML: the Organizational Design Modeling Language** [Horling & Lesser, 2005c; Horling, 2006]. ODML takes a very different and quantitative approach to the design of an organization. ODML tries to predict how the use of different organizational structures and constructs might affect the various organizational characteristics such as scalability, reliability, speed, and/or efficiency.

  In addition to being quantitative, the ODML language is also unique in that it is almost devoid of most traditional organizational concepts such as agents, roles, relationships, etc. Instead the central construct in ODML is a node, which has (a) a user-defined type (which is nothing but a symbol), (b) a set of *is-a* and *has-a* relationships with other nodes[9] and (c) a set of quantitative fields. These quantitative fields describe the variables, constants, modifiers and constraints that affect the node.

There is also a special node that represents the whole organization. The set of nodes collectively describe a high-level representation or organizational template of the set of possible organizations. These organizational templates can then be *instantiated* to form actual organizations by selecting values for the elementary variables in a way that satisfies all the specified constraints. The quantitative characteristics of an organization can be determined from the expressions that describe the special organizational node.

An obvious question is how can ODML be used to represent high-level organizational constructs like roles and agents? This question is answered by the authors as follows:

> Instead of directly incorporating the usual high-level organizational components, such as hierarchies, roles, agents, etc., ODML provides a set of relatively low-level primitives by which such structures can be defined. For example, a node with the user-defined type manager, having a has-a relationship with another node of type agent could embody a role-agent relationship. A sequence of has-a relationships between nodes could indicate a hierarchy. Although the high-level semantics for these nodes may only be implicit, the concrete characteristics and design ramifications are still directly and quantitatively captured by the nodes' fields. We feel that this approach can lead to an increased diversity of representable structures, by avoiding the assumptions and inevitable restrictions that can accompany frameworks with higher-level semantics.
>
> (Horling and Lesser [Horling & Lesser, 2005c])

Hence, ODML is a particularly nice way of representing the static design of a organization in way that allows the designer to determine the characteristics

9 These relationships are equivalent to standard *is-a* and *has-a* relationships between classes in any object oriented programming language.

of the organization once deployed. Therefore it can be used for worth-oriented domains. However, ODML is not directly suited to open systems and needs to be extended to handle systems that have dynamic environments.

2.2.3 Adaptive Run-time Organizational Design

As mentioned previously, agents that exhibit an adaptive organizational design can change their organizational structure at run-time in order to cater to changes in the task structure and environment. Such approaches to organization come under the umbrella term Organizational Self-Design or OSD.

The concept of OSD is not new and has been around since the work of Corkill and Lesser on the DVMT system [Corkill & Lesser, 1983], even though they did not fully develop the concept. More recently Dignum et. al. [Dignum *et al.* , 2004] have described OSD in the context of the reorganization of agent societies and attempt to classify the various kinds of reorganization possible according to the the reason for reorganization, the type of reorganization and who is responsible for the reorganization decision. According to their scheme, the type of reorganization done by our agents falls into the category of structural changes and the reorganization decision can be described as shared command. Also, in our agents, the reason for reorganization can either be protective or corrective.

Our research primarily builds on the work done by Gasser and Ishida [Gasser & Ishida, 1991; Ishida *et al.* , 1992], in which they use OSD in the context of a production system in order to perform adaptive work allocation and load balancing. In their approach, they define two organizational primitives – composition and decomposition, which are similar to our organizational primitives for agent spawning and composition. The main difference between their work and our work is that we use TÆMS as the underlying representation for our problems, which allows, firstly, the representation of a larger, more general class of problems and, secondly, quantitative reasoning over the task structures. The latter also allows us to incorporate different design-to-criteria schedulers [Wagner & Lesser, 2000]. Furthermore, in addition to load balancing and task allocation, our approach also attempts to coordinate the activities of the individual agents by allowing for various coordination mechanisms.

Some other approaches to OSD include:

- **Horling et. al.'s work on using self-diagnosis to adapt organizational structures** [Horling *et al.* , 2001]: This work presents a different, top-down approach to OSD that also uses TÆMS as the underlying representation. However, their approach assumes a fixed number of agents with designated (and fixed) roles. OSD is used in their work to change the interaction patterns between the agents and results in the agents using different subtasks or different resources to achieve their goals.

- **Sycara et. al.'s work on agent cloning** [Decker *et al.* , 1997; Shehory *et al.* , 1998]: This is another approach to resource allocation and load balancing. In this approach, the authors present agent cloning as a possible response to agent overload – if an agent detects that it is overloaded and that there are spare (unused) resources in the system, the agent clones itself and gives its clone some part of its task load. The cloned agents are perfect replicas of the original agents and fulfill the same roles and responsibilities as the original agents.

  Our OSD approach incorporates agent cloning as one of the general agent spawning strategies. The other general agent spawning strategy is agent breakup in which the spawned agents are specialized on a subpart of the spawning agent's task structure, which is no longer the responsibility of the spawning agent. That is, in agent spawning, the original task being handled by the agent doing the spawning is broken up into two or more subtasks which are then the responsibility of two different agents. In our OSD research, we have tried to study the tradeoffs between these two strategies and have developed a hybrid strategy that uses both agent cloning and agent breakup. (See Section 3.4.5.2 for details.)

Also our OSD approach deals with the explicit formation and adaption of the organization and with the coordination of the agents' tasks which are not handled by this work.

- **DeLoach et. al.'s work on The Organization Model for Adaptive Computational Systems (OMACS)** [DeLoach *et al.* , 2008; DeLoach, 2009]: The OMACS approach defines an organization in terms of the goals of the organization, the roles needed to achieve those goals, the constituent agents that make up the organization and the capabilities of the constituent agents. The OMACS approach also describes a set of relations/functions (achieves, requires, possesses, etc) that can be used to map agents to roles and goals so as to maximize an organization assignment function (OAF).

  The primary difference between the OMACS approach and our OSD approach is the primary focus of the work - The OMACS approach emphasizes the agents that make up the organization and tries to find a good allocation (one that maximizes the OAF function) of agents to the roles and goals of the organization. Our OSD approach, on the other hand, emphasizes the problems or goals that the organization is trying to achieve. The idea behind our approach is that the multiagent designer is primarily concerned with the set of problems that he/she wishes to solve and should not have to worry about the details of the organization that solves those problems. The OMACS approach is more suitable when there are a fixed set of pre-existing agents in the multiagent system whose capabilities are known. Our OSD approach is more geared towards real-world applications in grid/volunteer/cloud computing where a user has a workflow that he/she wishes to enact and OSD can be used for the purposes of task/resource allocation. Our OSD approach also focuses on the mechanism by which the reorganization occurs and we have presented a distributed algorithm that can be used for the task/resource allocation. The OMACS approach allows various reorganization algorithms to be used though most of them are centralized. Another difference between the OMACS approach and the OSD approach is that the OMACS approach is more similar to a one-off task/resource allocation scheme like the contract net protocol in that it treats every problem instance as separate and creates a different organization for each.

There are, however, similarities between the OMACS approach and our OSD approach. The TÆMS task representation language is very general and can be used to represent many of the concepts in the OMACS approach. For example, the goals in OMACS are Tasks in TÆMS ; the roles are equivalent to the executable methods; and the concept of roles requiring capabilities can be represented by the mapping of executable methods to the set of resources needed for executing the method. Finally, the assignment of agents to roles and goals is done in our approach by "creating" agents that have allocated the desired set of resources on the grid/cloud.

- **Kota et. al.'s work on self-organization** [Kota *et al.* , 2008, 2009]: In this work, an organization is defined by the relationships between the agents. An agent, A, may be related to another agent, B, in three different ways – (i) Agent A may be an acquaintance of Agent B, meaning that Agent A knows about the presence of Agent B but does not have any interactions with it; (ii) Agent A may be a peer of Agent B, which means that the interactions between the agents are limited in number; and (iii) Agent A may be a superior of Agent B, which means that the agents will have a high level of interaction between them. These relationships are used to allocate tasks amongst the agents, with agents preferring to perform tasks on their own if they have the requisite capacity. If not, they will first try to delegate tasks to their subordinates and then to their peers. Reorganization is used to change these relationships at run-time so as to increase the efficiency and performance of the organization.

Some key differences between their work and our OSD work are: (i) Their work is limited to task-oriented domains in which tasks form a tree structure consisting of subtasks, called service instances (SIs), which can be further divided into other SIs. This task structure is used to enforce a partial ordering on the SIs. Our work, on the other hand, is designed for worth-oriented domains. (ii) Their work uses a one-off task allocation scheme similar to the contract net protocol and OMACS – each arriving task is independently allocated of all the other tasks. The set of relationships are used to limit the agents considered for allocation decision and constrain the coordination protocol. Our OSD work, on the other hand, has explicit roles with multiple instances of a task structure being allocated in the same way. (iii) Their tasks don't have deadlines. They have to be executed in the order in which they were generated.

While there isn't a direct mapping from their work onto our OSD work, it is possible to (i) represent their tasks structures using TÆMS ; and (ii) achieve their form of reorganization through the use of task cloning while spawning a new agent.

- **Barber and Martin's work on Dynamic Reorganization of Decision-Making Groups** [Barber & Martin, 2001; Martin & Barber, 2006]: This work focuses on reorganization in which the decision making control and authority-over relationships amongst the agents are allowed to change during system operation, at run-time. That is, the agents are able to reorganize decision-making groups by dynamically changing (1) who makes the decisions for a particular goal and (2) who must carry out these decisions. However, the application specific resources, task responsibilities, and capabilities of the agents remain the same, and hence, this work assumes a fixed number of homogeneous agents.

Specifically, the authors assume that there are three decision making controls and styles: (1) *Command Driven*, where a master agent gives commands to one or more slave agents to do something; (2) *True Consensus*, in which all agents work together and decide what to do; and (3) *Locally Autonomous/Master*, in which an agent is solely responsible for making its own decisions. The authors allow the decision making style to vary at run-time and argue that the optimal style depends on the task and environmental conditions at hand.

Whereas there are fundamental differences between their work and our own OSD work (for example, the assumption about a fixed number of agents), there are also similarities. For example, our use of managers to coordinate high level tasks is similar to the use of master control with the manages being run in command driven mode. That is, the manager acts as a master agent for a high-level node and decides which of the subtasks/sub-nodes to execute. It then gives commands to the agents responsible for the sub-nodes to execute them. Also some of the tasks are the sole responsibility of one agent in which case the decisions are similar to the locally autonomous approach.

In fact, this work can be used complement our own OSD work — Since we allow a lot of flexibility in the choice of the coordination mechanisms being used to coordinate the inter-task relationships between our agents, we can use their approach to change the coordination relationship at run-time.

- **Goldman and Rosenschein's work on evolving an organization of agents** [Goldman & Rosenschein, 1997]: This work presents an evolutionary method of generating an organization based on Conway's Game of Life [Gardner, 1985] and applies it to the information domain. In this domain, the agents try to divide a set of documents amongst a set of agents such that each agent ends up with a group of similar documents.

This work is similar to ours in that (1) it does not assume a fixed set of agents, (2) they use agent spawning when an agent holds too many documents and (3) agents may die and release their documents, which is similar to agent composition in our approach.

However, this work cannot directly be applied to worth oriented domains and has not been formally specified. In fact, it is difficult to see how their method can be generalized to a broad set of problems.

Other older approaches to OSD include the work of So and Durfee [So & Durfee, 1993], who describe a *top-down model of OSD in the context of Cooperative Distributive Problem Solving (CDPS)*. Still other researchers have focused on dynamic organizations in open systems and for modeling human organizations. For example, both [Artikis *et al.* , 2009] and [Boella *et al.* , 2009] primarily deal with norms in open systems and describe how the agents might change their own rules of behavior. Since our approach does not deal with open systems, we do not discuss the evolution of norms in our approach (although using different coordination algorithms can capture some of the ideas presented in these works). Similarly, [Lamieri & Mangalagiu, 2009] primarily deals with using agent based modeling to study human organizations. Our approach, on the other hand, is solely concerned with software organizations.

2.2.4 Self-Organizing Multiagent Systems

The term *Self-organizing system*[10] [Bonabeau *et al.* , 1999; Bernon *et al.* , 2006; Parunak, 1997; Serugendo *et al.* , 2006b] is generally used to refer to systems in which intelligence, organization and complex behavior *emerges* from the interactions of relatively simple agents. Such systems have the following characteristics:

- The agents themselves are relatively simplistic in that they possess very limited intelligence and reasoning capabilities. Whereas most, if not all, multiagent systems are considered to have agents that are limited in their capabilities for acquiring knowledge, reasoning and taking decisions (a concept that is commonly known as bounded rationality [Simon, 1957, 1991; Fox, 1979]), the agents being considered in self-organizing systems are especially constrained. The agents in such systems do not (usually) have the ability to reason about their goals, their actions, the goals of other agents or their organizational structures. Instead such agents typically follow a very simple protocol or algorithm to guide their actions.

- The agents in most of these systems do not interact or communicate directly with each other. Instead they interact indirectly through the environment — that is, the agents change their environment in subtle ways that may be detected by some of the other agents. Moreover the agents are spatially distributed throughout the environment. This means that any one "spatial location" will have only a few agents occupying it, which will limit interaction between the agents. This is because the agents can only affect a small area of the environment in which they are located and can only sense the spatial region in which they are located.

[10] Note that we will drop the multiagent prefix in this discussion. It should be obvious that we are dealing specifically with self organizing multiagent systems and not any other self-organizing systems.

- The complex behavior of the system *emerges* from the interactions of the constituent agents. By emergence, we mean that higher system-level behavior develops from elementary behavior that is not explicitly represented in agents themselves. Usually this emergent behavior is considerably more complex than that would be indicated by the sum of the behavior of the individual agents.

- There is no centralized authority, either internal or external to the system, that controls or guides the behavior of the system.

Examples of such systems are abound in nature. For example, ants foraging for food, termites building a nest, flocking birds and schools of fishes all exhibit self-organizing behavior [Parunak, 1997]. Such systems have also been used for engineering applications such as school timetabling, flood forecasting, land-use allocation and traffic simulation [Bernon *et al.* , 2006].

The key advantages of self-organizing systems include their scalability and robustness. Such systems are scalable primarily because the agents are spatially distributed and interact with each other only through the environment. Since only a few agents are present in any one location, any one agent will only be interacting with a limited number of other agents. Hence, self-organizing systems can usually scale to tens of thousands of agents. Such systems are robust because (a) there are no central points of failure since control is completely distributed; and (b) it is difficult to game or spoof the system since most of the agents are running a random non-deterministic algorithm in order to effectively explore the state space.

The disadvantage of self-organizing systems is that it is very hard to guarantee desirable or optimal behavior since the system-level behavior emerges from behavior that is not explicitly represented at any level. Hence, it is difficult to predict the exact characteristics (such as performance, efficiency, etc) of the resultant organization. Hence, a number of researchers [Wooldridge & Jennings, 1998;

Moore & Wright, 2003] have argued that emergent behavior should be limited. For example, Wooldridge and Jennings argue the following:

> ... emergent functionality is akin to chaos. In short, the dynamics of multi-agent systems are complex, and can be chaotic. It is often difficult to predict and explain the behavior of even a small number of agents; with larger numbers of agents, attempting to predict and explain the behavior of a system is futile.
>
> (Wooldridge and Jennings [Wooldridge & Jennings, 1998])

Furthermore, it is not completely obvious how such a system can be adapted for use in general-purpose worth-oriented domains. Hence, even though self-organizing systems do have their applications and strengths, we don't think they can be applied to a wide variety of general-purpose domains, such as the ones in which we are interested.

From the above discussion, it should be obvious that there are significant differences between OSD based systems and self-organizing systems[11]. For example, our OSD agents are capable of complex reasoning about their goals, problem domain and organization. Furthermore, our OSD agents interact directly with each other by sending each other messages. There are also a lot of other differences that we have not covered here in the interest of brevity.

2.2.5 Non-organizational approaches to task and resource allocation

There are other non-organizational approaches to task and resource allocation. Most of these approaches are based on some form of a market mechanism. It has been argued that since agents are rationally bounded, they cannot make all the decisions needed to efficiently allocate tasks and resources amongst themselves using central planning. In such situations, decentralization becomes necessary and

[11] Though some researchers do consider OSD to be a subset of self-organizing systems. For example see [Serugendo *et al.* , 2006a].

in such decentralized settings, market mechanisms are a effective way of allocating tasks and resources:

> ... it is the very complexity of the division of labor under modern conditions which makes competition the only method by which such coordination can adequately be brought about...
>
> As decentralization has become necessary because nobody can consciously balance all the considerations bearing on the decisions of so many individuals, the coordination cannot be effected by "conscious control," but by arrangements which convey to each agent the information he must possess in order effectively to adjust his decisions to those of others.
>
> This is precisely what the price system does under competition, and which no other system promises to accomplish. The more complicated the whole, the more dependent we become on that division of knowledge between individuals whose separate efforts are coordinated by the impersonal mechanism for transmitting the relevant information known by us as the price system.
>
> (von Mises and Hayek as reported in [March & Simon, 1993])

In the following subsection, we describe the Contract Net Protocol (CNP), one of the simplest, yet most famous methods of allocating tasks amongst the agents. CNP is based on agents bidding for tasks. A lot of other task and resource allocation mechanisms are loosely based on the contract net protocol. See for example [Fatima & Wooldridge, 2001] and [Kinnebrew *et al.* , 2008].

Note that task allocation in multiagent systems is a special case of resource allocation. The key difference between the two is that tasks are often subdivided into subtasks which often have interdependencies between them. For a general description of some of the issues in multiagent resource allocation, see [Chevaleyre *et al.* , 2006].

2.2.5.1 The Contract Net Protocol

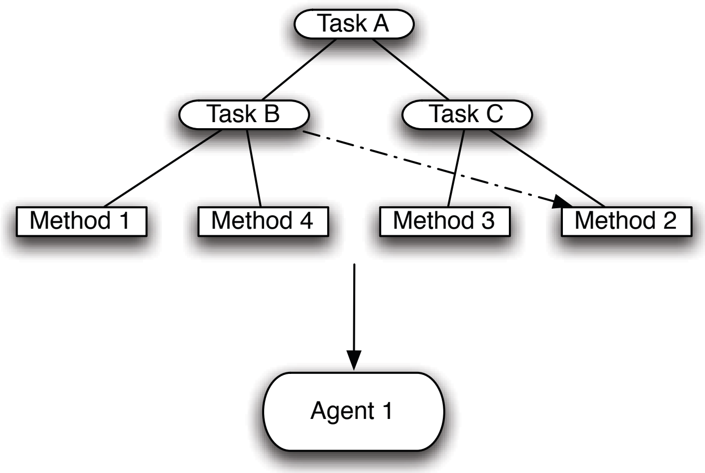
The Contract Net Protocol (CNP) [Smith, 1980, 1988] is a distributed task-allocation protocol in which the agents use negotiation to allocate tasks amongst themselves. In CNP, the agents enact two major roles — *Manager Roles* and *Contractor Roles*. Agents enacting manager roles are responsible for monitoring the execution of tasks and for processing the results of that execution. Agents enacting contractor roles, on the other hand, are responsible for the actual execution of the tasks. Note that agents may be managers and contractors at the same time and there is no *a priori* role allocation/assignment being done — instead the agents can dynamically choose their roles during the course of problem solving.

A typical operation of the CNP is as follows: An agent that has a task to perform checks to see if it can do the complete task by itself. If it can, the agent executes the task with no further communication/coordination with its fellow agents. If not, it acts as a manager agent and breaks up the task into its component subtasks. For each component subtask, the manager agent then sends out a *Request-for-Bids* message to its fellow agents. These agents evaluate this Request-for-Bids message to see if they can perform the subtask. If they can, they respond to the manager agent with *Bid* messages. Upon expiration of the bid deadline, the manager evaluates the bids and sends an *Award* message to the best bidder (based on whatever evaluation criteria that the manager agent used to select the best bid). The best bidder than enacts the contractor role and is responsible for executing the subtask.

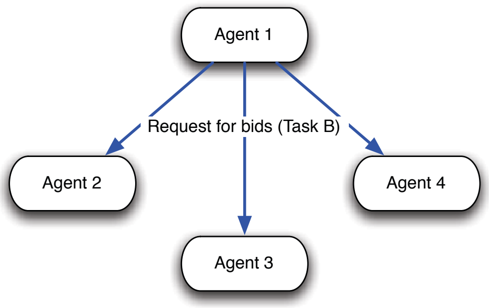
The contractor then runs the same protocol to perform the subtask — that is if it can't complete the allocated subtask on its own, it breaks up the subtask into sub-subtasks and sends out *Request-for-Bids* messages for each of its sub-subtasks. The operation of CNP is shown in Figure 2.4.

Key differences between our OSD approach and the CNP approach are:

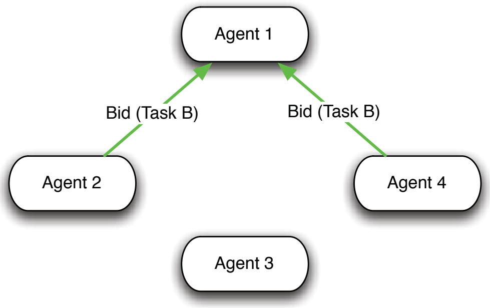
1. The CNP approach is a one-off task allocation scheme in which all the task instances are executed independently of each other. The set of agents used to execute a particular instance of a task has no bearing on any subsequent instances of the same task. In our OSD approach, on the other hand, the



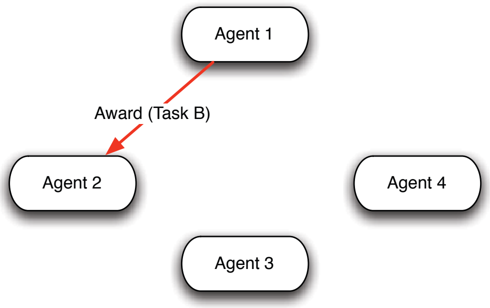
(a) A problem, Task A, with the given task structure arrives at Agent 1



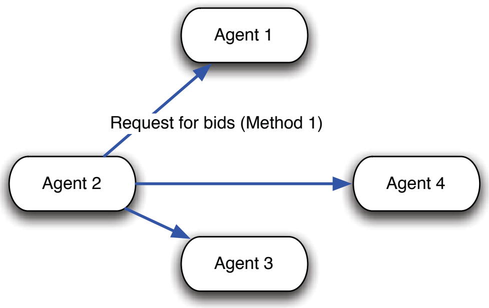
(b) Agent 1 breaks up Task A and sends *Request-for-Bids* for Task B to the all the other agents



(c) Agents 2 and 4 respond to Agent 1 with bids for Task B



(d) Agent 1 awards the contract for Task B to Agent 2



(e) Agent 2 breaks-up the task into its methods and sends *Request-for-Bids* to the other agents

Figure 2.4: Operation of The Contract Net Protocol

task allocation is determined by the roles of the agents and the cost of task allocation is amortized over all the task instances.

2. In the CNP approach, the organizational structure is implicit in the way in which the tasks are allocated. Furthermore, the organizational structure is regenerated for each task instance. Our OSD method uses an explicit organizational structure.

2.3 Chapter Summary

This chapter introduced the reader to the term "organizations" and reviewed the existing literature on the design of organizations.

Section 2.1 described what an organization is and Section 2.1.1 discussed the factors that influence the choice of an organizational structure.

Section 2.2 discusses the existing literature relevant to our work — Section 2.2.1 describes an organizational taxonomy that can be used to classify and characterize the various approaches to organizational design. In Section 2.2.2, we saw three representative approaches to the static compile-time design of an organizational structure. This was followed by a discussion of the existing research on adaptive run-time organizational design in Section 2.2.3. This research forms the background and basis of this dissertation. Next, Section 2.2.4 describes self-organizing systems and discusses the ways in which such systems differ from the kind of systems that we are interested in. Finally 2.2.5 discusses some non-organizational approaches to task and resource allocation.

Chapter 3

APPROACH

Every vital organization owes its birth and life to an exciting and daring idea. (James Bryant Conant)

Human beings, viewed as behaving systems, are quite simple. The apparent complexity of our behavior over time is largely a reflection of the complexity of the environment in which we find ourselves.
(Herbert Simon)

The organization of this chapter is as follows: We start with an introduction to TÆMS and task structures. We follow this up with a brief description of our approach (in Section 3.2) and a formal definition of our task and resource model (in Section 3.3). We then cover the many facets of Organizational Self Design (OSD) in Section 3.4. Finally, we discuss the implemented architecture of our agents in Section 3.5.

3.1 Task Structures and TÆMS

The first step in trying to solve any complex problem is to analyze the problem and break it up into its constituent sub-problems. The sub-problems can then be solved and the results of solving these sub-problems can be combined into a solution to the original problem. Often the sub-problems themselves are too complex to be solved and need to be further decomposed. This recursive decomposition of problems can continue to any arbitrary level, until each problem is small enough and constrained enough to be solvable on its own. This process is known as task

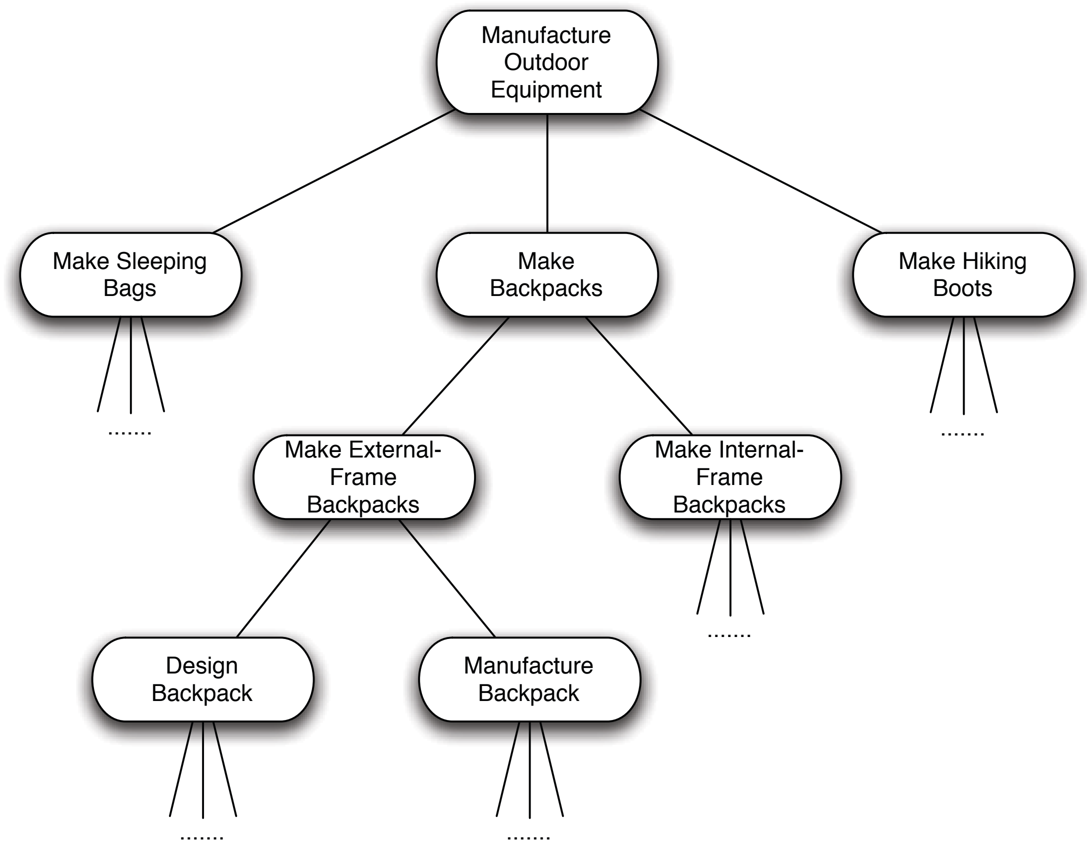


Figure 3.1: A simple task decomposition tree for the manufacturing domain. Each node of the tree represents a goal. The children of a node represent the sub-goals of that goal.

decomposition and results in a task decomposition tree, similar to the one shown in Figure 3.1. This is the principle on which most of traditional AI planning [Ghallab *et al.* , 2004; Rich & Knight, 1991; Russell & Norvig, 2002] is based.

For most complex problems, a simple task decomposition tree is often insufficient and cannot be used to accurately represent and model the wide range of ramifications that occur in real-life problems. Some of the problems that cannot be modeled using a simple task decomposition tree are:

- Many problems have alternative ways of solving them and attempting to solve a problem in a certain way may result in multiple outcomes. For example, a sleeping bag may be filled with down feathers or with a synthetic material. Hence, some mechanism is required to represent the alternatives and reason about them based on quality/cost tradeoffs.

- The sub-problems may interact with each other in arbitrary ways that need to be modeled. For example, some problems may need to be completed before other problems are started. In our running example, a sleeping bag has to be designed and manufactured before it can be shipped.

- The environment[1] in which the agents operate is often complex and directly affects the performance of the agents and their ability to solve the problems being attempted. According to the classification scheme presented in Russell and Norvig[Russell & Norvig, 2002], most multiagent systems have an environment that is continuous, dynamic (the price of down feathers might change), non-deterministic (the attempt to ship some sleeping bags might fail if the shipping company's workers go on strike) and inaccessible (the agents cannot know about all the factors that affect the cost of raw materials).

  Some mechanism is needed to represent some of the effects of a dynamic, non-deterministic and inaccessible environment on the task being attempted. In particular, some means of representing the probability of failing on a task or of a task having multiple outcomes with different characteristics (eg cost and duration) is needed.

For these and other reasons, a better representation scheme is needed to model the complex, real-life problems that the agents are attempting to solve.

[1] The environment is the logical entity of a multiagent system in which the agents and other objects/resources are embedded. This definition is due to Weyns et. al.[Weyns *et al.* , 2004].

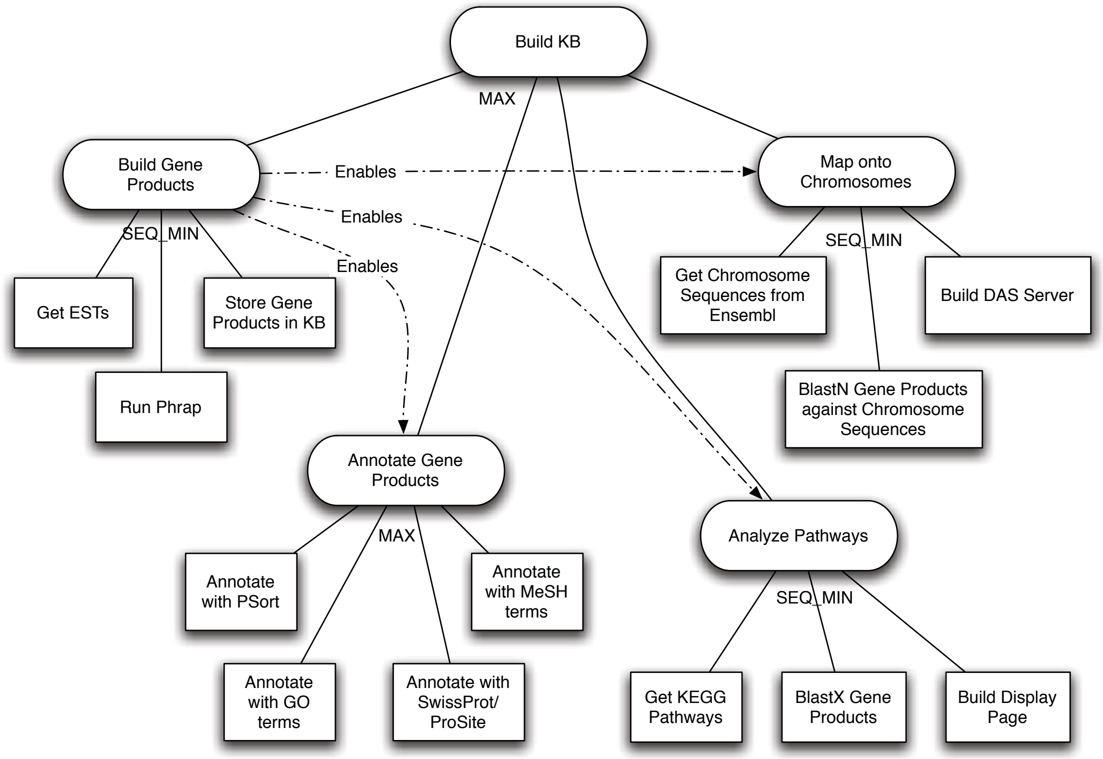


Figure 3.2: An example of a TÆMS task structure.

TÆMS (Task Analysis, Environment Modeling and Simulation) [Decker, 1995, 1996; Lesser *et al.* , 2002; Horling *et al.* , 1999] is one such representational framework that can be used represent and reason about complex task environments. TÆMS has been used in a wide range of applications including distributed sensor networks [Decker, 1995], information gathering [Lesser *et al.* , 2000], hospital scheduling [Decker & Li, 2000], EMS [Chen & Decker, 2005], and military planning [Wagner, 2004; Zimmerman *et al.* , 2007; Maheswaran *et al.* , 2008].

We will be using TÆMS as the underlying representational framework for reasoning about organizational structures. An example of a TÆMS task structure is given in Figure 3.2. Salient features of TÆMS , which make it especially suited for representing a wide range of task structures and environments, are:

- Representation of tasks at various levels of abstraction. In TÆMS , tasks are represented using directed acyclic graphs (DAGs). The root nodes (the ones which don't have any parents) represent high-level tasks or goals. The subnodes of a node represent the subtasks and methods that make up the high level task. The leaf nodes are at the lowest level of abstraction and represent executable methods - the primitive actions that the agent can perform. Hence, high-level goals are divided into subgoals which have to be achieved and the primitive actions are the ways in which the subtasks can be achieved.

- Representation of the multiple outcomes of executable methods along with their quantitative characteristics. Examples of characteristics that we might wish to model are the quality achievable on successfully completing an executable method and cost and time duration needed to perform the method. Furthermore, some methods may have multiple outcomes with different probabilities of occurrence and each of these outcomes may have different quantitative characteristics. TÆMS allows the annotation of executable methods with their outcomes and quantitative characteristics, which provides a way of reasoning about cost/quality tradeoffs.

- Various mechanisms for combining subtasks. In TÆMS , each node of the DAG is labeled with a characteristic accumulation function (CAF, formerly known as a quality accumulation function or QAF) that describes how many subgoals need to be achieved in order to achieve the high-level goal, the method of computing the run-time characteristics of a task based on the run-time characteristics of the subtasks and, in some cases, the order in which the subgoals need to be achieved[2]. For example, all the subtasks and executable

[2] Ideally, CAFs should *not* be used to specify order or sequencing constraints. Task interrelationships should be used for such purposes (see the next bulleted item). However, quite a few researchers have used CAFs to specify ordering,

methods of a node labeled with an MIN CAF need to be achieved in order for the node to accrue quality and the quality of that node will be the minimum of the qualities of all the sub-nodes of that node. Similarly, one and only one subgoal should be completed for a node labeled with an EXACTLY_ONE CAF. CAFs are mainly used to provide a way of selecting between the alternative ways of achieving a task and provide semantics for combining the subtasks of a task.

- The ability to represent interrelationships between the tasks. TÆMS allows the representation of both hard and soft relationships and the quantitative effects of those relationships. An example of a "hard" relationship is *enables.* If some task, A, enables another task, B, then A must be completed before B can begin. An example of a "soft" relationship is *facilitates.* If some task, A, facilitates a task B, then completing A before beginning work on B might cause B to take less time, or cause B to produce a higher-quality result, or both.

For our work, we are assuming that a TÆMS task structure is available for the problem being attempted. In particular, we won't look at the planning problem and will not try to generate TÆMS task structures from basic operators, rules or templates. Instead we are assuming that the problem has been thoroughly analyzed by human or other means and a fully specified TÆMS task structure is provided as input to the multiagent system.

3.2 Our Approach

As stated previously, we take a run-time approach to organizational design in which the agents in the multiagent system are responsible for designing their own

mostly as a short-hand for explicitly enumerated inter-task relationships.

organization. This approach may be referred to as *Organizational Self-Design* or OSD.

We assume that the user of the multiagent system generates problem solving requests for the problems that he/she wishes the system to solve. Each problem has with it, an associated TÆMS task structure[3], desired solution characteristics (for example, minimum solution quality or maximum cost) and a deadline by which the request must be fulfilled.

In our approach, the system starts up with a single agent that is responsible for solving the complete problem. As new problem solving requests arrive, the agent checks to see if it can complete the requests by their given deadlines and in such a way that the solution characteristics are met. If it finds that it is overloaded and cannot complete its requests by their specified deadlines or cannot meet the specified solution characteristics, it spawns off new agents to handle some parts of the problem. This spawning process effectively breaks up the task structure — each agent is now responsible for some subpart of the task structure and the nodes in that subpart represent the roles that that particular agent is enacting. The newly spawned agents may themselves spawn off new agents and assign them some of their roles if they feel that they are overloaded and cannot adequately complete their responsibilities. Note that the spawning process has added agents to the organization and caused a (re-) distribution/allocation of tasks and resources amongst the agents. If an agent is free for an extended period of time, it may decide to combine with another agent to save computational resources. Hence, we propose two organizational primitives that are responsible for generating the organizational structure: spawning and composition.

The spawning process can be likened to that of a human agent that enlists

[3] In an analogy to an OOP systems, the task structures can be equated to classes and the problem instances to the instance objects.

the services of other humans when it finds that the problems that it is trying to solve are beyond the limits of its bounded rationality. The spawning process is then effectively equivalent to the division of labor and specialization that occurs in human organizations. The composition process can be likened to a human agent that resigns from the organization because it isn't sufficiently challenged and working to its full capacity.

We would like to emphasize that the organization is being built in a bottom-up fashion because each agent is individually responsible for its own spawning and composition decisions irrespective of what the other agents are doing. There is no centralized authority or single agent designing the organization.

3.3 Task and Resource Model

To ground our discussion of OSD, We now formally describe our task and resource model. In our model, the primary input to the multi-agent system (MAS) is an ordered set of problem solving requests or task instances, $< P_1, P_2, P_3, ..., P_n >$, where each problem solving request, P_i, can be represented using the tuple $< \mathbf{t_i}, \mathbf{a_i}, \mathbf{d_i} >$. In this scheme, $\mathbf{t_i}$ is the underlying TÆMS task structure, $\mathbf{a_i} \in \mathbf{N}^+$ is the arrival time and $\mathbf{d_i} \in \mathbf{N}^+$ is the deadline of the i^{th} task instance[4]. The task $\mathbf{t_i}$ is not "seen" by the MAS before the time $\mathbf{a_i}$, i.e., the MAS has no prior knowledge about the task $\mathbf{t_i}$ before the arrival time, $\mathbf{a_i}$. In order for the MAS to accrue quality, the task $\mathbf{t_i}$ must be completed before the deadline, $\mathbf{d_i}$.

Furthermore, every underlying task structure, $\mathbf{t_i}$, can be represented using the tuple $< T, \tau, M, Q, E, R, \rho, C >$, where:

- T is the set of tasks. The tasks are non-leaf nodes in a TÆMS task structure and are used to denote goals that the agents must achieve. Tasks have

[4] N is the set of natural numbers including zero and N^+ is the set of positive natural numbers excluding zero.

a characteristic accumulation function (see below) and are themselves composed of other subtasks and/or methods that need to be achieved in order to achieve the goal represented by that task. Formally, each task T_j can be represented using the pair (q_j, s_j), where $q_j \in Q$ and $s_j \subset (T \cup M)$. For our convenience, we define two functions $\textsc{Subtasks}(\text{Task}) : T \rightarrow \mathscr{P}(T \cup M)$ and $\textsc{Supertasks}(\text{TÆMS node}) : T \cup M \rightarrow \mathscr{P}(T)$, that return the subtasks and supertasks of a TÆMS node respectively[5].

- $\tau \in T$, is the root of the task structure, i.e. the highest level goal that the organization is trying to achieve. The quality accrued on a problem is equal to the quality of task τ.

- M is the set executable methods, i.e., $M = \{m_1, m_2, ..., m_n\}$, where each method, m_k, is represented using the outcome distribution, $\{(o_1, p_1), (o_2, p_2), ..., (o_m, p_m)\}$. In the pair (o_l, p_l), o_l is an outcome and p_l is the probability that executing m_k will result in the outcome o_l. Furthermore, each outcome, o_l is represented using the triple (q_l, c_l, d_l), where q_l is the quality distribution, c_l is the cost distribution and d_l is the duration distribution of outcome o_l. Each discrete distribution is itself a set of pairs, $\{(n_1, p_1), (n_2, p_2), ..., (n_n, p_n)\}$, where $p_i \in \Re^+$ is the probability that the outcome will have a quality/cost/duration of $n_l \in N$ depending on the type of distribution and $\sum_{i=1}^{m} p_l = 1$.

- Q is the set of characteristic accumulation functions (CAFs). The CAFs are a set of functions that determine how the characteristics of a task are computed from the characteristics of its subtasks/methods. See Section 3.3.2 for a more thorough description of CAFs and for a formal definition.

[5] $\mathscr{P}$ is the power set of set, i.e., the set of all subsets of a set

- E is the set of (non-local) effects (NLEs) or inter-task relationships (ITRs). A non-local effect (NLE) describes how the execution of a method or the achievement of a task (the source of the NLE) affects the quality, cost and duration characteristics of another method or task (the destination or sink of the NLE). Note that since tasks do not have their own characteristic, per se, but depend on the characteristics of their subtasks, an NLE between a node, A and a task, B is just a short form for a group of equivalent NLEs between the Node A and all the methods that are descendents of Task B. Non-Local Effects are formally defined in Section 3.3.3 below.

- R is the set of resources.

- ρ is a mapping from an executable method and resource to the quantity of that resource needed (by an agent) to schedule/execute that method. That is $\rho(method, resource) : M \cdot R \rightarrow N$.

- C is a mapping from a resource to the cost of that resource, that is $C(resource) : R \rightarrow N^+$

3.3.1 Execution Model

This subsection describes how agents might accrue quality on a task.

In TÆMS tasks are used to represent goals that the agent might try to achieve. However, an agent can't directly perform or execute a task. An agent may only execute a method which are represented by the leaf nodes of a TÆMS task structure.

Since the agents don't directly perform/execute tasks, the characteristics (quality/cost/duration) of a task have to be computed recursively from the characteristics of its subtask. This recursion ends at the methods, which can be executed directly by the agents and which accrue actual quality.

Recall that methods are represented by an outcome distribution, $\{(o_1, p_1), (o_2, p_2), ..., (o_m, p_m)\}$. On completing the execution of a method, the agent will obtain one of these outcomes. Since each outcome is itself a triple of quality, costs and duration probability distributions, executing a method will result in a quality, cost and duration that is sampled from these distributions. Note that in our model, the methods do not accrue quality until they are actually executed.

For our convenience, we define the following functions:

- QualityD, returns the quality distribution of an outcome.
- DurationD, returns the duration distribution of an outcome.
- CostD, returns the cost distribution of an outcome.
- ExecutedP(TÆMS node, time instance) : $T \cup M, N \rightarrow \{0, 1\}$ is a predicate function that returns 1 if a TÆMS node has been "executed" by specified time instance parameter. We define a node as having been executed if it has accrued positive quality by the given time instance.
- Start(TÆMS node, time instance) : $T \cup M, N \rightarrow N^+$ returns the time at which a TÆMS node is started, if it is started before the specified time instance parameter. The value of this function is undefined for a node, $n \in (T \cup M)$ if ExecutedP$(n, t) = 0$. For a method, this function returns the time at which it was started. For a task, this function is defined as follows:

$$\textsc{Start}(T, t) = \min_{T' \in \textsc{Subtasks}(T) \wedge \textsc{ExecutedP}(T', t) = 1} \textsc{Start}(T', t) \tag{3.1}$$

- End(TÆMS node, time instance) : $T \cup M, N \rightarrow N^+$ returns the time at which a TÆMS node is completed, if it is completed before the given time instance parameter. The value of this function is undefined for a node, $n \in (T \cup M)$ if

$\textsc{ExecutedP}(n,t) = 0$. For a method, this function returns the time at which it was completed. For a task, this method is defined as follows:

$$\textsc{End}(T,t) = \max_{T' \in \textsc{Subtasks}(T) \wedge \textsc{ExecutedP}(T',t)=1} \textsc{End}(T',t) \tag{3.2}$$

3.3.2 Characteristic Accumulation Functions

The set of CAFs, Q, can be subdivided into three sets, i.e $Q =< \mathrm{QAF}, \$\mathrm{AF}, \mathrm{DAF} >$. These three sets are defined below:

1. The set of quality accumulation functions (QAFs), which determine how a task accrues quality given the quality accrued by its subtasks. In the following equations, let $\mathrm{Q}(T,t)$ be the quality accrued by task T at time instance t. For our research, we have focused on four QAFs for our purposes:

   - Min: The Min QAF means the quality of a task is the *minimum* of the qualities of its executed subtasks. The Min QAF is often used to indicate an *AND* relationship between a task and its subtasks because to accrue positive quality on a task with a Min QAF, *all* its subtasks have to be executed[6].

   $$\mathrm{Q}_{min}(T,t) = \min_{T' \in \textsc{Subtasks}(T)} \mathrm{Q}(T',t) \tag{3.3}$$

   - Max: The Max QAF denotes that the quality accrued by a task is the *maximum* of the qualities accrued by its subtasks. The Max QAF is usually used to indicate an *OR* relationship, because to accrue positive quality on a task with a Max QAF, *at least one* of the subtasks of a task has to be completed.

   $$\mathrm{Q}_{max}(T,t) = \max_{T' \in \textsc{Subtasks}(T)} \mathrm{Q}(T',t) \tag{3.4}$$

[6] A subtask that hasn't been completed will have a quality of 0 and the min of any positive number and 0 will always be 0.

- SUM: The SUM QAF implies that the quality of a task is the *sum* of the qualities of its executed subtasks. The SUM QAF is used to represent tasks in which the quality accrued monotonically increases with the number of subtasks executed. Formally,

$$\mathrm{Q}_{\sum}(T,t) = \sum_{T' \in \mathrm{SUBTASKS}(T)} \mathrm{Q}(T',t) \tag{3.5}$$

- EXACTLY_ONE: The EXACTLY_ONE QAF is used to denote an *exclusive or* relationship. The quality of a task with an EXACTLY_ONE QAF is equal to the quality of its executed subtask when only one of its subtasks is executed and is zero in all other situations.

$$\mathrm{Q}_{EO}(T,t) = \begin{cases} \sum_{T' \in \mathrm{SUBTASKS}(T)} \mathrm{Q}(T',t) & \Big|\{T' | T' \in \mathrm{SUBTASKS}(T) \\ & \quad \wedge \mathrm{Q}(T',t) > 0\}\Big| = 1 \\ 0 & \mathit{otherwise} \end{cases} \tag{3.6}$$

Note that these are not the only set of possible QAFs and it is very easy to add other QAFs to our approach, including QAFs based on anytime functions. (See [Wagner, 2004] for the set of QAFs used in the DARPA COORDINATORS project.)

2. The set of cost accumulation functions ($AFs), which describe how the cost of achieving a task depends on the costs of achieving its subtasks. Currently, we have only defined a single $AF —SUM. If $\mathrm{C}(T,t)$ is the cost of "performing" task T at time t, then the SUM $AF indicates that the cost of attempting to achieve task T is the *sum* of the costs of trying to achieve all the subtasks of T. That is,

$$\mathrm{C}(T,t) = \sum_{T' \in \mathrm{DESCENDENTS}(T) \wedge T' \in M} \mathrm{C}(T',t) \tag{3.7}$$

Again it's fairly simple to use alternative cost accumulation functions.

3. The set of duration accumulation functions (DAFs), which describe how the duration of a task can be computed from the durations of its subtasks. Again, we have only defined a single duration accumulation function. However, instead of defining this function in terms of the duration of its subtasks, we define this duration in terms of the start and end times of its subtasks. This is because the duration of a task will depend on whether the subtasks are done consecutively or in parallel.

   Hence, the duration accumulation function is defined as:

$$\mathrm{D}(T,t) = \textsc{End}(T,t) - \textsc{Start}(T,t) \tag{3.8}$$

3.3.3 Non-Local Effects

This subsection formally defines a non-local effects (NLEs).

An NLE is defined by the five-tuple $< tp, T_S, T_D, \delta, p >$, where $tp \in \{\textsc{Enables}, \textsc{Disables}, \textsc{Facilitates}, \textsc{Hinders}\}$ is the type of the NLE, $T_S \in (T \cup M)$ is the source of the NLE, $T_D \in (T \cup M)$ is the destination/sink of the NLE, $\delta \in N^+$ is used to denote the delay between the accrual of positive quality at the source and the effect of the NLE on the sink, and p is a set of (potentially empty) parameters that define the characteristics of the NLE. As should be obvious from this definition, we are interested in four types of NLEs — Enables, Disables, Facilitates and Hinders. These NLEs are described and formally defined below[7]. In all of these formal definitions, assume that $Q_{<\mathrm{NLE},T_S,T_D,\delta,p>}(T_D, t)$ is the quality of the

[7] Note that all of the formal definitions of the NLEs assume that the sink of the NLE is a method. If the sink of the NLE is a Task, replace this NLE with a set of equivalent NLEs from the source to all the descendent methods of the sink.

sink after the application of the NLE, and $Q(T_D, t)$ is the quality of the sink in the absence of the NLE[8].

- Enables: The Enables NLE is used to enforce an ordering constraint between two TÆMS nodes — If method A enables method B, then method A has to be completed before method B is started. Formally,

$$Q_{<\text{Enables},T_S,T_D,\delta,()>}(T_D, t) = \begin{cases} Q(T_D, t) & \text{If } \textsc{Start}(T_D, t) - \textsc{End}(T_S, t) >= \delta \\ 0 & \text{otherwise} \end{cases} \tag{3.9}$$

$$C_{<\text{Enables},T_S,T_D,\delta,()>}(T_D, t) = C(T_D, t) \tag{3.10}$$

$$D_{<\text{Enables},T_S,T_D,\delta,()>}(T_D, t) = D(T_D, t) \tag{3.11}$$

- Disables: The Disables NLE is the converse of the Enables NLE — That is, if a method, A disables a method, B, then Method B should be started before Method A ends. Formally,

$$Q_{<\text{Disables},T_S,T_D,\delta,()>}(T_D, t) = \begin{cases} Q(T_D, t) & \text{If } \textsc{Start}(T_D, t) - \textsc{End}(T_S, t) < \delta \\ 0 & \text{otherwise} \end{cases} \tag{3.12}$$

$$C_{<\text{Disables},T_S,T_D,\delta,()>}(T_D, t) = C(T_D, t) \tag{3.13}$$

$$D_{<\text{Disables},T_S,T_D,\delta,()>}(T_D, t) = D(T_D, t) \tag{3.14}$$

[8] In these formal definitions, all the effects of the NLEs are calculated by first determining the quality/cost/duration characteristics of the sink assuming that no NLEs are present. Then the effects of the NLEs are incorporated into the characteristics of the sink. This is the opposite of how the local-scheduler works, since it takes these NLEs into account *before* choreographing either the source or the sink.

Note that both the Enables and the Disables NLEs enforce *hard* constraints on the ordering between the source and sink. If this constraint is violated, the sink accrues a quality of zero.

The Facilitates and Hinders NLEs, on the other hand, apply a *soft* constraint on the sink. Executing the source before the sink has an effect on the characteristics of the sink which is in direct proportion to the ratio of the quality of the source to the maximum possible quality attainable at the source[9]. This ratio is defined as follows:

$$R(T_S, t) = \frac{Q(T_S, t - \delta)}{Q_{\text{Max}}(T_S)} \tag{3.15}$$

where δ is the delay parameter of the NLE and

$$Q_{\text{Max}}(T_S) = \begin{cases} \max_{(o,p)\in\text{OUTCOMES}(T_S)} \left[\max_{(m,q)\in\text{QUALITYD}(o)} m\right] & \text{If } T_S \in M \text{ i.e. the source is a method} \\ \min_{T'_S\in\text{SUBTASKS}(T_S)} Q_{\text{Max}}(T'_S) & \text{If } T_S \in T \wedge \text{CAF}(T_S) = \text{Min} \\ \max_{T'_S\in\text{SUBTASKS}(T_S)} Q_{\text{Max}}(T'_S) & \text{If } T_S \in T \wedge \text{CAF}(T_S) \in \{\text{Max}, \text{Exactly\_One}\} \\ \sum_{T'_S\in\text{SUBTASKS}(T_S)} Q_{\text{Max}}(T'_S) & \text{If } T_S \in T \wedge \text{CAF}(T_S) = \text{Sum} \end{cases} \tag{3.16}$$

9 Another way of saying this would be to say that the sink always accrues some quality, irrespective of whether or not the source accrues quality before it. The quality would simply change as a result of applying the source before it.

- FACILITATES: The FACILITATES NLE, allows a source node to have a facilitation effect on the sink method by either increasing the quality of the sink or by decreasing its cost and/or duration or both. The classic example of a facilitation effect is the relation between sorting and searching an array. Sorting an array is not necessary for searching it. However, sorting an array before searching it allows us to use a binary search, which speeds up the search process. Hence, we can say that sorting an array facilitates the search.

  In TÆMS , the degree of the facilitation effect of the source on the sink depends on (1) the quality accrued by the source node relative to the maximum possible quality attainable by the source (See Equation 3.15); and (2) the input parameter $p = (\phi_q, \phi_c, \phi_d)$. Here, ϕ_q, ϕ_c, ϕ_d are constants between 0 and 1 and represent the quality, cost and duration powers respectively. Specifically,

$$\text{If } p = (\phi_q, \phi_c, \phi_d)$$

$$Q_{<\text{FACILITATES},T_S,T_D,\delta,p)>}(T_D,t) = Q(T_D,t) \cdot [1 + \phi_q R(T_S,t)] \tag{3.17}$$

$$C_{<\text{FACILITATES},T_S,T_D,\delta,p>}(T_D,t) = C(T_D,t) \cdot [1 - \phi_c R(T_S,t)] \tag{3.18}$$

$$D_{<\text{FACILITATES},T_S,T_D,\delta,p>}(T_D,t) = D(T_D,t) \cdot [1 - \phi_d R(T_S,t)] \tag{3.19}$$

- HINDERS: The HINDERS NLE is the converse of the FACILITATES NLE, i.e. if a source node hinders a sink method, it decreases the quality accrued by the sink method and/or increases its cost and/or duration. Formally,

$$\text{If } p = (\phi_q, \phi_c, \phi_d)$$

$$Q_{<\text{FACILITATES},T_S,T_D,\delta,p>}(T_D,t) = Q(T_D,t) \cdot [1 - \phi_q R(T_S,t)] \tag{3.20}$$

$$C_{<\text{FACILITATES},T_S,T_D,\delta,p>}(T_D,t) = C(T_D,t) \cdot [1 + \phi_c R(T_S,t)] \tag{3.21}$$

$$D_{<\text{FACILITATES},T_S,T_D,\delta,p>}(T_D,t) = D(T_D,t) \cdot [1 + \phi_d R(T_S,t)] \tag{3.22}$$

3.3.4 Extensions to TÆMS

Generally, TÆMS task structures can be thought of as being the output of a planning process [Ghallab *et al.* , 2004]. This is because TÆMS task structures are basically high-level plans for achieving some goal, in which the steps required for achieving the goal — as well as the possible contingency situations — have been pre-computed offline and represented in the task structure.

Most scheduling [Wagner & Lesser, 2000] and organizational research [Carley, 2002a] assumes that the task structure is provided as an input to the system. Since TÆMS task structures are used to represent many contingencies, alternatives, uncertain characteristics and run-time flexible choices, the process of organizing can be thought of as the process of performing long-term task and resource allocation.

For a number of applications, for example protein-ligand docking, generating a complete task structure offline (at design time) and providing it as input to the system is often not feasible. This is because the number and type of nodes in the task structure might depend on the result of some execution that has to be done at run-time.[10]

For example, in protein-ligand docking, the number of docking configurations or attempts, and hence the number of docking tasks in the TÆMS task structure, depends on the specific protein-ligand pair provided as input to the system. Furthermore factors such as the protein/ligand flexibility and the presence of metals can influence the docking mechanism used.

Hence, for this application, we needed some way of generating TÆMS nodes at run-time while constraining and guiding the kind of nodes that can be generated,

[10] Note that it might still be possible to represent the complete task structure, *a priori*, by having the task structure contain the maximum number of nodes possible. However, this would often result in an exponential increase in the size of the task structure.

without having to resort to an expensive general purpose planner or plan-repair algorithm.

To this end, we extended the TÆMS task representation language by adding *Template Nodes*, $\int = (tp, T_C, T_E, q)$, where $tp \in \{\textsc{Iteration}, \textsc{Selection}\}$, is the type of the template node; $T_C \in (T \cup M)$ is a *control node*, which constrains the expansion of this template node; $T_E \subset (T \cup M \cup \int)$ represents the *expansion* of this node. In the case of an ITERATION template node, the expansion node is "generated" a fixed number of times depending on the results of executing T_C and $|\textsc{Subtasks}(T_E)| = 1$. In the case of a SELECTION node, one of the subtasks of T_E is selected as the expansion of this node. Finally, $q \in Q$ is a CAF with similar semantics to the CAF of a task.

3.3.5 Assumptions

We also make the following set of assumptions in our research:

1. The agents in the MAS are drawn from the infinite set $A = \{a_1, a_2, a_3, ...\}$. That is, we do not assume a fixed set of agents — instead agents are created (spawned) and destroyed (combined) as needed. This does not imply that there is not in fact a cost associated with each agent. We can easily model such costs in our approach and our approach will try to minimize these costs.

2. All problem solving requests have the same underlying task structure, i.e. $\exists \mathbf{t} \forall_\mathbf{i} \mathbf{t_i} = \mathbf{t}$, where $\mathbf{t}$ is the task structure of the problem that the MAS is trying to solve. We believe that this assumption holds for many of the practical problems that we have in mind because TÆMS task structures are basically high-level plans for achieving some goal in which the steps required for achieving the goal—as well as the possible contingency situations—have been precomputed offline and represented in the task structure. Because it represents

many contingencies, alternatives, uncertain characteristics and run-time flexible choices, “the same underlying task structure” can play out very differently across specific instances.

3. All resources are exclusive, i.e., only one agent may use a resource at any given time. Furthermore, we assume that each agent has to “own” the set of resources that it needs — even though the resource ownership can change during the evolution of the organization.

4. All resources are non-consumable.

3.4 Organizational Self Design

3.4.1 Agent Roles and Relationships

The organizational structure is primarily composed of roles and the relationships between the roles. One or more agents may enact a particular role and one or more roles must be enacted by every agent. The roles may be thought of as the parts played by the agents enacting the roles in the solution to the problem and reflect the long-term commitments made by the agents in question to a certain course of action (that includes task responsibility, authority, and mechanisms for coordination). The relationships between the roles are the coordination relationships that exist between the subparts of a problem.

In our approach, the organizational design is directly contingent on the task structure of the problems being solved (the *global task structure*) and the environmental conditions under which the problems need to be solved. Here, the environmental conditions refer to such attributes as the task arrival rate, the task deadlines and the available resources.

To form or adapt their organizational structure, the agents use two organizational primitives: agent spawning and composition. These two primitives result in a change in the assignment of roles to the agents. Agent spawning is the generation of a new agent to handle a subset of the roles of the spawning agent. Agent composition, on the other hand, is orthogonal to agent spawning and involves the merging of two or more agents together[11] — the combined agent is responsible for enacting all the roles of the agents being merged. Hence, OSD can be thought of as a search in the space of all the role assignments for a suitable role assignment that minimizes or maximizes a performance measure.

In order to participate in the organization, and to apply these primitives, the agents need to explicitly represent and reason about the role assignments and must maintain some organizational knowledge. This knowledge is represented in each agent using a TÆMS task structure, called the *local task structure*. Hence, we define a role as a local task structure. These local task structures are obtained by rewriting the global task structure and represent the local task view of the agent vis-a-vis its role in the organization and its relationship to other agents. Hence, all reorganization involves rewriting of the global task structure. However, note that the global task structure is *NOT* stored in any one agent, i.e. no single agent has a global view of the complete organization. Instead each agent's organizational knowledge is limited to the tasks that it must perform and the other agents that it must coordinate with — it is this information that is represented using the local task structures.

To allow the agents to store information about other agents in the task structure, we augment the basic TÆMS task representation language presented above by adding organizational nodes ($\mathcal{O}$). Like TÆMS nodes, organizational nodes come in two flavors (i.e. $\mathcal{O} = (T_{\mathcal{O}} \cup M_{\mathcal{O}})$: (a) organizational tasks, ($T_{\mathcal{O}}$), which are used to aggregate other organizational nodes; and (b) organizational methods, ($M_{\mathcal{O}}$), that

[11] Note that any two agents can be merged together — that is, there is no requirement for merged agents to be spawned off from the same agent.

are used to represent either organizational knowledge or organizational actions that have some fixed semantics. To differentiate organizational nodes from "regular" TÆMS nodes (i.e. nodes that are in $T \cup M$), we will refer to non-organizational nodes as *domain nodes* (denoted as $\mathcal{D}$). We define the following organizational nodes:

1. Container-Nodes: $\Sigma \subseteq T_{\mathcal{O}}$[12], are aggregates of domain nodes and other organizational nodes. Formally, $\Sigma = \{\sigma_1, \sigma_2, ..., \sigma_n\}$, where each $\sigma_i =< t_i, s_i >$. In this context $t_i \in \{\textsc{Root}, \textsc{Clone}, \textsc{Coordination}\}$ is the type of the container and determines its purpose; and $s_i \subset (\mathcal{D} \cup \mathcal{O})$ is the set of subtasks/nodes in that container.

2. Non-Local-Nodes: $\Diamond \subset M_{\mathcal{O}}$, are used to represent a domain node in some *other* agent's local task structure. Non-Local-Nodes are used to represent nodes in the global task structure that the agent knows the identity (label) of but does *not* know the characteristics (e.g. quality, cost duration) of[13]. Formally, $\Diamond = \{\diamond_1, \diamond_2, ..., \diamond_n\}$; each $\diamond_i$ can be represented using a set consisting of a single element, $\eta \in \{\textsc{Label}(d) \mid d \in \mathcal{D}\}$ that encapsulates the identity of an existing domain node.

3. Clone Selectors: $\mathcal{S}_{\mathcal{C}} \subset M_{\mathcal{O}}$ are used to select amongst the clones of a node. The purpose of a selector node within a clone-container is to *enable* one or more of the clones, so that the enabled nodes can be "executed" by their agents owning those clones. See Section 3.4.1 for a more detailed formal description.

[12] Currently, $\Sigma = T_{\mathcal{O}}$, that is, the only type of organizational tasks that have been defined are container nodes. However, we might need to add other organizational tasks in the future.

[13] At least initially at the time of breakup. It can however learn these characteristics through some coordination mechanism.

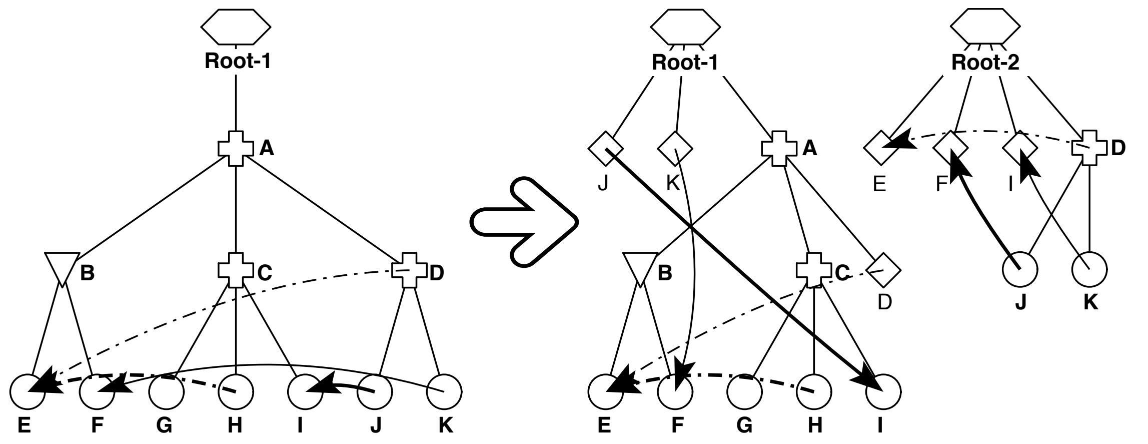


Figure 3.3: The *Breakup* Task Rewriting Primitive. This figure shows the Breakup of Root-1 at Node D. The ◇ nodes represent non-local nodes, that are the responsibility of some other agent.

4. NLE-Inheritors: $\mathcal{N} \subset M_{\mathcal{O}}$, are methods whose sole purpose is to transfer the non-local effect from a non-cloned node to a cloned node or vice versa. See Section 3.4.1 for the rationale behind these node.

To allow for a change in an agent's organizational knowledge, we define four rewriting operations on a local task structure, which are described below. However, before any of these rewriting operators can be applied, we need to create an aggregator node (σ), called a *root node* for storing "extra" organizational nodes that are created by the rewriting operations and that can not be affixed to any other part of the task structure. Recall, that we start off with a single agent whose local task view is equivalent to the global task view, $\mathbf{t}$. Hence, the created root node will be $\sigma_1 =< \textsc{Root}, \{\mathbf{t}\} >$. This node is shown on the left of Figure 3.3.

Breakup: The rationale behind the breakup operator is to divide the workload of an agent so that parts of it can be assigned to a new agent during the spawning process. If the workload of an agent consisted only of executable methods (M), this would be a simple case of picking some subset of M for the spawned agent.

However, in our problem domain, methods are (recursively) aggregated into tasks using CAFs and may have interrelationships (NLEs) with other tasks and methods. Hence, executable methods cannot be executed in isolation without considering all the interdependent effects of that execution.

Hence, when a spawning agent divides a local task structure, A into two subparts B (for itself) and C (for the spawned agent), it still needs to maintain some knowledge about the tasks/methods in C while, at the same time, allowing the spawned agent to have as much autonomy as possible about the execution of C. Specifically the agent will need to know about the subset of nodes in C that are interrelated to the nodes in B, either through NLEs or through subtask relations. We will call this subset the *related set*, of B. Similarly, the spawned agent will need to know some information about the nodes in B that are interrelated to the nodes in C through NLEs (i.e. the *related set* of C).

Furthermore, to allow for the maximum autonomy of both the spawning agent and the spawned agent, we limit this knowledge to consist of (1) the identity (label) of the nodes in the related set and (2) the relationship (i.e. subtask or NLE) through which they are related. Once the agent has been spawned, the two agents can negotiate a coordination mechanism for the relationship (for details see Section 3.4.2).

This knowledge will be preserved by creating non-local nodes ($\diamond$'s) to replace the nodes in the related set. During the breakup rewriting operation, the NLEs will be altered to point to/from the non-local nodes instead of the the domain nodes in the related sets. These non-local-nodes will be added to the root-node. This process is illustrated in Figure 3.3 and the algorithm for the breakup operator is shown in Algorithm 1.

Algorithm 1 BREAKUP $(\tau \in \Sigma, \upsilon \in \mathcal{D})$

```
1: $\overline{\tau} \Leftarrow$ DESCENDENTS$(\tau)$ − DESCENDENTS$(\upsilon)$
2: $\overline{\upsilon} \Leftarrow$ DESCENDENTS$(\upsilon)$
3: for all $\{ N \mid N \in$ NLEs$(\tau) \}$ do
4:   if (SOURCE$(N) \in \overline{\tau}$ and SINK$(N) \in \overline{\upsilon}$) or (SOURCE$(N) \in \overline{\upsilon}$ and SINK$(N)$
     $\in \overline{\tau}$) then
5:     $x \Leftarrow$ GETNONLOCALNODE(SOURCE$(N)$)
6:     $y \Leftarrow$ GETNONLOCALNODE(SINK$(N)$)
7:     $M \Leftarrow$ COPYNLE$(N)$
8:     REPLACENODE$(N$, SOURCE$(N)$, $x)$
9:     REPLACENODE$(M$, SINK$(N)$, $y)$
10:  end if
11: end for
12: $x \Leftarrow$ GETNONLOCALNODE$(\upsilon)$
13: REPLACENODE$(\tau, \upsilon, x)$
14: return CREATEROOTNODE$(\upsilon)$
```

In this algorithm, υ is the TÆMS node selected for assignment to the spawned agent[14], $\overline{\tau}$ (line 1) is the set of all the TÆMS nodes that will remain in the leftover part, while $\overline{\upsilon}$ (line 2) is the set of all the TÆMS nodes that will be assigned to the spawned agent. The For loop and If statements in lines 3 and 4 are responsible for finding all the NLEs that have a source node in the remaining part ($\overline{\tau}$) and a sink in the spawned part ($\overline{\upsilon}$) or vice versa. Finally, lines 5–9 are responsible for (a) creating non-local-nodes for both the source and sink of this NLE; (b) creating a copy of this NLE; and (c) changing different ends of the two NLEs to point towards the non-local nodes instead of the domain nodes.

The running time of this algorithm is $O(n \cdot m)$, where n is the number of TÆMS nodes in the task structure and m is the number of NLEs in the task structure.

[14] Note that we allow non-contiguous TÆMS nodes (or υ's) to be broken up via iterations of this algorithm.

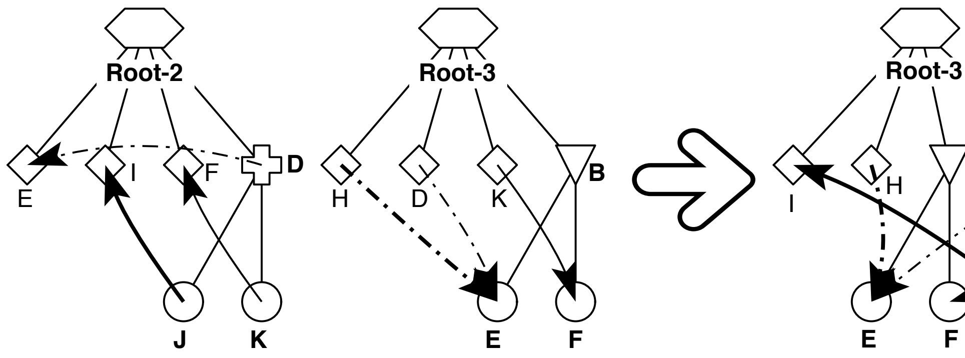


Figure 3.4: The *Merging* Task Rewriting Primitive. This figure shows the merging of Root-3 and Root 2. Notice how the non-local nodes (◇'s) E and F in Root-2 and non-local nodes D and K in Root-3 are destroyed as a result of the merge operation.

Merging: The idea behind the merging operator is to allow two agents to be composed into a single agent. Hence, merging involves combining two different local task structures from two different agents to form one local task structure.

Two requirements for the merging operation are (a) merging should be the exact inverse of breakup, i.e. if A is a task structure that was broken into B and C, merging B and C should give A; and (b) merging should be associative, i.e. the resultant local task structure formed after merging should not depend on the order in which the constituent local task structures were combined. Stated in another way, if using n breakup operations on a root node, σ, generates n local task structures ($\{\sigma_1, \sigma_2, ..., \sigma_n\}$), then n merging operations on these task structures, in *any* order, should regenerate σ. Note that we allow any two arbitrary task structures to be merged and hence allow any two agents to be composed with one another — hence the order of the merging operations can be completely independent and different from the order of the breakup operations.

An example of a merging operation is shown in Figure 3.4 and the algorithm for the merging operator is shown in Algorithm 2. In order to fulfill these

Algorithm 2 MERGE ($\tau \in \Sigma, \upsilon \in \Sigma$)

```
 1: Let $\upsilon =< \text{ROOT}, s_\upsilon >$
 2: for all $\{ y \mid y \in \text{DESCENDENTS}(s_\upsilon) \}$ do
 3:   $x \Leftarrow \text{FINDNODE}(\tau, \text{LABEL}(y))$
 4:   if $\text{NULL}(x)$ then
 5:     $\text{DELETENODE}(\upsilon, y)$
 6:     if $y \in s_\upsilon$ then
 7:       $\text{ADDNODE}(\tau, y)$
 8:     end if
 9:   else if $(x \in \diamondsuit) \wedge (y \in \diamondsuit)$ then
10:     $\text{MERGENODES}(\tau, x, y)$
11:   else if $x \in \mathcal{D} \wedge y \in \diamondsuit$ then
12:     $\text{DELETENODE}(\upsilon, y)$
13:   else if $x \in \diamondsuit \wedge y \in \mathcal{D}$ then
14:     $\text{REPLACENODE}(\tau, x, y)$
15:   end if
16: end for
17: return $\tau$
```

requirements, firstly, the domain nodes in the two local task structures, σ_1, σ_2, have to be merged to form the same graph structure as in the global task structure. This is done in lines 4–8 of the algorithm. Furthermore, any non local nodes that might exist in DESCENDENTS(σ_1) that have corresponding domain nodes in σ_2 have to be eliminated and vice versa. This is done in lines 11–15 of the algorithm. Finally, any two non local nodes that have the same identity should be merged into a single non-local node (lines 9–11) or formally $\exists \diamond_1, \exists \diamond_2 \mid (\diamond_1 =< \eta_1 > \wedge \diamond_2 =< \eta_2 > \wedge \eta_1 = \eta_2) \Rightarrow \diamond_1 = \diamond_2$.

The running time of Algorithm 2 is $O(n^2)$, where n is the *maximum* of number of TÆMS nodes in the two task structures being merged.

Cloning: When we discussed the breakup operator above, we said that the rationale was to use it to *"divide the workload of an agent so that parts of it can be assigned to a new agent ..."*. However, a precondition for applying the breakup operator is that number of executable methods in the root node, σ,

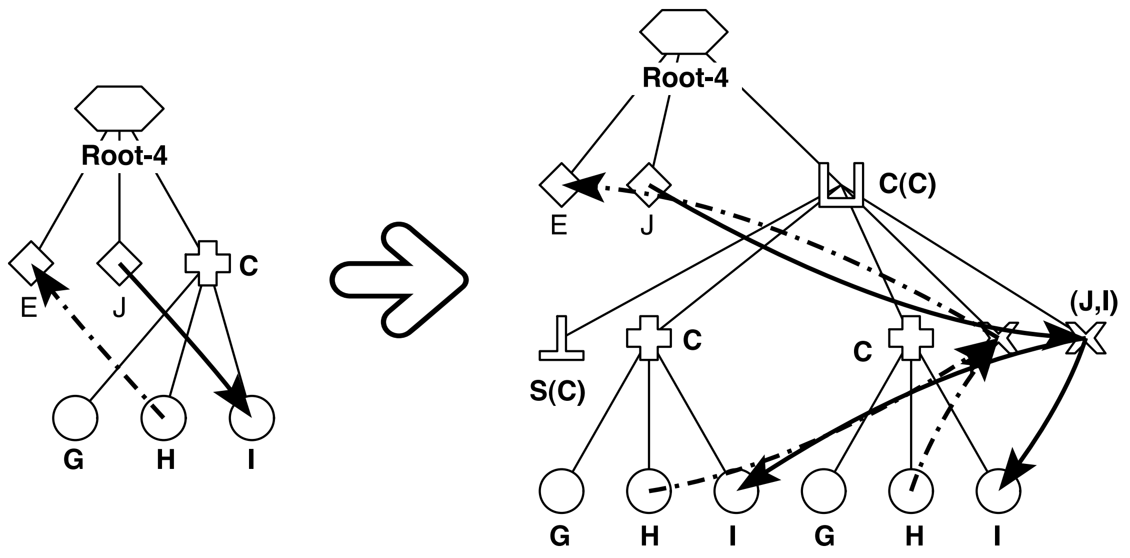


Figure 3.5: The *Cloning* Task Rewriting Primitive. This figure shows the cloning of Node C in Root-4. The $\sqcup$ node (Node C(C)) is used to represent a clone container; the $\perp$ node (Node S(C))is used to select which clone to "start", while the $\times$ nodes represent the NLE-inheriting-methods.

(of the local task structure) of the spawning agent, A, should be greater than 1; $(|\{x \mid x \in \textsc{Descendents}(\sigma) \wedge x \in M\}| > 1)$. This precondition exists because it makes no sense for A to spawn off a new agent, B, and assign it the one and only executable method that agent A was executing — effectively freeing up A but creating a just as much overloaded agent, B.

To overcome this restriction, we introduce a *cloning* operator that is responsible for making two copies, $< c_1, c_2 >$ of a substructure, $v \in \mathcal{D}$[15] so that the root task, τ can be broken up at node, v, and the breakaway part, c_2, be allocated to a new agent. Hence, the cloning operator is always meant to be used in association with the breakup operator and the breakup operation should come after the cloning operation.

An example of the cloning operator is shown in Figure 3.5 and the algorithm

[15] Note that we allow both tasks and methods to be cloned

is described in Algorithm 3. To clone a node, υ in a root task, τ, we first create a new container node, $\sigma_c =< \textsc{Clone}, \{\upsilon\} >$, called a *clone container* and replace υ in τ with σ_c. The clone container will be used to "hold" all the created clones.

Algorithm 3 Clone $(\tau \in \Sigma, \upsilon \in \mathcal{D})$

```
1:  $\overline{\tau} \Leftarrow$ Descendents($\tau$) $-$ Descendents($\upsilon$)
2:  $\overline{\upsilon} \Leftarrow$ Descendents($\upsilon$)
3:  $\phi \Leftarrow$ CreateCloneContainer($\upsilon$)
4:  $\phi \Leftarrow$ CreateCloneSelector()
5:  for all $\{ x \mid x \in \overline{\upsilon} \}$ do
6:    $y \Leftarrow$ CopyNode($x$)
7:    AddNode($\phi, y$)
8:  end for
9:  for all $\{ N \mid N \in$ NLEs($\upsilon$) $\}$ do
10:   if Source($N$) $\in \overline{\tau}$ then
11:     $x \Leftarrow$ CreateInheritingNode()
12:     AddNode($\phi, x$)
13:     $L \Leftarrow$ CopyNLE($N$)
14:     $M \Leftarrow$ CopyNLE($N$)
15:     ReplaceNode($N$, Sink($N$), $x$)
16:     ReplaceNode($L$, Source($L$), $x$)
17:     ReplaceNode($M$, Source($M$), $x$)
18:     $y \Leftarrow$ FindNode($\phi$, Sink($M$))
19:     ReplaceNode($M$, Sink($M$), $y$)
20:   else if Sink($N$) $\in \overline{\tau}$ then
21:     {Similar to the source}
22:   end if
23: end for
24: return $\phi$
```

Next we need some mechanism to select amongst the clones. That is, when a new task instance arrives, we have to pick one of the clones (and by inference, one of the owning agents) to run that instance. To do this, we create a clone selector $s_c \in \mathcal{S}_\mathcal{C}$ method, and add this method to the clone container. Formally, $\mathcal{S}_\mathcal{C} = \{s_1, s_2, ..., s_n\}$ is the set of all clone selectors, where each individual selector, $s_i =< t_i, n_i >$. In this context $t_i \in \{\textsc{Load-Balancing}, \textsc{Robustness}\}$ is the reason behind cloning these nodes[16]; and $n_i = \{(c_1, i_1), (c_2, i_2), ..., (c_n, i_n)\}$ is the set of mappings from a clone to information about that clone. This information may include details like the set of task instances assigned to a particular clone, the average quality, cost and duration of the previous executed instances, etc.

Finally, there might be some NLEs in the clones c_1 and c_2 that have a source or destination as a non-clone node. (Formally, $\{e \in E \mid [\textsc{Source}(e) \in \textsc{Descendants}(c_1) \wedge \textsc{Sink}(e) \in (\textsc{Descendants}(\sigma) - \textsc{Descendants}(c_1))] \vee [\textsc{Sink}(e) \in \textsc{Descendants}(c_1) \wedge \textsc{Source}(e) \in (\textsc{Descendants}(\sigma) - \textsc{Descendants}(c_1))]\}$). Such NLEs that transcend clone boundaries have to handled carefully in order to (a) preserve their original semantics and (b) allow the presence of clones to be transparent to the non clone nodes. In order to achieve this effect, we create special methods called *NLE-Inheritors*, ($\mathcal{N}$). These methods are simply conduits for the effects from the cloned nodes to the non-clone nodes.

In Algorithm 3, line 3 is responsible for creating a new clone container and line 4 creates a clone selector. Lines 5–8 are responsible for cloning (creating actual copies of) the domain nodes. Finally, lines 9–23 are responsible for creating the NLEs.

The running time of this algorithm is $O(n \cdot m)$, where n is the number of TÆMS nodes in the task structure and m is the number of NLEs in the task structure.

Iterate: The *Iterate* operator allows us to deal with ITERATION template node

[16] In addition to being used for load balancing, another advantage of the cloning operator is that it can be used to increase the *robustness capacity* of an agent by having multiple agents work on the same task simultaneously. See Section 3.4.7 for details.

($\oint \in \int$) by (a) checking the results of executing the control node to determine the number of iterations, n, of the template node required and (b) checking the number of existing copies, m of the *expansion node*, e — If $m < n$, $n - m$ extra copies of the expansion subtree are generated according to the algorithm presented in Algorithm 4.

Algorithm 4 ITERATE ($\tau \in \Sigma, \oint \in \int$)

```
1: Let $\oint = (tp, T_C, T_E, q)$
2: $n \Leftarrow$ RESULTS($T_C$)
3: $m \Leftarrow$ |SUBTASKS($\oint$)|
4: $\overline{\tau} \Leftarrow$ DESCENDENTS($\tau$) − DESCENDENTS($T_E$)
5: $\overline{\oint} \Leftarrow$ DESCENDENTS($T_E$)
6: for $i \leftarrow 1, (n - m)$ do
7:   for all $\{ x \mid x \in \overline{\oint} \}$ do
8:     $y \Leftarrow$ COPYNODE($x$)
9:     ADDNODE($\oint, y$)
10:    for all $\{ N \mid N \in$ NLES($x$) $\}$ do
11:      if SOURCE($N$) $\in \overline{\tau}$ then
12:        $z \Leftarrow$ FINDINHERITINGNODE($N$)
13:        $M \Leftarrow$ COPYNLE($N$)
14:        REPLACENODE($M$, SOURCE($M$), $y$)
15:        REPLACENODE($M$, SINK($M$), $z$)
16:      else if SINK($N$) $\in \overline{\tau}$ then
17:        {Similar to the source}
18:      end if
19:    end for
20:  end for
21: end for
```

The running time of Algorithm 4 is $O(l \cdot n \cdot m)$, where l is the number of iterations, n is the number of TÆMS nodes in the task structure and m is the number of NLEs in the task structure.

These operators result in the rewriting of a local task structure. In the case of agent spawning, the spawning agent, A, selects a node, $v \in \mathcal{D}$ either for breakup or for cloning (followed by breakup)[17] runs the breakup/cloning+breakup operator to divide its local task structure into two parts, $< \sigma_1, \sigma_2 >$, and then spawns a new agent, B, with σ_2 as its local task structure.

For agent composition, on the other hand, composing agent, A with a local task structure σ_1, selects another agent, B with a local task structure σ_2, to compose with. Agent A then sends a message to Agent B requesting composition. Agent B then call the merging operator to merge σ_1 and σ_2 to form a single local task structure, σ. Agent B can now be killed and the composition operation is now compete.

3.4.2 Coordinating the agents

We allow various coordination mechanisms to be used with our approach. Coordination between the agents is also achieved by rewriting the local task structures of the agents. Recall that a newly spawned agent will have a local tasks structure consisting of domain nodes and organizational nodes. The organizational nodes consist of, amongst other things, non-local-nodes ($\Diamond$) that represent domain nodes in other agents. These non-local nodes form coordination points between the agents and will be overwritten with coordination nodes corresponding to the selected coordination mechanism.

To select a coordination mechanism, (1) the newly spawned agent starts off a negotiation phase in which it sends a mechanism proposal for all the non-local-nodes to the agents that own the corresponding domain nodes; and (2) the other agents can either accept the coordination mechanism or they can send a counter-proposal. This exchange is repeated until both the agents commit to the same coordination

[17] See Section 3.4.5.2 for a description of different strategies that can be used to choose between using the breakup operator on its own or in conjunction with the cloning operator.

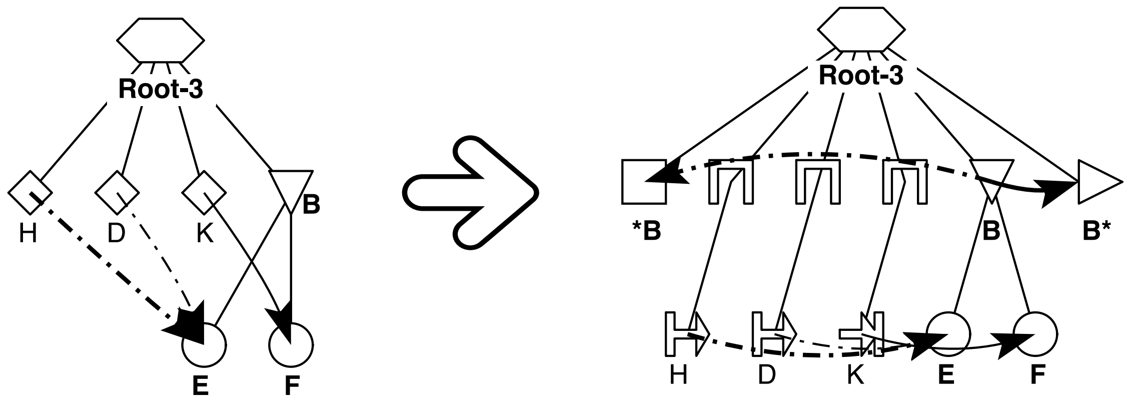


Figure 3.6: Figure showing the addition of coordination nodes to the task structure. The ⊓ node represents a coordination task. The↦ methods (Nodes H and D) involve sending results to the specified nodes while the ⇥ method (Node K) indicates that the agent needs to wait for a result from whatever agent is processing that node. The □ node is added to allow Node B to wait for the preceding node, while the ▷ node is used to start the succeeding nodes.

mechanism (or until all the coordination mechanisms have been exhausted, in which case a default mechanism is selected).

Whereas many coordination mechanisms can potentially be supported by our approach, we have currently only implemented the *send-results/wait-for-results mechanism*[18]. In this mechanism, each agent executes methods independently of the other agents and sends a result to the other agents as soon as they become available. If to execute a method, an agent needs results from another agents, it simply waits for those results. The addition of this coordination mechanism to the local task structures is shown in Figure 3.6. In the future, we would like to implement more complicated coordination mechanisms.

[18] This mechanism is equivalent to the GPGP DO Commitment.

3.4.3 Norms

Norms generally come in three forms — obligations, permissions and prohibitions [McNamara & Prakken, 1999]. In our approach, obligations are enforced through the coordination mechanism being used. Permissions, on the other hand, are implicit in the local task structures of the agents and, by inference, in the way in which the global task structure has been divided amongst the agents. If an Agent, A has a local task structure τ, then Agent A has the permission to perform all the tasks and methods represented by $\textsc{Descendents}(\tau)$.

Prohibitions, however, have to be explicitly represented. Since roles are represented in our system using TÆMS nodes $(T \cup M)$, prohibition norms are implemented as constraints on the nodes of a TÆMS task structure. Currently, we have implemented two kinds of prohibition norms: (a) unary norms of the form $U(tp, T_N)$, where $tp \in \{\textsc{Never-Breakup}, \textsc{Always-Breakup}\}$, is the *type* of the norm — The $\textsc{Never-Breakup}$ type ensures that a TÆMS node will never be selected for breakup by an agent[19] and the $\textsc{Always-Breakup}$ norm forces the agent to breakup at that node; and (b) binary norms of the form $B(tp, T_N, T_M)$, where $\{tp \in \textsc{Same}, \textsc{Different}\}$ is the type of the norm and $T_N, T_M \in (T \cup M)$ are TÆMS nodes. The $\textsc{Same}$ norm indicates that the nodes T_N and T_M should be done by the same agent, whereas the $\textsc{Different}$ norm implies that the nodes T_N and T_M should be performed by different agents.

3.4.4 Formal Definition of OSD

For our purposes, we define OSD as a function that maps an ordered set of problem solving requests (or task instances) and a set of performance criteria to an ordered set of organizations, together with their execution profile and aggregate utilities. Formally,

[19] See Section 3.4.5.1 for more information on how a node is selected for breakup.

$$OSD(\mathbf{P}, \mathfrak{U}, \mathfrak{C}) \rightarrow < \mathbf{O}, \alpha, \gamma > \tag{3.23}$$

where:

- $\mathbf{P} =< P_1, P_2, P_3, ..., P_n >$ is an ordered set of problem solving requests or task instances as formally defined in Section 3.3 above.

- $\mathfrak{U}$(Problem Instance, Execution Profile) : $P, \alpha \rightarrow N^{+}$[20] is a utility-calculation function that is used to encode a set of domain-dependent optimizing criteria for the problem instances.

- $\mathfrak{C}$(Organizational Instance) : $O \rightarrow N$[21] is a function that returns the "cost" of creating that particular organizational instance. This includes the cost of spawning the agents required for creating the organization, the cost of allocating resources for the agents and the cost of coordinating the agents.

- $\mathbf{O} =< O_1, O_2, O_3, ..., O_m >$, is an ordered set of m organizational instances or organizations. Each organization, O_i can be further denoted using the pair $< \mathbf{A_i}, \mathbf{C_i} >$, where:
  - $\mathbf{A_i} = \{a_1, a_2, ..., a_y\}$ is the finite set of Agents that are present in the i^{th} organization, where each agent, a_j, is a pair $< \tau_j, \mathrm{OR}_j >$. In this tuple:
    * τ_j is an augmented TÆMS task structure representing the local-task structure of Agent j.
    * OR_j is a set of ordered pairs of the form $< \Gamma_k \in R, v_k \in N >$, representing the fact that v_k amounts of resource Γ_k are owned by agent a_j.

[20] The execution profile is formally defined below.

[21] The organizational instance is defined below.

- $\mathbf{C_i} = \{c_1, c_2, ..., c_z\}$ is a finite set of coordination relationships that exist in the organization. Each c_j is an ordered pair, denoted by $< a_j \in A_i, b_j \in A_i >$, denoting the fact that agent a_j has a coordination relationship with b_j [22]

- α, is the execution profile of the multiagent system as a whole. The execution profile maps each problem instance in $\mathbf{P}$ to the *actual* quality/cost/duration characteristics of its execution. Formally, if $\alpha = < \alpha_1, \alpha_2, ..., \alpha_n >$, then $\alpha_i = (P_i, \beta_i)$. Here, P_i is a problem instance and the $\beta_i = \{(N_1, q_1, c_1, d_1), (N_2, q_2, c_2, d_2), ..., (N_m, q_m, c_m, d_m)\}$, is a set of mappings from domain TÆMS nodes in $\mathcal{D}$ to the quality (q_j), cost (c_j) and duration (d_j) characteristics of that node. That is q_j, c_j and d_j represent the actual quality accrued, the cost incurred and the time spent by the organization while "executing" node N_j.

- γ, is the aggregate utility accrued by the multiagent system minus the organizational costs. Formally,

$$\gamma = \sum_{i=1}^{|\mathbf{P}|} \mathfrak{U}(P_i, \beta_i) - \sum_{j=1}^{|\mathbf{O}|} \mathfrak{C}(O_j) \tag{3.24}$$

Using Organizational Self-Design, our system will try to maximize either the aggregate utility, γ accrued by the multiagent system or the average domain-dependent utility of the organization minus the organizational costs depending on the designer's preference:

$$\frac{\sum_{i=1}^{|\mathbf{P}|} \mathfrak{U}(P_i, \beta_i)}{|\mathbf{P}|} - \frac{\sum_{j=1}^{|\mathbf{O}|} \mathfrak{C}(O_j)}{|\mathbf{O}|} \tag{3.25}$$

[22] Note that we only allow pairwise coordination relationships between the agents. This is because the coordination relationships are generated due to the presence of NLEs between two nodes that are the responsibility of two different agents. Also note that multi-agent coordination relationships can easily be modeled using pairwise relationships.

3.4.5 Organization Formation and Adaptation

Algorithm 5 FindRolesForSpawnedAgent (SpawningAgentRoles) : $(T \cup M) \rightarrow (T \cup M)$

```
1: $\mathscr{R} \leftarrow$ SpawningAgentRoles
2: selectedRoles $\leftarrow$ nil
3: for roleSet in $[\mathscr{P}(\mathscr{R}) - \{\phi, \mathscr{R}\}]$ do
4:   if Cost$(\mathscr{R}, roleSet) <$ Cost$(\mathscr{R}, selectedRoles)$ then
5:     selectedRoles $\leftarrow$ roleSet
6:   end if
7: end for
8: return selectedRoles
```

Algorithm 6 GetResourceCost(Roles) : $(T \cup M) \rightarrow \Re$

```
1: $\mathscr{M} \leftarrow (Roles \cap M)$
2: cost $\leftarrow$ 0
3: for resource in $R$ do
4:   maxResourceUsage $\leftarrow$ 0
5:   for method in $\mathscr{M}$ do
6:     if $\rho(method, resource) >$ maxResourceUsage then
7:       max $\leftarrow \rho(method, resource)$
8:     end if
9:   end for
10:  cost $\leftarrow$ cost +
       [C(resource) $\cdot$ maxResourceUsage]
11: end for
12: return cost
```

3.4.5.1 Role allocation during spawning

One of the key questions that the agent doing the spawning needs to answer is — *which of its local-roles should it assign to the newly spawned agent and which of its local roles should it keep to itself?* The onus of answering this question falls on the FindRolesForSpawnedAgent() function, shown in Algorithm 5 above. This function takes the set of local roles that are the responsibility of the spawning agent and returns a subset of those roles for allocation to the newly spawned agent. The

Algorithm 7 GetExpectedDuration(Roles) : $(T \cup M) \rightarrow N^+$

```
1: $\mathscr{M} \leftarrow (Roles \cap M)$
2: exptDuration ← 0
3: for [$outcome =< (q, c, d), outcomeProb >$] in $\mathscr{M}$ do
4:     exptOutcomeDuration ← 0
5:     for (n,p) in d do
6:         exptOutcomeDuration ← $n \cdot p$
7:     end for
8:     $exptDuration \leftarrow exptDuration + [exptOutcomeDuration \cdot outcomeProb]$
9: end for
10: return exptDuration
```

running time of Algorithm 5 is $O(2^n)$, where n is the number of roles of the spawning agent[23]. This subset is selected based on the results of a cost function as is evident from line 4 of the algorithm. Since the use of different cost functions will result in different organizational structures and since we have no a priori reason to believe that one cost function will out-perform the other, we evaluated the performance of three different cost functions based on the following three different heuristics:

Allocating top-most roles first: This heuristic always breaks up at the top-most nodes first. That is, if the nodes of a task structure were numbered, starting from the root, in a breadth-first fashion, then this heuristic would select the local-role of the spawning agent that had the lowest number and breakup that node (by allocating one of its subtasks to the newly spawned agent). We selected this heuristic because (a) it is the simplest to implement, (b) fastest to run (the role allocation can be done in constant time without the need of a search through the task structure) and (c) it makes sense from a human-organizational perspective as

[23] The running time is exponential in the number of roles of the spawning agent because we iterate over the power set of the set of these roles in Line 3 of this algorithm. To overcome this exponential running time, we can make a simplification by only iterating over the set of roles, $\mathscr{R}$, in Line 3 instead of considering the power set. We we can still generate every possible organization since any two or more spawned agents can always compose together after they are spawned.

this heuristic corresponds to dividing an organization along functional lines.

Minimizing total resources: This heuristic attempts to minimize the total cost of the resources needed by the agents in the organization to execute their roles. (See Algorithm 6). If $\mathscr{R}$ be the local roles of the spawning agent and R' be the subset of roles being evaluated for allocation to the newly spawned agent, the cost function for this heuristic is given by: $\textsc{Cost}(\mathscr{R}, R') \leftarrow \textsc{GetResourceCost}(\mathscr{R} - R') + \textsc{GetResourceCost}(R')$. The running time of Algorithm 6 is $O(r \cdot n)$, where $r = |R|$ is the number of potential resources and n is the number of TÆMS nodes in the task structure.

Balancing execution time: This heuristic attempts to allocate roles in a way that tries to ensure that each agent has an equal amount of work to do. For each potential role allocation, this heuristic works by calculating the absolute value of the difference between the expected duration of its own roles after spawning and the expected duration of the roles of the newly spawned agent. If this difference is close to zero, then the both the agents have roughly the same amount of work to do. Formally, if $\mathscr{R}$ be the local roles of the spawning agent and R' be the subset of roles being evaluated for allocation to the newly spawned agent, then the cost function for this heuristic is given by: $\textsc{Cost}(\mathscr{R}, R') \leftarrow |\textsc{GetExpectedDuration}(\mathscr{R} - R') - \textsc{GetExpectedDuration}(R')|$. See Algorithm 6. The running time of this algorithm is $O(n \cdot m)$, where n is the number of TÆMS methods in the task structure and m is the number of outcomes of a method.

To evaluate these heuristics, we ran a series of experiments that tested the performance of the resultant organization on randomly generated task structures. The results are given in Section 4.3.

3.4.5.2 Selecting a Spawning Strategy

As described in Section 3.4, if an agent is overloaded (i.e. it can't complete the problems in its task queue before their respective deadlines), it *spawns* off a new agent to handle part of its load. While spawning off a new agent, the overloaded agent has two options:

1. It could divide the problem into smaller subproblems and assign one of the smaller problems to the newly spawned agent. We will refer to this approach as *breakup* since this approach uses the *breakup* operator as defined in Section 3.4.1.

2. It could assign half of the outstanding problems in its task queue to the newly spawned agent. The individual problems are solved in their entirety by the two agents. In this approach, the spawning agent has effectively cloned itself. Hence, we refer to this approach as *cloning*. This approach makes use of the cloning operator defined in 3.4.1.

Each of these two approaches have their own advantages and disadvantages: (1) Breakup may be the only option if the task is too "big" for any single agent to do on its own. Similarly cloning may be the only option if task cannot be broken up into smaller parts. (2) Breakup will typically use less resources than cloning, especially if the subtasks use a different set of resources. (3) Also, breakup would be better in situations in which the agents include a learning component, since the number of instances over which the information is being learned would be larger. (4) If, however, the subtasks are interdependent breakup would require more coordination between the agents executing the interdependent parts. Hence, cloning would be better in such situations.

To analyze the tradeoff between breakup and cloning, we compared five spawning approaches:

1. **Breakup:** In this approach the task structure is always broken up into smaller subtasks using the *breakup* operator as described in Section 3.4.1 above.We

used the *Balancing Execution Time* (BET) heuristic to select the node to be allocated to the new agent, as defined in Section 3.4.5.1.

2. **Prefer Breakup:** This approach is the same as the *Breakup* approach, with the exception that if *Breakup* is infeasible, the *Cloning* approach (described below) is used. We define breakup as being infeasible if the local-task-structure of the spawning agent consists of a *single* executable method. (Formally, if σ is the root node (of the local task structure) of the spawning agent, A, then $\mathit{feasible}(\mathit{Breakup}(\sigma)) \Leftrightarrow |\{x \mid x \in \textsc{Descendents}(\sigma) \wedge x \in M\}| > 1)$. This feasibility condition exists because it makes no sense for A to spawn off a new agent, B, and assign it the one and only executable method that agent A was executing — effectively freeing up A but creating a just as much overloaded agent, B.

3. **Cloning:** In this approach, the root of the task structure is always cloned and assigned to the newly spawned agent. All the agents in this approach are exact replicas in that all of them have equivalent roles, in which they are responsible for the complete task structure.

4. **Prefer Cloning:** This is similar to the *Cloning* approach, with the exception that if *Cloning* is infeasible given the current task load, the agent will *Breakup* according to the BET heuristic. We define cloning to be infeasible if the number of clones of a node is greater than or equal to the number of outstanding tasks in the spawning agent's task queue. Cloning is infeasible in such cases because cloning assigns task instances to specific clones (and by inference their owning agents). The only way to assign a task instance to a new clone, in such cases, would be to transfer an instance from an existing clone to the new clone. This would free up the existing clone but would equally overload the new clone.

5. **Hybrid Model:** This is a hybrid model, that we designed on the basis of a preliminary set of experiments. This model uses a combination of cloning the highest level goal and breakup according to the BET heuristics. It works by computing a utility value $U_{Breakup}(\vartheta)$, which is the expected utility of breaking up according to the BET heuristics. Here, ϑ is the node that was selected for breakup by the BET heuristic. If $U_{Breakup}(\vartheta) > \chi$, where χ is a constant called the breakup threshold, the agent chooses to breakup. Otherwise it clones the highest level node.

   To compute $U_{Breakup}(\vartheta)$, we start by initializing it to another constant . Then according to various "truisms" about the current local-task-structure of the agent, the selected breakup node and the environmental conditions, the value of $U_{Breakup}(\vartheta)$ is either increased or decreased according to the formula: $U_{Breakup}(\vartheta) = U_{Breakup}(\vartheta)(1-(\xi \cdot var))$, where both ξ (a constant) and var have values between -1 and 1. Both the value of ξ and var depend on the evidence being considered. That is:

$$U_{Breakup}(\vartheta) = \psi \cdot \prod_{i=1}^{4} [1 - (\xi_i \cdot var_i)] \tag{3.26}$$

   For our results, we have considered the following parameters:

   - The difference in execution time between the selected breakup node and the leftover node as defined by the BET heuristic. If this difference is large, the spawning agent and the spawned agent will be unbalanced and hence it makes sense to prefer cloning over breakup.

$$var_1 = \frac{\text{Execution Time(Breakaway Part)} - \text{Execution Time(Leftover Part)}}{\text{Execution Time(Original Task)}} \tag{3.27}$$

   - The ratio of the number of NLEs that will have to be "broken" up as a result of the breakup to the total number of NLEs in the task structure.

The larger the number of these NLEs, the greater the coordination cost between the spawning agent and the spawned agent. Hence, it makes more sense to prefer cloning over breakup in cases where there are a large number of NLEs.

$$var_2 = \frac{\text{\# of NLEs Broken}}{\text{Total \# of NLEs}} \tag{3.28}$$

- The ratio of the difference in resource cost between breakup and cloning divided by the total resource cost. This ratio is used to tradeoff the increase in resource cost when selecting cloning over breakup.

$$var_3 = \frac{\text{Resource Cost(Breakup)} - \text{Resource Cost(Cloning)}}{\text{Total Resource Cost}} \tag{3.29}$$

- The average amount of time available for each task instance divided by the expected time needed for performing the task. This is a measure of the excess load in the system.

$$var_4 = \frac{\text{Average Time Available(instance)}}{\text{Expected Time Required(instance)}} \tag{3.30}$$

3.4.6 Detecting the need for organizational change

As organizational change is expensive (requiring clock cycles, allocation/deallocation of resources, etc.) we want a stable organizational structure that is well suited to the task and environmental conditions at hand. Hence, we wish to change the organizational structure only if the task structure and/or environmental conditions change. Also to allow temporary changes to the environmental conditions to be overlooked, we want the probability of an organizational change to be inversely proportional to the time since the last organizational change. If this time is relatively short, the agents are still adjusting to the changes in the environment - hence the probability of an agent initiating an organizational change should be high. Similarly, if the time since the last organizational change is relatively large, we wish to have a low probability of organizational change.

To allow this variation in probability of organizational change, we use simulated annealing to determine the probability of keeping an existing organizational structure. This probability is calculated using the annealing formula: $p = e^{-\frac{\Delta E}{kT}}$ where ΔE is the "amount" of overload/underload, T is the time since the last organizational change and k is a constant. The mechanism of computing ΔE is different for agent spawning than for agent composition and is described below. From this formula, if T is large, p, or the probability of keeping the existing organizational structure is large. The probability of organizational change, q, can be calculated using the formula, $q = 1 - p$. Note that the value of p is capped at a certain threshold in order to prevent the organization from being too sluggish in its reaction to environmental change.

3.4.6.1 Agent Spawning

Agent spawning should only occur when the agent doing the spawning is too overloaded to complete the tasks in its task queue by their given deadlines. The obvious question then is: "*How can the agents know when they are overloaded?*".

One way in which the agents could detect overload would be to wait until they fail to complete their tasks on time (i.e. an agent could wait till the deadline on a task in its task queue is exceeded). Two problems with this approach are:

1. An agent executing a method may, on a single run at any given time, take significantly longer to finish it than normal. That is, the missed deadline may be a one-time occurrence. This is because the time taken to execute a method is often non-deterministic and probabilistic[24]. Hence, missing a

[24] Recall that in TÆMS , executable methods are probabilistic, in that, methods are allowed to have multiple outcomes with different probabilities. Furthermore, each outcome is allowed to take a varying amount of time to complete based on some probability distribution. Hence, it is often not possible to know, *a priori*, how long a method is going to take to execute.

deadline on the current execution run does not automatically imply that the agent is overloaded or that it will consistently miss deadlines in the future.

2. The overload diagnosis only occurs after the agents have already failed to execute tasks in their task queue. Ideally, we would prefer to diagnose and prevent task failures before they actually occur.

The first problem can easily be rectified if the agents wait for a certain percentage of the tasks in their respective task queues to fail before spawning off new agents. However, this solution does nothing to address the second problem.

To have a more pro-active approach to organizational structuring, we use two different items of information as described below:

1. **Meta-Information about the task:** By meta-information, we mean the information about the characteristics of a task. For our purposes, we make use of three pieces of meta-information about the tasks being attempted:

    - the *min* time or the absolute minimum time that agent must have in order to have a non-zero probability of completing the task;
    - the *expt* time or the expected time needed to complete the task; and
    - the *g_min* time[25] or the minimum time needed to guarantee task completion, in the absence of failures. (That is, the time needed to have a probability of 1 of completing the task, in the absence of failures).

[25] Note that the *g_min* time is the same as the worst-case execution time (or *wcet*) for executable methods. However, for some CAFs, this time may differ significantly from the worst-case execution time. To see why, consider a task, T, with a *SUM* CAF and two executable methods A and B. A and B have *g_min* times (*wcet*s) of 2 and 7 respectively. Now, the *wcet* of T is 9, which occurs when the agent decides to schedule both methods A and B. However, the *g_min* time is only 2, since the agent only needs to schedule method A to guarantee a non-zero quality for task T.

To understand the difference between the three, consider an executable method that has two outcomes, the first of which takes a minimum of 3 cycles and a maximum of 5 cycles to execute and the second of which takes a minimum of 5 and a maximum of 7 cycles to execute. Now, the agent needs at least three cycles to have a non-zero probability of completing this executable method, which would be the case if the executable method had the first outcome and took only 3 cycles to achieve it. Hence the *min* time is 3. Also, the agent needs at least 7 cycles to guarantee task completion[26] — hence the *g_min* time is 7. The *expt* time would be the amount of time it takes to complete the method on average and would depend on the probability of the two outcomes.

Formally, if $\textsc{Outcomes}(\text{Method}) : M \rightarrow \{(o,p) \mid \text{o is an outcome of M} \wedge \text{p is the probability of outcome o}\}$ be a function that returns the outcomes of a method and $\textsc{DurationD}(\text{Outcome}) : O \rightarrow \{(m,q) \mid m \in N \wedge q \in \Re^+\}$, be a function that returns the duration distribution of outcome, O, then:

$$min(m \in M) \leftarrow \min_{(o,p) \in \mathrm{OUTCOMES}(m)} \left[\min_{(m,q) \in \mathrm{DURATIOND}(o)} m \right] \tag{3.31}$$

$$expt(m \in M) \leftarrow \sum_{(o,p) \in \mathrm{OUTCOMES}(m)} p \cdot \left[\sum_{(m,q) \in \mathrm{DURATIOND}(o)} (m \cdot q) \right] \tag{3.32}$$

$$g\_min(m \in M) \leftarrow \max_{(o,p) \in \mathrm{OUTCOMES}(m)} \left[\max_{(m,q) \in \mathrm{DURATIOND}(o)} m \right] \tag{3.33}$$

The meta-information for a higher-level node (task) depends on both the CAF of the task and the meta-information of its subtasks. Formally, if

[26] The reasoning behind the *g_min* time is that the agent cannot control the outcome of a method and, furthermore, has no way of knowing the actual duration of an outcome. This is because an agent can only decide on whether or not to execute a particular method but has no way of influencing or controlling the actual execution of the method. Hence, to guarantee method completion, the agent needs to allocate the *maximum* amount of time that any outcome of the method might take to complete.

CAF(Task) : $T \rightarrow \{\textsc{Min}, \textsc{Max}, \textsc{Sum}, \textsc{Exactly\_One}\}$ is a function that returns the characteristic accumulation function of a task, then this meta-information can be recursively computed from the meta-information of the subtasks using the following formulas:

$$min(t \in T) \leftarrow \begin{cases} \sum_{t_j \in \text{SUBTASKS}(t)} min(t_j) & \text{if CAF(t)} = \textsc{Min} \\ \min_{t_j \in \text{SUBTASKS}(t)} min(t_j) & \text{if CAF(t)} \in \{\textsc{Sum}, \textsc{Max}, \\ & \qquad\qquad \textsc{Exactly\_One}\} \end{cases} \tag{3.34}$$

$$expt(t \in T) \leftarrow \begin{cases} \sum_{t_j \in \text{SUBTASKS}(t)} expt(t_j) & \text{if CAF(t)} = \textsc{Min} \\ \text{soh}_{t_j \in \text{SUBTASKS}(t)} expt(t_j) & \text{if CAF(t)} \in \{ \textsc{Sum}, \textsc{Max} \} \\ \text{avg}_{t_j \in \text{SUBTASKS}(t)} expt(t_j) & \text{if CAF(t)} = \textsc{Exactly\_One} \end{cases} \tag{3.35}$$

$$g\_min(t \in T) \leftarrow \begin{cases} \sum_{t_j \in \text{SUBTASKS}(t)} g\_min(t_j) & \text{if CAF(t)} = \textsc{Min} \\ \min_{t_j \in \text{SUBTASKS}(t)} g\_min(t_j) & \text{if CAF(t)} \in \{\textsc{Sum}, \textsc{Max}, \\ & \qquad\qquad \textsc{Exactly\_One}\} \end{cases} \tag{3.36}$$

2. **The effective time available for completion of a task:** The time available for completion of a task is computed by assuming that all tasks are equally important, that all tasks need to be executed and that the total time available needs to be equally divided amongst all the outstanding tasks.[27] If these assumptions hold, we can compute the time available using the steps given

[27] These assumptions do not guarantee optimality in any way. Rather it is trivial to demonstrate a case where an agent would gain more quality by ignoring certain tasks. These assumptions have been chosen for simplicity rather than for optimality.

Algorithm 8 Algorithm to compute the effective time available for a task

```
1: Let $AT(T \cup M) \rightarrow N^+$ be a function that returns the arrival time of a
   task/method
2: Let $DT(T \cup M) \rightarrow N^+$ be a function that returns the deadline of a task/method

3: Fig. 3.7b: Divide the total time into consecutive time slices bounded by the
   arrival times and deadlines of each task.
4: Let $Start(ts)$ and $End(ts_i)$ be two functions that respectively return the starting
   and ending times of a time slice, $ts$.
   {Fig. 3.7c: Compute the time available per task for each time slice}
5: for each time slice, $ts_i$ do
6:    $n[ts_i] \leftarrow$ number of outstanding tasks in time slice i
7:    $d[ts_i] \leftarrow End(ts_i) - Start(ts_i) + 1$
8:    $t_{avail}[ts_i] \leftarrow n[ts_i]/d[ts_i]$
9: end for
   {Fig. 3.7d: Compute the total effective time available for each task}
10: for each task, $t_j$ in the task queue do
11:    $MySlices \leftarrow \{ts | AT(t_j) \leq Start(ts) \wedge End(ts) \leq DT(t_j)\}$
12:    $t_{avail}[t_j] \leftarrow \sum_{ts_i \in MySlices} t_{avail}[ts_i]$
13: end for
```

in Algorithm 8. Refer to Figure 3.7 for an example demonstrating the use of these steps.

Given these two pieces of information, the agent using Algorithm 9 to determine if agent spawning is necessary[28]. The crux of this algorithm is the *for* loop on line 2, which iterates over all the outstanding tasks and the *if* statements in lines 4 and 7, which compares the effective time available for a task, $t_{avail}(Task_i)$, against $min(Task_i)$. If the former is smaller, the agent is guaranteed to fail on the current task and the agent immediately spawns off a new agent (line 5). If, on the other hand, $t_{avail}(Task_i)$ is greater than $min(Task_i)$ but less than $g\_min(Task_i)$, the agent uses simulated annealing to calculate if a new agent should be spawned, with ΔE computed as shown in Equation 3.37. The reasoning behind computing ΔE

[28] The running time of Algorithm 9 is $O(n)$, where n is the number of outstanding tasks in the task queue.

Algorithm 9 Algorithm for determining if Agent Spawning is necessary

```
 1: Let $T_{curr} \leftarrow$ be the current time (the time instance at which this algorithm is
    run
 2: for $Task_i$ in $OutstandingTaskQueue$ do
 3:     Let $t_{avail}(Task_i) \leftarrow$ be the effective time available for $Task_i$, computed using
        Algorithm 8
 4:     if $t_{avail}(Task_i) < min(Task_i)$ then
 5:         SpawnAgent()
 6:     else
 7:         if $min(Task_i) < t_{avail}(Task_i) < g\_min(Task_i)$ then
 8:             SpawnAgent() with probability p calculated using the annealing formula,
                with
```

$$\Delta E = \frac{1}{\begin{bmatrix} \alpha \cdot [t_{avail}(Task_i) - min(Task_i)] \\ +(1-\alpha) \cdot [g\_min(Task_i) - min(Task_i)] \end{bmatrix}} \tag{3.37}$$

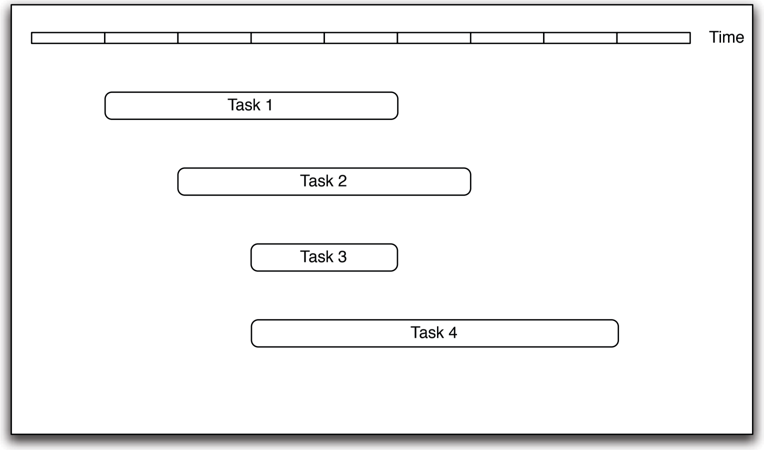
```
                where $\alpha$ is a constant.
 9:         else
10:             Do Nothing
11:         end if
12:     end if
13: end for
```

like this is that if $t_{avail}(Task_i)$ is less than $g\_min(Task_i)$, there is some probability of the agent failing $Task_i$ due to a lack of time.

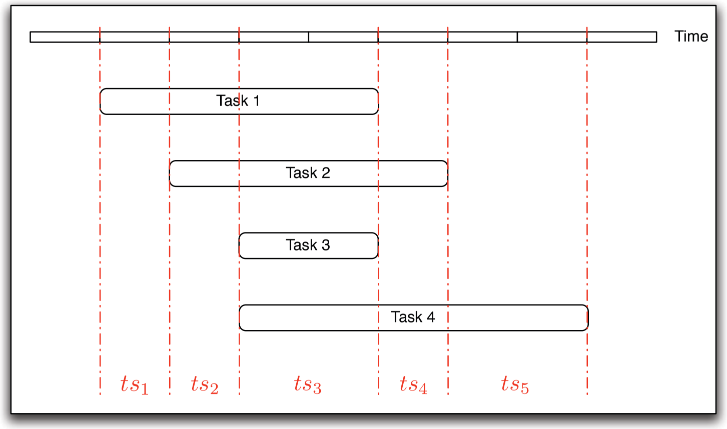
The probability of keeping an existing organizational structure in this case should then be proportional to (and ΔE should be inversely proportional to[29]): (a) the difference between the effective time available, $t_{avail}(Task_i)$, and $min(Task_i)$; and (b) the difference between $g\_min(Task_i)$ and $t_{avail}(Task_i)$, with α being a constant used to determine which of the terms (a) or (b) gets to dominate the calculation. We use former term, Term (a), because the closer $t_{avail}(Task_i)$ is to $min(Task_i)$ the greater the chance of the agent failing a task. We use the latter term, Term (b), because the greater the value of this difference, the greater the disparity between two outcomes of the method — one that takes a large amount of time and one that takes a smaller amount of time.

Note that since agent spawning is triggered by an indication that the agents might fail on the tasks in the task queue as opposed to actual failure on a task, the agents can be thought to have a pro-active approach to organizational design.

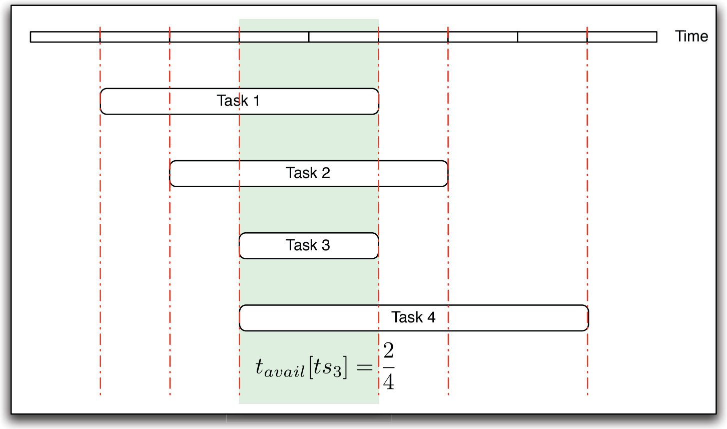
[29] Recall that if ΔE is large, p or the probability of keeping an existing organizational structure is low.



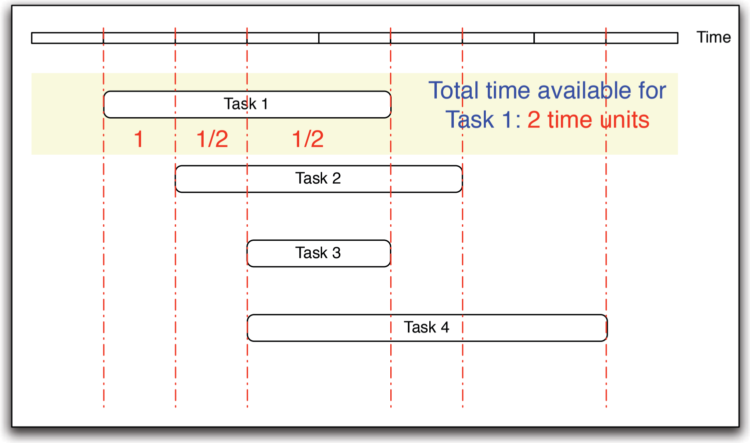
(a) The four outstanding tasks represented on a timeline.



(b) The total time is divided into consecutive time slices, bounded by the arrival time and deadline of the tasks.



(c) The *time available per task* is computed for each time slice.



(d) The time available for a task is computed by summing up the *time available per task* for each time slice in which the task appears.

Figure 3.7: Figure demonstrating the steps required to compute the effective time available for completion of a task on an example task queue with four outstanding tasks. Tasks 1, 2, 3 and 4 have arrival times of 1, 2, 3 and 3 respectively. The deadlines for the four tasks are, in order, 5, 6, 5, and 8.

3.4.6.2 Agent Composition

Agent composition is exactly orthogonal to agent spawning as agent composition only occurs when the agents are underloaded. In such a situation, some of the agents will be sitting idle waiting for tasks to arrive. These idle agents will either be utilizing resources while waiting, or more likely, will have resources allocated to them that could be used elsewhere in the system. An example of the former case is when agents are using CPU cycles while waiting (busy waiting). An example of the latter case is when agents have been allocated network bandwidth that is being unused while the agents are sitting idle. In either case, there is an inefficiency in the allocation and use of resources. In such cases, it makes sense to combine some of the free agents with other agents thus freeing unused resources.[30]

To calculate if agent composition is necessary, we again use the simulated annealing equation. However, in this case, ΔE is computed differently and is proportional to the amount of time for which the agent was idle. In particular, $\Delta E = \beta \cdot Idle\_Time$, where β is a constant and $Idle\_Time$ is the amount of time for which the agent was idle. If the agent has been sitting idle for a long period of time, ΔE is large, which implies that p, the probability of keeping the existing organizational structure, is low. Since agent composition only occurs after the agent is already idle (and hence already wasting resources), the agents can also be thought to have a reactive approach to organizational design.

Hence, organizational design in our agents is both pro-active (in the case of agent spawning) and reactive (in the case of agent composition). This combination of pro-active and reactive behavior gives our agents the ability to complete as many

[30] Another reason for combining agents might be to reduce the coordination overhead associated with the communication delay between the agents. If the communication time is greater than the time saved due to the greater parallelism between the tasks (as a result of having multiple agents), it makes sense to combine some of the agents.

tasks as possible, as the agents react fast to bad news (inability to complete the outstanding tasks) and slowly to good news (the agents being underloaded).

3.4.7 Robustness Mechanisms

Both of our robustness mechanisms involve three parts: (a) monitoring for agent failure; (b) maintaining state information for all the agents; and (c) restarting failed agents.

Furthermore, the underlying mechanism for monitoring and restarting is the same across the robustness mechanisms. Monitoring is achieved by sending out periodic *Are-You-Alive* messages to the set of monitored agents and waiting for *Alive* reply messages. If a reply is not received within a certain interval, we assume that the agent is dead and send a restart message to the environment. The individual mechanisms, however, differ in *who* is responsible for the monitoring and *which* set of agents are monitored.

State information is needed to restart a failed agent. At a minimum, this state information should contain the *organizational state* (i.e. the local task structure) of the agent being restarted. However, the local task information is not sufficient for restarting an agent in a complex domain. The restarted agent will still need information about the *execution context*, i.e. information about the outstanding task instances, information about the methods of a task instance that have already been executed (so that the agent does not try to re-execute them) and information about coordination commitments (because the subtasks have non-local effects and are interdependent on each other).

The two coordination mechanisms also use the same underlying mechanism for maintaining state information about the agents being monitored. Whenever an agent receives an *Are-You-Alive* message from its monitor, it adds its current state information (including information on both the organizational state and the execution context) to the *Alive* reply message. This state information is cached by the agents doing the monitoring and is used to restart failed agents.

3.4.7.1 Citizens Approach

The citizens approach involves creating a special monitoring agent (called a *sentinel agent*), which is responsible for all the robustness related responsibilities of the organization. This approach is the simplest to execute — the sentinel agent is the sole monitor that is responsible for monitoring all the agents in the organization.

The primary disadvantage of the citizens approach is that the sentinel agent has global knowledge about the complete organization — a problem we were trying to avoid by using the OSD approach in the first place. Furthermore, the sentinel can (a) quickly become overwhelmed by all the messages that it needs to track and (b) become a central point of failure[31]. The solution might be to add multiple sentinel agents — we will now need to create an organization for the sentinels (for which we could, again, use OSD) and a way of monitoring the monitors.

Hence, we focus on developing algorithms for the survivalist approach and use the citizens approach for comparison.

3.4.7.2 Survivalist Approach

In the survivalist approach, there are no special agents responsible for monitoring and restarting failed agents. Instead the domain agents divide the monitoring responsibilities amongst themselves. Furthermore, some/all domain agents may be replicated in order to (a) increase the robustness capacity of the organization; (b) decrease the response time to a failure, and (c) process task instances in parallel, thus helping to balance the load.

[31] It's unreasonable to assume that the other agents might fail, but the sentinel will never fail

The obvious advantage of the survivalist approach is that no one agent is overburdened with the monitoring responsibilities. Also there is no central point of failure and no agent with global knowledge of the organization. Furthermore, the survivalist approach can take into account the interplay between a satisficing organizational structure and probability of failure. For example, one way of achieving a higher level of robustness in the survivalist approach, given a large numbers of agent failures, would be to relax the task deadlines. However, such a relaxation would result in the system using fewer agents in order to conserve resources, which in turn would have a detrimental effect on the robustness. These advantages come at a cost of increased complexity of the monitoring mechanism.

3.4.7.3 Creating a monitoring set of agents

The monitoring set of an agent, *Agent A*, is defined as the set of agents that are responsible for monitoring *Agent A* for failures. We assume that the minimum cardinality of this set, N is an input to the organization[32]. Also in our approach, all monitoring is mutual, i.e. if *Agent A* is in the monitoring set of *Agent B* (i.e. if *Agent A* is responsible for monitoring the health of *Agent B*), then *Agent B* is in the monitoring set of *Agent A*. This is by design, because *Agent A* on receiving an *are-you-alive* request from *Agent B*, already knows that *Agent B* is alive and does not need to send *Agent B* a separate request.

Each agent is responsible for determining its monitoring set. At the time an agent, say *Agent A*, is first spawned, it runs the following algorithm:

1. *Agent A* determines its related set. The related set of *Agent A* is the set of agents that have a coordination relationship with *Agent A*. (This coordination

[32] It should be possible to develop an algorithm for learning the optimal value of N given the environment conditions — i.e. the probability of failure. We plan to incorporate this into our future work.

relationship would exists because of interdependent tasks and NLEs in the task structures of the agents).

2. If the number of agents in the related set of *Agent A* is greater than N, *Agent A* sends a message to each of the related agents, requesting the cardinality of their respective monitoring sets, and then goes to Step 3. If this number is less than N, *Agent A* adds all of the related agents to its monitoring set and then jumps to Step 4.

3. *Agent A* picks the N related agents with the lowest monitoring-set cardinalities to be in its monitoring set.

4. *Agent A* sends messages to all the other non-related agents, requesting their monitoring-set cardinalities. (This can be done using a single broadcast message). *Agent A* then iteratively selects agents with the lowest monitoring-set cardinalities until either (a) it has N agents in its monitoring set or (b) until all the agents have been exhausted (i.e. there are less than N agents in the whole organization).

Finally, once *Agent A* has determined its monitoring set, it can send a message to each of the agents in its set requesting them to monitor its health. In addition to being distributed, other advantages of this algorithm are: (a) Steps 1–3, can be piggy-backed onto the coordination-mechanism negotiation messages exchanged with the agents in the related set and (b) This scheme will reduce the frequency of *are-you-alive* messages transmitted since the agents will be communicating in-band as a part of their normal tasks processing.

3.4.7.4 Augmenting the robustness capacity of an organization

The robustness capacity of an organization is defined as the number of agent failures that an organization can withstand. The robustness capacity is equal to the kill count minus 1, where the kill count is the *minimum* number of agents that need to be killed in order to kill the organization (i.e. ensure that the remaining agents cannot complete tasks without having to restart more agents).

The robustness capacity of an organization is dependent on (a) the underlying global task structure and (b) the way it has been divided amongst the agents. The CAFs of the global task structure, especially the root CAF, determines the number of alternatives available for achieving a task. For example, a one-level deep task structure with a *MAX* CAF and three subtasks would have three alternative ways of achieving the task. If each of these three alternatives was divided amongst three agents, the resultant organization would have a kill count of 3 and a robustness capacity of 2.

Figure 3.8 shows how the global task structure and its breakup amongst the agents affects the robustness capacity of an organization. The first task structure has a *MIN* CAF as its root, so *either* of Agents 1, 2 or 3 can be killed in order to kill the organization. The kill count is 1 and, hence, the robustness capacity is 0. The second task structure has a *SUM* as its root, so *all* the three agents need to be killed in order to kill the organization. Hence, the kill count is 3 and the robustness capacity is two. The third task structure is similar to the second one except that it has an enablement from Method J to F. With the task structure divided amongst the agents as shown, if Agent 3 is killed, there is no way to complete Method J. This, in turn, means that Method F will never be enabled, i.e. the quality of Method F will always be 0. Since Task B has a *MIN* CAF, the quality of Task B will also be 0 and, as a result, Agent 1 has effectively been poisoned. Hence, only Agents 2 and 3 need to be killed to kill the organization and the kill count is 2.

Augmenting the robustness capacity of an organization is the process of adding agents to the organization so as to increase its kill count. Again, we are assuming that the desired kill count, K is an input to the organization. A trivial

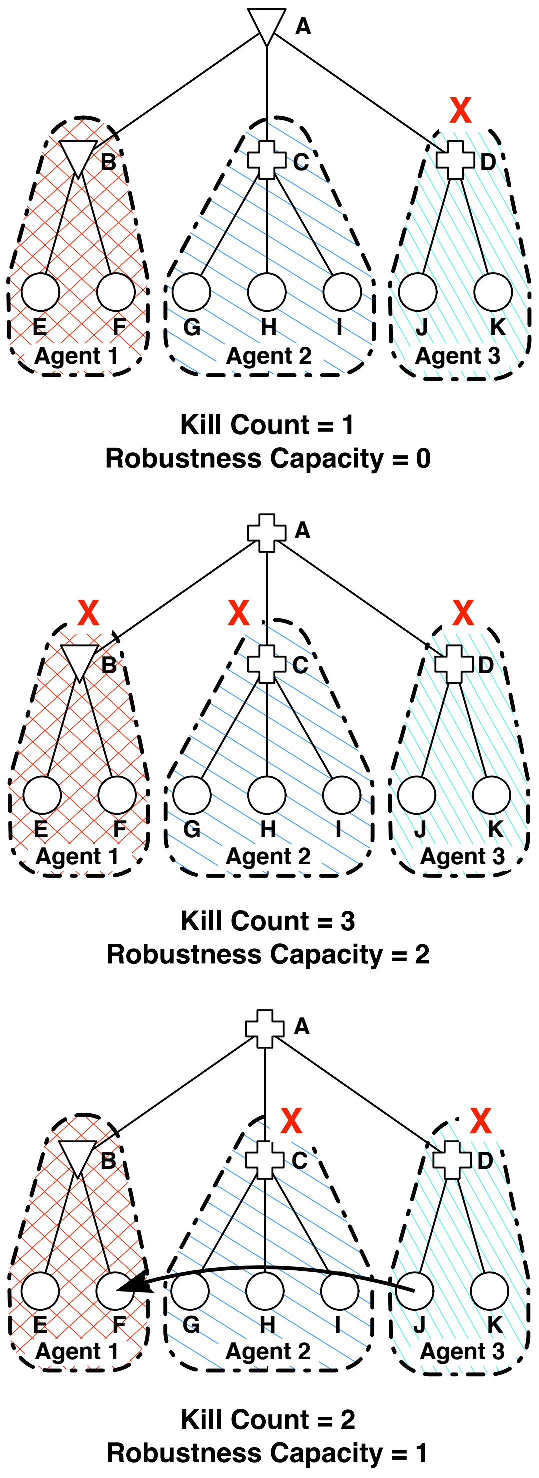


Figure 3.8: **Computing Robustness Capacity:** Figure showing how the global task structure and its breakup amongst the agents affects the robustness capacity. The X's on the agents show which agents need to be killed in order to kill the organization.

way to do this would be to replicate each agent *K-1* times. However, this would be inefficient as it does not take into account the existing kill-count of the organization.

The first step towards increasing the robustness capacity of an organization would be to compute the existing kill-count, and then to "add" agents by breaking up the global task structure and spawning agents in a way that increments this kill-count. Unfortunately, the bad news is that computing the kill-count of an organization based on an underlying TÆMS task structure is NP-hard (This result is similar to the findings in [Zhang *et al.* , 2009]). An informal proof follows:

This proof is based on the reduction of a minimum set covering problem to a TÆMS based organization, where the kill-count of that organization would be the solution to this problem. Assume a ground set M consisting of m elements, $\{e_1, e_2, ..., e_m\}$ and n subsets $\{s_1, s_2, ..., s_n\}$. Create a TÆMS task structure, with a *MAX* CAF as the root and the subsets, $\{s_1, s_2, ..., s_n\}$ as its subtask nodes. Finally replace each node s_i with a *MIN* CAF task, the subtasks of which will be the methods, $\{e_{i,1}, e_{i,2}, ..., e_{i,j}\}$, where each method corresponds to an element of s_i. Finally, assign m agents to the organization, where each method corresponding to e_i is assigned to agent a_i. This reduction will provably take polynomial time.

Since, (a) the problem of computing the kill count/robustness capacity of a problem is NP-hard and (b) augmenting the organization by spawning agents at specific places will interfere with other desirable characteristics such as balancing the execution time and maximizing quality, we chose an alternative approach to augmenting the robustness capacity.

In our approach, the initial root node of the global task view is cloned $K-1$ times and each clone is allocated to a separate agent. These agents are responsible for individually forming their own independent organizations and spawning and composing with agents independently.

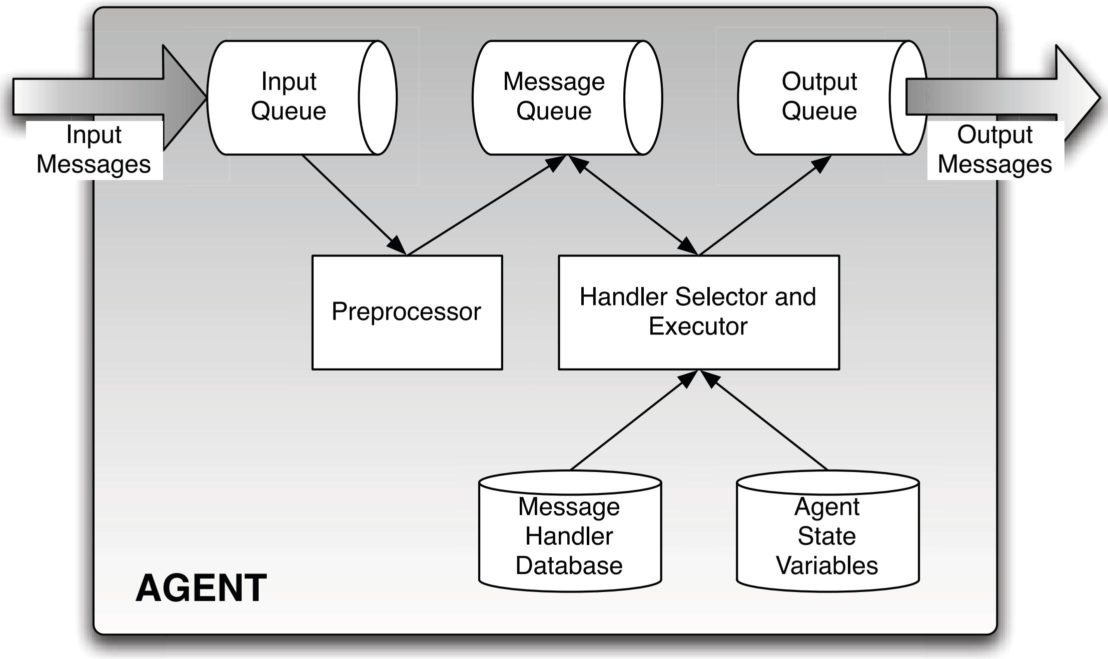


Figure 3.9: The generic architecture of the agents.

3.4.7.5 Frequency of *are-you-alive* messages

Ideally, we want each agent in the monitoring set of an agent A to send an *are-you-alive* request at a different time. To achieve this, we initialize each agent with a random seed. The *next-poll-time* is initialized to the *poll-interval* plus this random seed. Also the next-poll-time is recalculated on receiving *any* message from the monitored agent.

3.5 Agent Architecture

We have implemented a generic, message-driven architecture for our agents as shown in Figure 3.9. The agents receive input messages from the environment and other agents which are buffered in the input queue. At the start of each execution cycle (clock tick), the input messages are pre-processed and transferred to the message queue. The handler selector, then, dequeues a message from the message queue, determines the message type and then based on the message type finds one or more message handlers that are capable of processing messages of the given type. Each message handler is then called in turn with the message. This process is repeated until there are no more messages left on the message queue. The message handlers may generate new messages that are either enqueued in the message queue or the output queue (if they are meant for other agents or for the environment.)

The logical view of the OSD agents, showing the flow of messages and other data between the various message handlers is shown in Figure 3.10. A description of each of these message handlers is given below:

Initializer: The initializer is called once, per agent, in response to an *INITIALIZE* message from the environment, and is responsible for setting up the agent's organizational knowledge.

Scheduler and CM Selector: The scheduler and CM Selector is responsible for (a) selecting a local schedule for the agent and (b) negotiating a coordination mechanism with agents in this agent's related set.

Coordinator: The coordinator receives (a) *DO_TASK* messages from the environment and other agents; and (b) *COORD_RESULTS* messages from other agents. The *DO_TASK* message are used by (1) the environment to announce the arrival of new tasks to the agent; and by (2) other agents to request that this agent perform a task. (For example, the clone selector will select amongst all the clones and generate this message.) The *COORD_RESULT* messages are used to send results to other agents that have an NLE with tasks that are the responsibility of this agent.

Method Executor: As evident by its name, the method executor is responsible for picking up the next method to execute and sending it off to the environment for execution.

Agent Spawner: The Agent Spawner is responsible for spawning off new agents. To do this, the spawner constantly monitors the outstanding methods in the

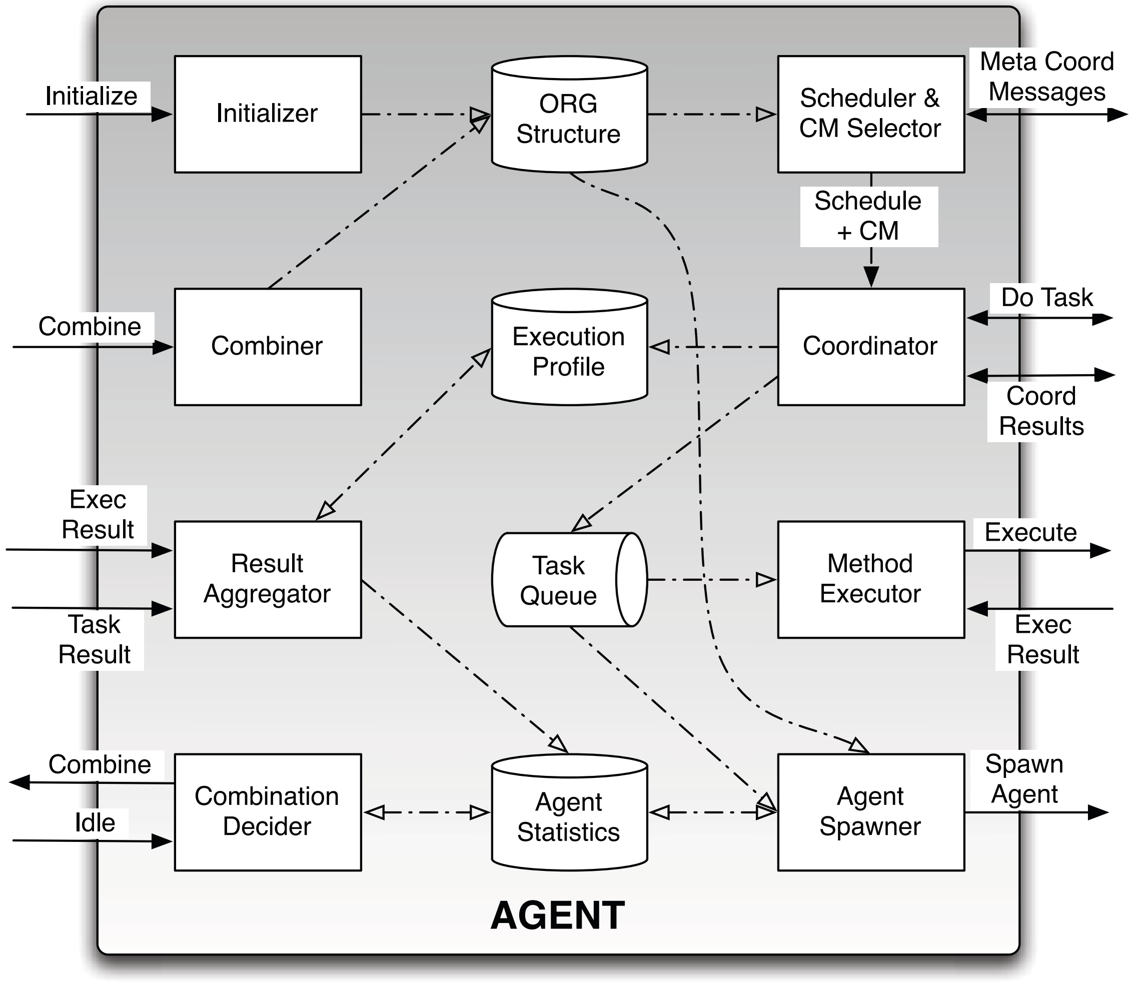


Figure 3.10: The logical view of the OSD agents showing the flow of messages and data between the various message handlers. The solid arrows show message flow and dashed arrows show other data flow.

agent's task queue to see if the agent can complete all of them by their respective deadlines. If not (See Section 3.4.6), it spawns off one or more new agents. To do this it selects a node for breakup/cloning (See Section 3.4.5.1) and applies either the breakup or the cloning operator depending on the spawning strategy (See Section 3.4.5.2). It then sends a *SPAWN_AGENT* message to the environment to create a new agent.

Result Aggregator: The result aggregator is responsible for determining the execution characteristics of a task from the execution characteristics of its subtasks. The result aggregator also sends a *TASK_RESULT* message to the parent agents once all the outstanding subtasks of a task have been executed.

Combination Decider: The combination decider gets called when the agent has remained idle for a given number of cycles. It uses simulated annealing to determine if the agent should be combined with another agent (See Section 3.4.6). If it decides in the affirmative, it sends a *COMBINE* message to the composed agent.

Combiner: This module performs the actual agent composition and is called in response to a *COMBINE* message from the composer agent. It calls the merging operator defined in Section 3.4.1.

3.6 Chapter Summary

This chapter forms the core of this thesis and describes our approach to Organizational Self-Design.

Section 3.1 starts off this chapter with a description of task structures and motivates the use of TÆMS as our problem representation language. Section 3.2, then briefly describes our overall approach to OSD and introduces the reader to our two organizational design primitives — agent spawning and agent composition.

This is followed by a formal description of our task and resource model in Section 3.3. This section includes a description of our execution model (in Section 3.3.1) and a formal definition of the Characteristic Accumulation Functions (CAFs) (in Section 3.3.2) and the Non-Local-Effects (in Section 3.3.3). Section 3.3.4 introduces the reader to some extensions to the TÆMS task representation language that we proposed and implemented for our research. Finally, Section 3.3.5 describes some assumptions that we made about the task and resource model in our research.

This chapter then moves on to an actual description of the OSD approach in Section 3.4. This section includes a description of the agent roles and relationships in Section 3.4.1. This is followed by an illustration of the coordination mechanism employed (in Section 3.4.2) and norms (in Section 3.4.3). Next, we formally define OSD in Section 3.4.4.

Section 3.4.5 then describes the mechanism and algorithm used to form and adapt an organizational structure. This includes a description of our task allocation heuristics in Section 3.4.5.1 and the spawning strategies in Section 3.4.5.2.

Section 3.4.6 describes how the agents might detect a need for organizational change, including a need for agent spawning (in Section 3.4.6.1) and a need for agent composition (in Section 3.4.6.2).

This is followed by a description of our robustness mechanisms in Section 3.4.7. The citizens approach to robustness is described in Section 3.4.7.1 while the survivalist approach to robustness is described in Section 3.4.7.2.

Finally, we end this chapter with a description of our agent architecture in Section 3.5.

Chapter 4

EXPERIMENTS IN THE USE OF ORGANIZATIONAL SELF-DESIGN

The true method of knowledge is experiment. (William Blake)

A theory is something nobody believes, except the person who made it. An experiment is something everybody believes, except the person who made it. (Albert Einstein)

It doesn't matter how beautiful your theory is, it doesn't matter how smart you are. If it doesn't agree with experiment, it's wrong
(Richard Feynman)

This chapter is divided into four major sections — First, I compare the performance of my approach with the Contract Net approach, the most commonly used one-off mechanism for organization and coordination[1]. Next, I evaluate the performance of the three different task allocation heuristics outlined in Section 3.4.5.1 against a random task allocation strategy. I follow this up with an evaluation of the five spawning strategies presented in Section 4.4. Finally, I test the robustness of my approach.

[1] Refer to Section 2.2.5.1 for details on the Contract Net Protocol.

4.1 Comparison with the Contract Net Protocol

To evaluate our approach, we ran a series of experiments that simulated the operation of both the OSD agents and the Contract Net agents on various task structures with varied arrival rates and deadlines. At the start of each experiment, a random TÆMS task structure was generated with a specified depth and branching factor[2]. During the course of the experiment, a series of task instances (problems) arrive at the organization and must be completed by the agents before their specified deadlines.

To directly compare the OSD approach with the Contract Net approach, each experiment was repeated several times — using OSD agents on the first run and a different number of Contract Net agents on each subsequent run. We were careful to use the same task structure, task arrival times, task deadlines and random numbers for each of these trials.

We divided the experiments into two groups: experiments in which the environment was static (fixed task arrival rates and deadlines) and experiments in which the environment was dynamic (varying arrival rates and/or deadlines).

The two graphs in Figure 4.1, show the average performance of the OSD organization against the Contract Net organizations with 8, 10, 12 and 14 agents. The results shown are the averages of running 40 experiments. 20 of those experiments had a static environment with a fixed task arrival time of 15 cycles and a deadline window of 20 cycles. The remaining 20 experiments had a varying task arrival rate - the task arrival rate was changed from 15 cycles to 30 cycles and back to 15 cycles after every 20 tasks. In all the experiments, the task structures were randomly generated with a maximum depth of 4 and a maximum branching factor of 3. Note that (a) the size (depth and branching factor) of these task structures

[2] We chose to use randomly generated task structures because the only available repository of task structures is from the DARPA coordinators project and these are unsuitable for our purposes.

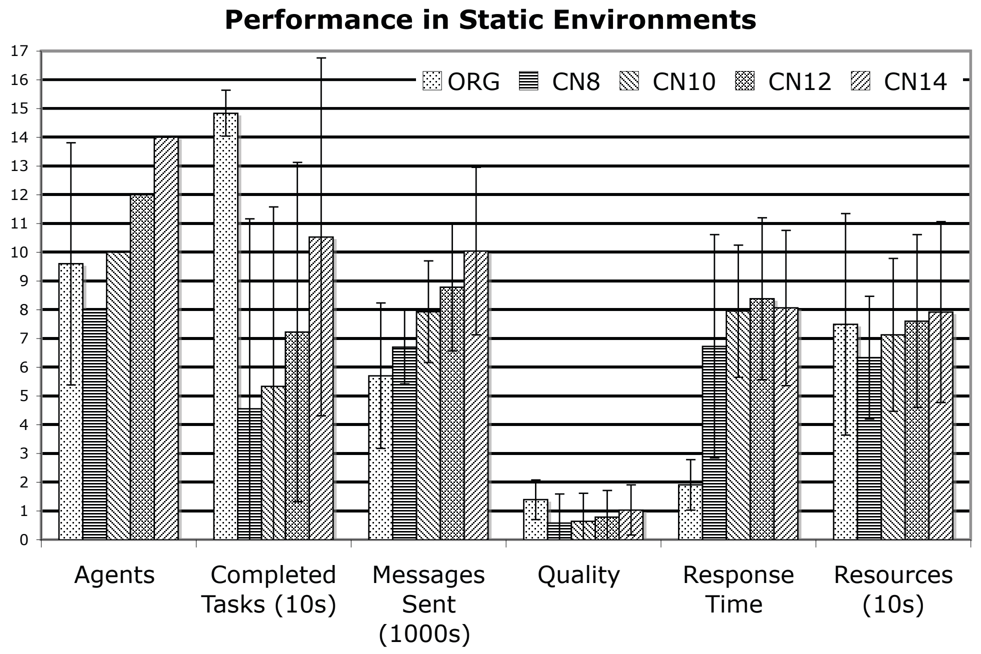
is greater than the typical size of the task structures for the real world applications that we looked at; and (b) TÆMS can be used to represent task structures at various levels of abstraction — so any method of a task structure can be further refined by breaking it up into its component tasks and methods. Hence, by considering task structures of a particular size, we are looking at problems at a particular level of abstraction. The runtime for all the experiments was 2500 cycles.

Graphs 4.2a, 4.2b, 4.2c and 4.2d show the variation in the various measured characteristics over time for a randomly chosen experiment under static environmental conditions for a range of performance characteristics. Similarly, graphs 4.3a, 4.3b, 4.3c and 4.3d demonstrate the performance of the various groups of agents under dynamic conditions.

We tested several hypotheses relating to the comparative performance of our OSD approach using the *Wilcoxon Matched-Pair Signed-Rank* tests [Daniel, 2000]. *Matched-Pair* signifies that we are comparing the performance of each system on precisely the same randomized task set within each separate experiment. The tested hypothesis are:

1. **The OSD organization requires fewer agents to complete an equal or larger number of tasks when compared to the Contract Net organization:** To test this hypothesis, we tested the stronger null hypothesis that states that the contract net agents complete more tasks. This null hypothesis is rejected for all contract net organizations with less than 14 agents (static: $p < 0.0003$; dynamic: $p < 0.03$). For large contract net organizations, the number of tasks completed is statistically equivalent to the number completed by the OSD agents, however the *number of agents* used by the OSD organization is *smaller*: 9.59 agents (in the static case) and 7.38 agents (in the dynamic case) versus 14 contract net agents[3]. Thus the original hypothesis,

[3] These values should not be construed as an indication of the scalability of our



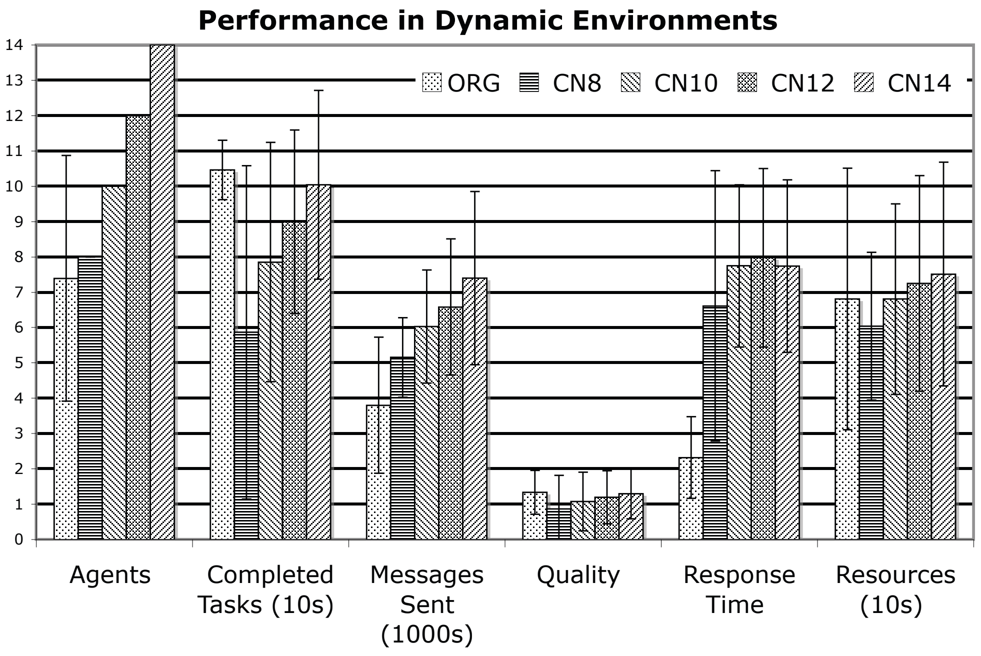


Figure 4.1: **Comparison to the CNP — Average performance over all experiments:** Graph comparing the average performance of the OSD organization with the Contract Net organizations (with 8, 10, 12 and 14 agents). The error bars show the standard deviations.

that OSD requires fewer agents to complete an equal or larger number of tasks, is upheld.

2. **The OSD organizations achieve an equal or greater average quality than the Contract Net organizations:** The null hypothesis is that the Contract Net agents achieve a greater average quality. We can reject the null hypothesis for contract net organizations with less than 12 agents (static: $p < 0.01$; dynamic: $p < 0.05$). For larger contract net organizations, the average quality is statistically equivalent to that achieved by OSD.

3. **The OSD agents have a lower average response time as compared to the Contract Net agents:** The null hypothesis that OSD has the same or higher response time is rejected for all contract net organizations (static: $p < 0.0002$; dynamic: $p < 0.0004$).

4. **The OSD agents send less messages than the Contract Net Agents:** The null hypothesis that OSD sends the same or more messages is rejected for all contract net organizations ($p < .0003$ in all cases except 8 contract net agents in a static environment where $p < 0.02$)

Hence, as demonstrated by the above tests, our agents perform better than the contract net agents as they complete a larger number of tasks, achieve a greater quality and also have a lower response time and communication overhead. These results make intuitive sense given our goals for the OSD approach. We expected the OSD organizations to have a faster average response time and to send fewer messages because the agents in the OSD organization are not spending time and

approach. We have tested our approach on organizations with more than 300 agents, which is significantly greater than the number of agents needed for the kind of applications that we have in mind (i.e. web service choreography, efficient dynamic use of grid computing, distributed information gathering, etc.).

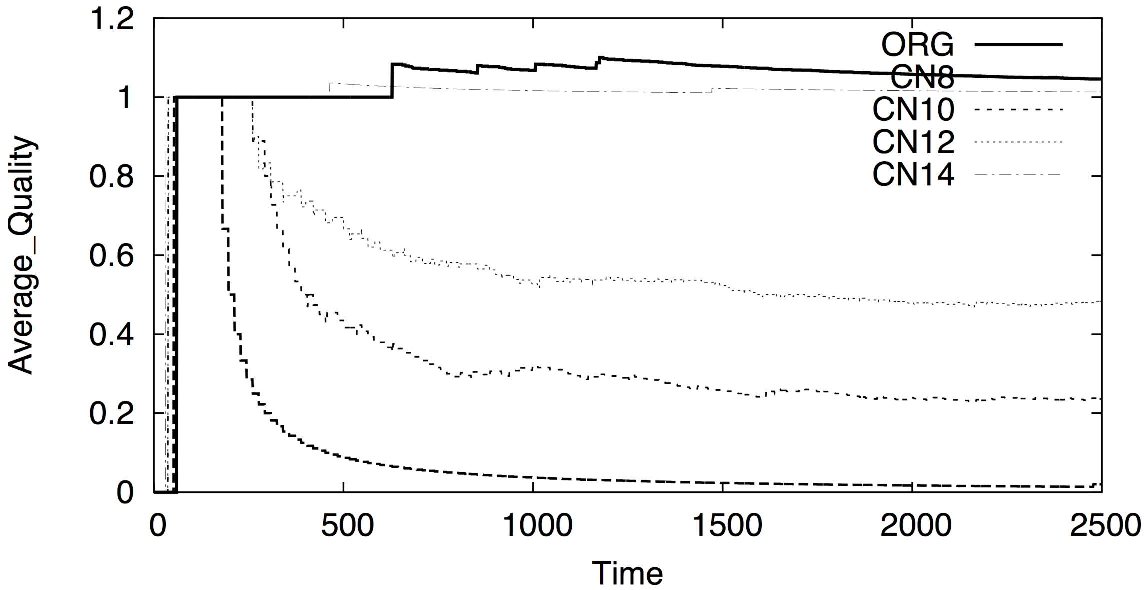
messages sending bid requests and replying to bids. The quality gained on the tasks is directly dependent on the number of tasks completed, hence the more the number of tasks completed, the greater average quality. The results of testing the first hypothesis were slightly more surprising. It appears that due to the inherent inefficiency of the contract net protocol in bidding for each and every task instance, a greater number of agents are needed to complete an equal number of tasks.

4.2 Comparing Apples to Oranges

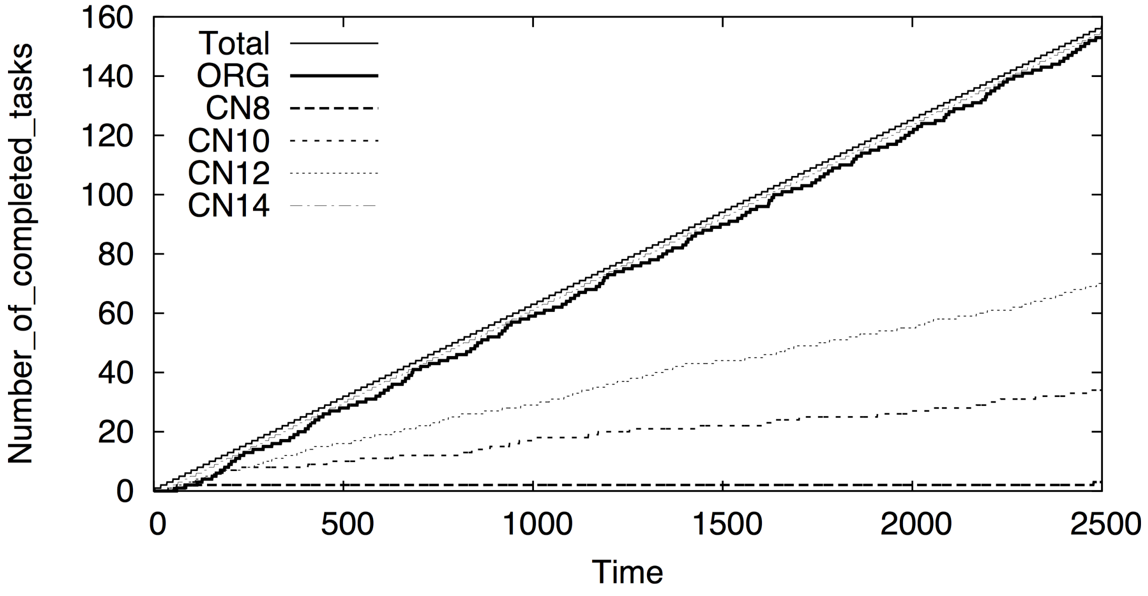
The next two sets of experiments are used to evaluate the various algorithms that form the core of OSD framework — the first set of experiments (in Section 4.3) compares the three task allocation heuristics while the second set (in Section 4.4) is used to evaluate the five agent spawning strategies. These experiments evaluate the performance of these algorithms on randomly generated task structures.

Since the task structures are being randomly generated, two task structures can have vastly varying characteristics (such as the maximum quality achievable, the minimum amount of time needed to accrue positive quality, etc.), which makes a direct comparison of the results difficult. To allow experiments with vastly different task characteristics to be compared, we needed a unified way of reasoning about such task characteristics. Towards this end, we define three terms (See Figure 4.4 for the general idea behind these values):

- The expected *serial-execution-time* (*SET*), is defined as the minimum *expected* duration of time needed for a single agent to perform a task on its own. Due to the presence of NLEs such as *facilitates* and *hinders*, the *SET* is **not** simply the sum of the expected durations of the executable methods of a task. This is because the order in which the methods are executed will affect the amount of time needed to perform a task. To define our *SET* time, we need to order the methods so that the complete execution run takes the minimum amount

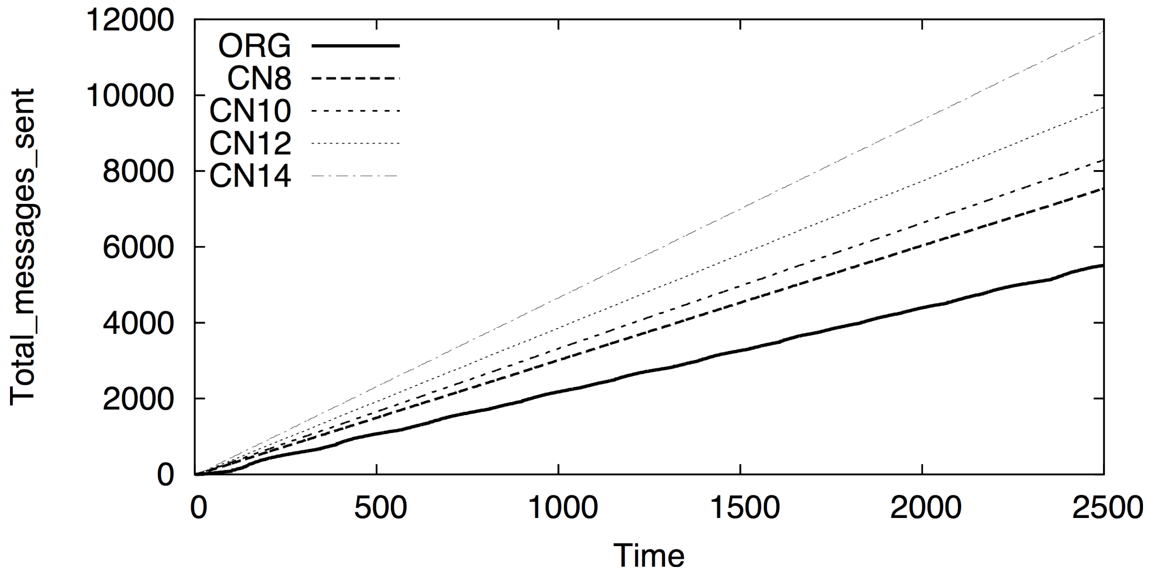


(a) Average Quality Achieved

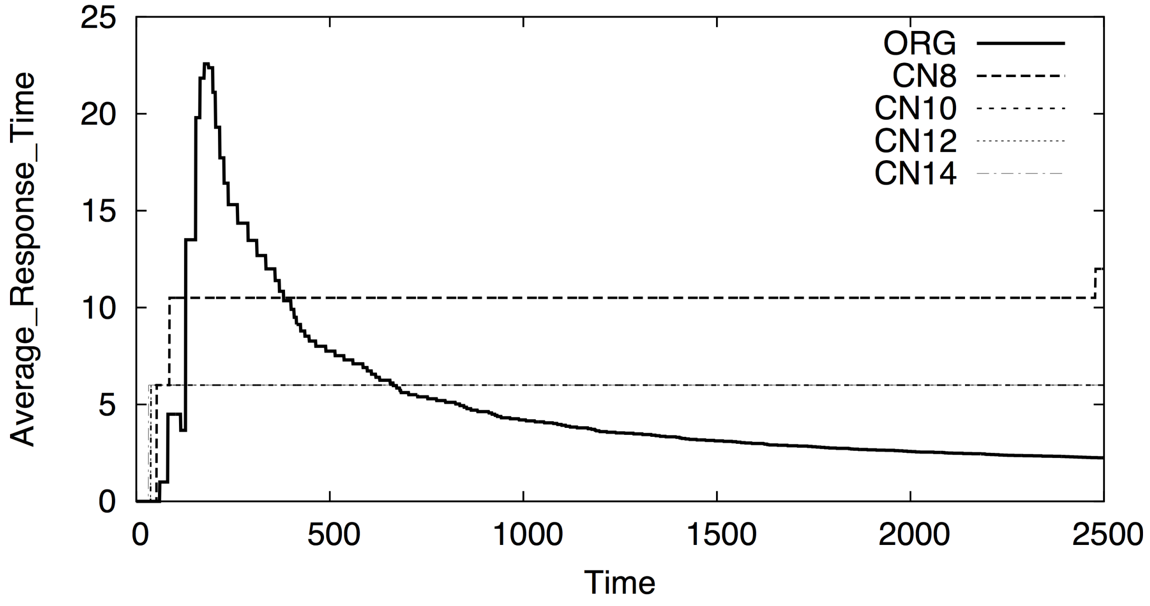


(b) Number of tasks completed

Figure 4.2: **Comparison to the CNP — Performance in static environments:** Graph comparing the performance of our OSD approach against the Contract Net Protocol in a static environment for a single randomly-selected experiment.

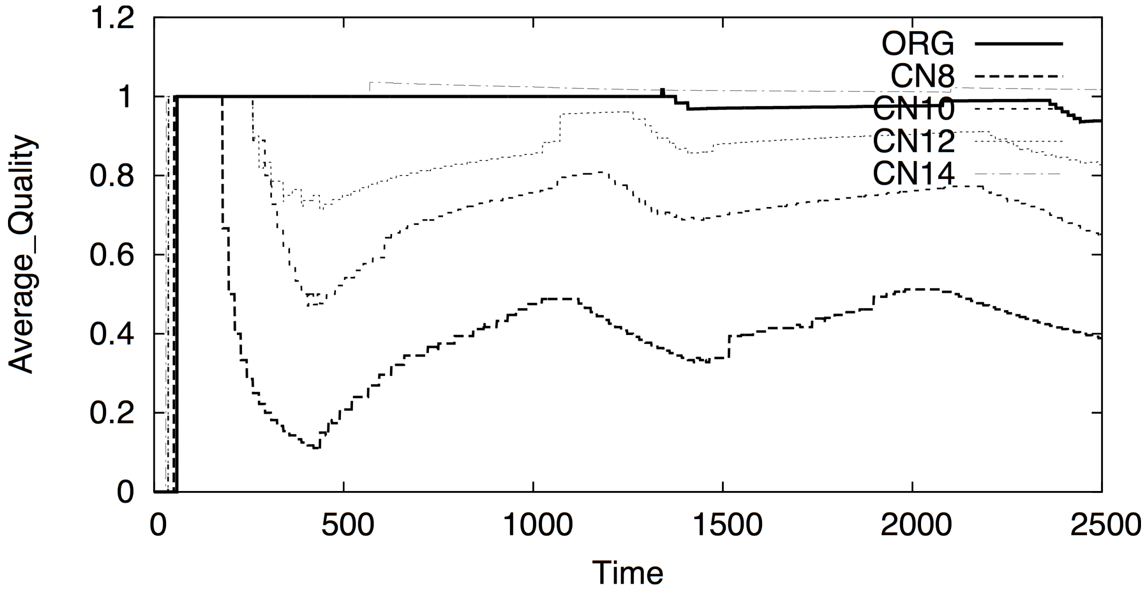


(c) Number of Messages Sent

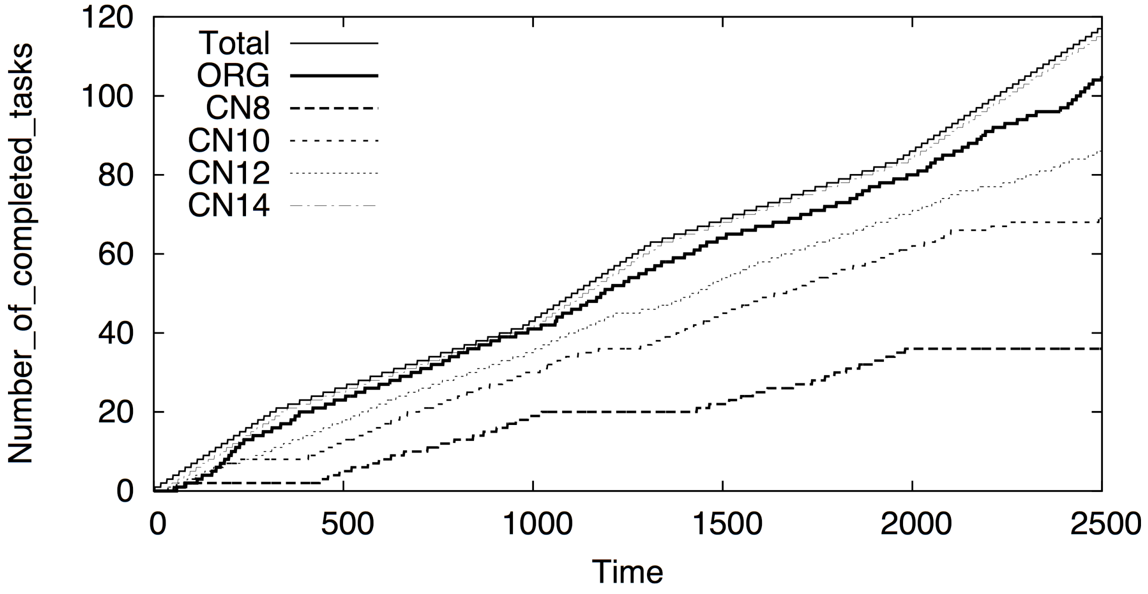


(d) Average Response Time

Figure 4.2: **Comparison to the CNP — Performance in static environments:** Graph comparing the performance of our OSD approach against the Contract Net Protocol in a static environment for a single randomly-selected experiment.

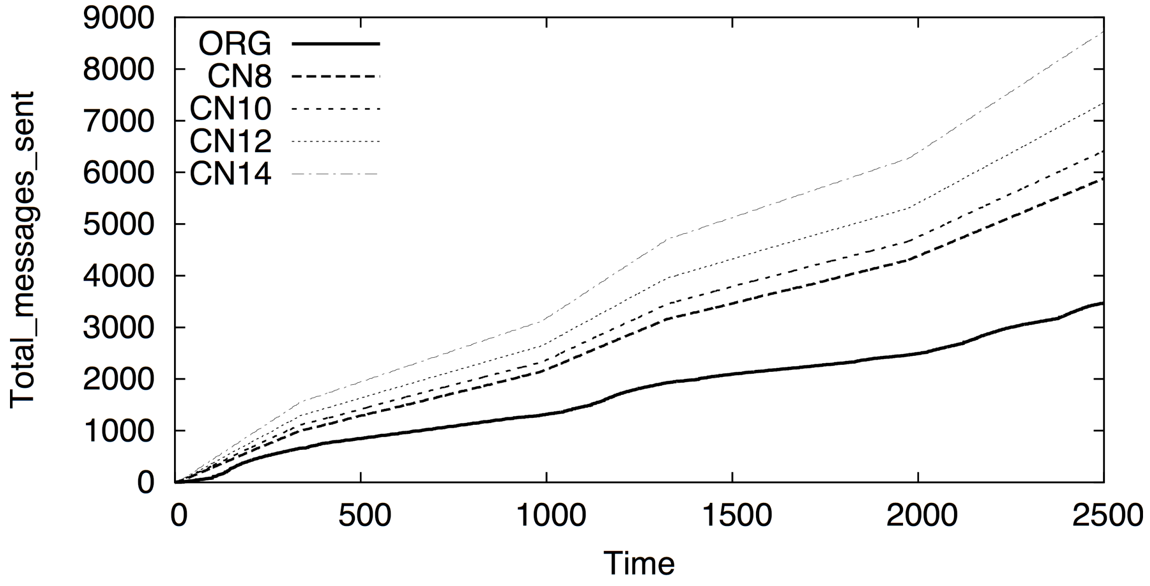


(a) Average Quality Achieved

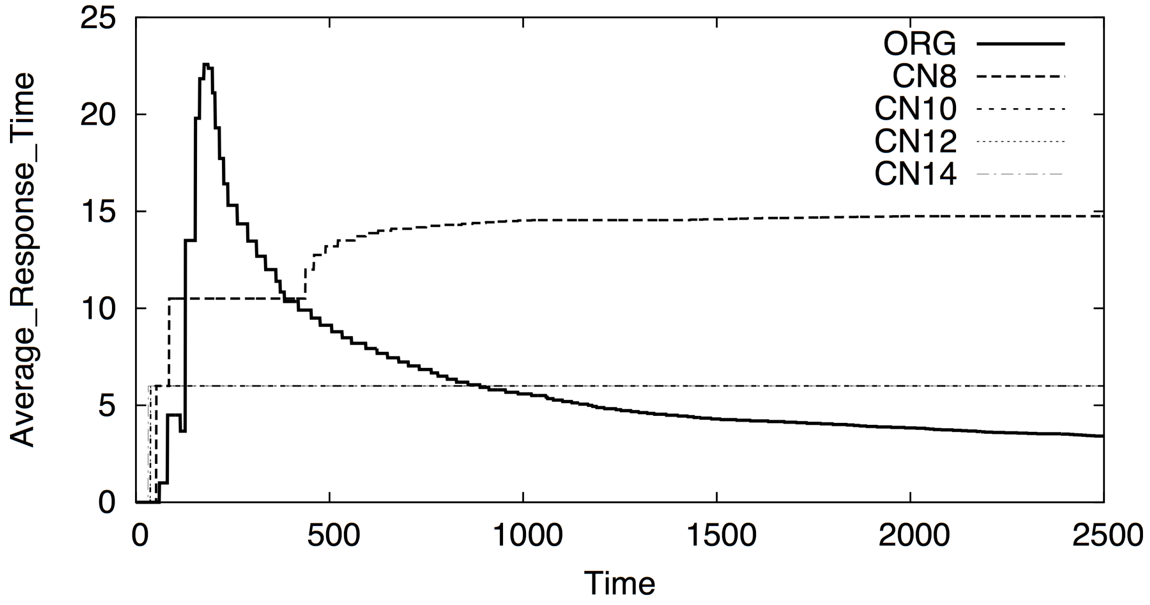


(b) Number of tasks completed

Figure 4.3: **Comparison to the CNP — Performance in dynamic environments:** Graph comparing the performance of our OSD approach against the Contract Net Protocol in a dynamic environment for a single randomly-selected experiment.

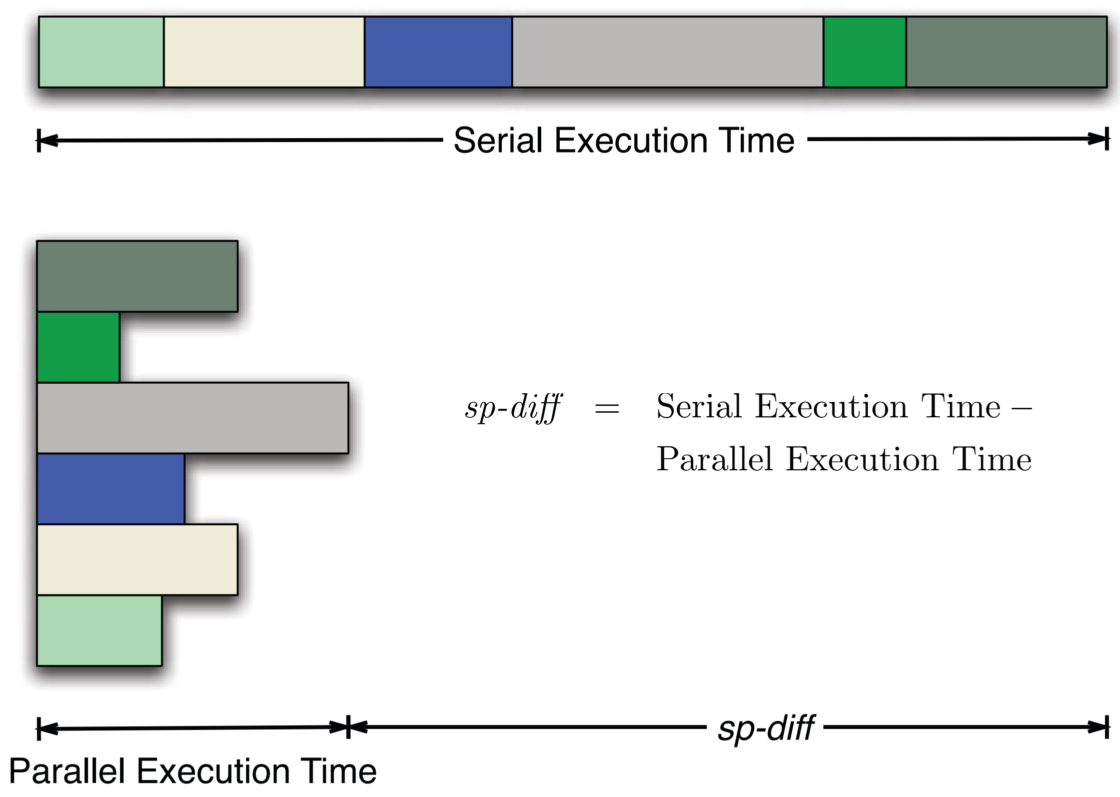


(c) Number of Messages Sent



(d) Average Response Time

Figure 4.3: **Comparison to the CNP — Performance in dynamic environments:** Graph comparing the performance of our OSD approach against the Contract Net Protocol in a dynamic environment for a single randomly-selected experiment.



$$\textit{sp-diff} = \text{Serial Execution Time} - \text{Parallel Execution Time}$$

Figure 4.4: Computation of the *sp-diff* Values

of time possible. This can be done by performing a topological sorting of all the executable methods, taking care to order them so that each method would take the minimum amount to time possible to execute.

- The expected *parallel-execution-time* (*PET*), is the minimum *expected* amount of time needed to perform a task *assuming maximum parallelism*, i.e. each agent is responsible for executing a single method and all the agents can execute methods in parallel. Again computing the *PET* time is not simply a matter of taking the maximum of the executable times of the methods in a task.

- Finally, the *sp-diff* is defined as the difference between the above two times, i.e. *sp-diff* = *SET*−*PET*.

The *sp-diff* value is used to control both the arrival rate of new task instances and the deadlines of the instance themselves as follows:

- The arrival *sp-diff* multiple, $a_{\textit{sp-diff}}$, is used to control the task arrival rate (defined as the rate at which a new task instance is generated). The arrival rate is set to $a_{\textit{sp-diff}} * \textit{sp-diff}$.

- The deadline *sp-diff* multiple, $d_{\textit{sp-diff}}$, is used to set the deadline window, defined as the difference between the task deadline and the arrival time. The deadline window for a task is set to $PET + d_{\textit{sp-diff}} * \textit{sp-diff}$.

4.3 Evaluation of the three task allocation heuristics

4.3.1 Experimental Design

To determine which heuristic performs the best, given a set of task structures, environmental conditions and performance criteria, we performed a series of experiments that were controlled using the following five variables:

- The depth of the task structure was varied from 3 to 5.
- The branching factor was varied from 3 to 5.
- The probability of any given task node having a MIN CAF was varied from 0.0 to 1.0 in increments of 0.2. The probability of any node having a SUM CAF was in turn modified to ensure that the probabilities add up to 1[4].
- The arrival *sp-diff* multiple, $a_{sp\text{-}diff}$, was set to the following values, in turn: 0.05, 0.1, 0.5 and 1.0.
- The deadline *sp-diff* multiple, $d_{sp\text{-}diff}$, was set to 0.1, 0.5 and 1.0.

The total number of NLEs was fixed at 10 and the experiments were run for a total of 2500 cycles.

Each experiment was repeated 20 times, with a new task structure being generated each time — these 20 experiments formed an *experimental set*. Hence, all the experiments in an experimental set had the same values for the exogenous variables that were used to control the experiment. Note that a static environment was used in each of these experiments, as we wanted to see the performance of the arrival rate and deadline slack on each of the three heuristics. The final evaluation was done on 648 experimental sets or 12,960 experiments.

4.3.2 Actual Results

We tested the potential of these three heuristics on the following performance criteria:

[4] Since our preliminary analysis led is to believe that the number of Max and Exactly_One CAFs in a task structure have a minimal effect on the performance of the allocation strategies being evaluated, we set the probabilities of the Max and Exactly_One CAFs to 0 in order to reduce the combinatorial explosion of the full factorial experimental design.

1. The average number of agents used. (*Lower* is better.)

2. The total number of organizational changes. (*Lower* is better.)

3. The total messages sent by all the agents. (*Lower* is better.)

4. The total resource cost of the organization. (*Lower* is better.)

5. The percentage of tasks completed. The percentage of tasks completed is defined as the number of tasks completed divided by the number of tasks generated. (*Higher* is better.)

6. The average quality accrued. The average quality is defined as the total quality accrued during the experimental run divided by the sum of the number of tasks completed and the number of tasks failed. (*Higher* is better.)

7. The average response time of the organization. The response time of a task is defined as the difference between the time at which any agent in the organization starts working on the task (the start time) and the time at which the task was generated (the generation time). Hence, the response time is equivalent to the wait time. For tasks that are never attempted/started, the response time is set at final runtime minus the generation time. (*Lower* is better.)

8. The average runtime of the tasks attempted by the organization. This time is defined as the difference between the time at which the task completed or failed and the start time. For tasks that were never stated, this time is set to zero. (*Lower* is better.)

9. The turnaround time is defined as the sum of the response time and runtime of a task. (*Lower* is better.)

The average performance of these three heuristics is shown in Figure 4.5.

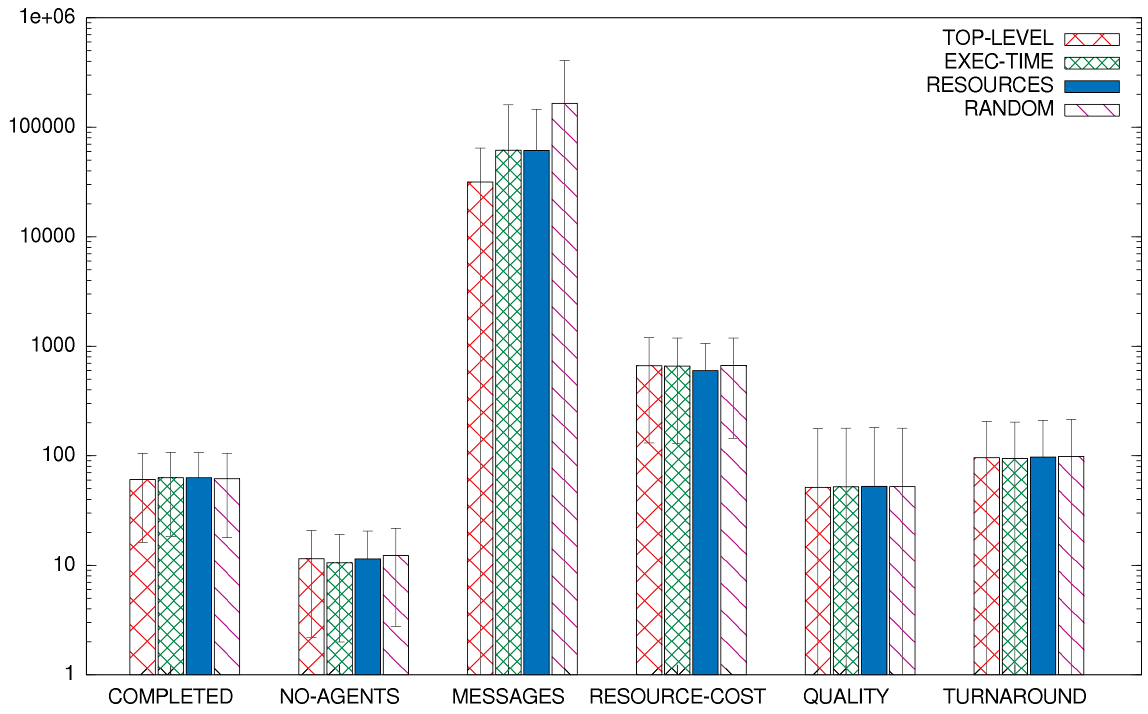


Figure 4.5: Graph showing the average performance of the three breakup heuristics for a variety of measured performance criteria. The y-axis uses a logarithmic scale.

From these results, it is difficult to see much difference between these three heuristics — This graph demonstrates the lack of a clear winner amongst the three heuristics for most of the performance criteria that we evaluated. We suspected this to be the case because different heuristics are better for different task structures and environmental conditions, and since each experiment starts with a different random task structure, we couldn't find one allocation strategy that always dominated the other for all the performance criteria.

To test whether these heuristics were indeed equivalent and to determine the conditions under which one heuristic out-performs the others, we again ran the *Wilcoxon Matched-Pair Signed-Rank* tests on the experiments in each of the experimental sets. The null hypothesis in each case was that there is no difference between the pair of heuristics for the performance criteria under consideration. We were interested in the cases in which we could reject the null hypothesis with 95% confidence ($p < 0.05$). We noted the number of times that a heuristic performed the best or was in a group that performed statistically better than the rest. These counts are given in Tables 4.1 and 4.2.

The number of experimental sets in which each heuristic performed the best or statistically equivalent to the best is shown in Table 4.1. The breakup of these numbers into (1) the number of times that each heuristic performed better than all the other heuristics and (2) the number of times each heuristic was statistically equivalent to another group of heuristics, all of which performed the best, is shown in Table 4.2. Both of these tables allow us to glean important information about the performance of the three heuristics.

Table 4.1: The number of times that each heuristic performed the best or statistically equivalent to the best for each of the performance criteria. Heuristic Key: BET is *Balancing Execution Time*, TF is *Topmost First*, MR is *Minimizing Resources* and Rand is a *random allocation strategy*, in which every TÆMS node has a uniform probability of being selected for allocation.

Criteria/Heuristic	BET	TF	MR	Rand
No-Agents	**620**	277	311	204
No-Org-Changes	**611**	259	165	14
Total-Messages-Sent	196	**571**	97	35
Total-Resource-Cost	302	222	**573**	212
Percent-Completed	**539**	415	443	414
Average-Quality	**537**	330	440	393
Average-Response-Time	**428**	414	331	253
Average-Runtime	**461**	337	362	324
Average-Turnaround-Time	**486**	380	369	296

Table 4.2: Table showing the number of times that each individual heuristic performed the best and the number of times that a certain group of statistically equivalent heuristics performed the best. Only the more interesting heuristic groupings are shown. *All* shows the number of experimental sets in which there was no statistical difference between the three heuristics and a random allocation strategy

Criteria/Heuristic	BET	TF	MR	Rand	BET+TF	BET+MR	TF+MR	BET+TF+MR	All
No-Agents	**315**	0	24	1	10	14	1	80	184
No-Org-Changes	**332**	13	24	0	136	30	0	99	10
Total-Messages-Sent	54	**390**	16	4	97	1	40	15	25
Total-Resource-Cost	71	0	**313**	1	3	14	8	27	182
Percent-Completed	135	20	38	17	6	15	8	12	**342**
Average-Quality	165	4	49	25	2	18	10	7	**299**
Average-Response-Time	131	69	37	32	66	1	9	82	**141**
Average-Runtime	174	16	43	39	35	5	22	29	**202**
Average-Turnaround-Time	157	36	42	14	41	20	15	41	**208**

Some interesting observations from these tables are:

- As can be seen, the *Balancing Execution Time* (BET) heuristic appears to perform the best for a number of performance criteria. In particular, the BET heuristic uses the lowest number of agents in 620 out of the 648 experimental sets (First row of Table 4.1). Furthermore, in 315 out of these 620 experimental sets, the BET heuristic performed statistically significantly better than *all* the other heuristics. (In the remaining 305 cases, it was a part of a group of heuristics that were statistically equivalent *and* spawned the lowest number of agents.) This result was as expected because the BET heuristic always tries to bifurcate the agents into two agents that have a more or less equal task load. This results in fewer unevenly loaded agents, which, in turn, results in fewer total agents being used.

  Also as expected, the BET heuristic achieved the lowest number of organizational changes in the largest number of experimental sets (611 out of 648). In fact, it was over twice as good as its second best competitor, *Topmost first* (TF) at 259 experimental sets. This shows that if the agents are conscientious in their initial task allocation, there is a lesser need for organizational change later on, especially for static environments.

- The *Topmost First* (TF) heuristic outperforms all the other heuristics in the number of messages sent — the agents in the resultant organization sent the lowest number of messages in 571 experimental sets. This was almost three times as much as it second best competitor, BET (which performed the best or was in a group that performed the best in 196 out of the 648 experimental sets.) Furthermore, the organizations built using the TF heuristic statistically outperformed all the other heuristics by sending a lower number of messages in 390 experimental sets.

This result can be explained by the fact that most of the messages that are exchanged between the agents are used for inter-agent coordination. Since the TF heuristic breaks up the agents at the top of the task tree, it is equivalent to dividing the organization along functional lines. This results in fewer NLEs being broken up since most of the NLEs are between two methods. This, in turn, results in a lower need for coordination between the agents and, hence, a lower number of messages exchanged between the agents.

- Also, as expected, the *Minimizing Resources* (MR) heuristic had the lowest resource cost amongst all the heuristics. In fact, the MR heuristic resulted in the lowest resource usage in 573 of the 648 experimental sets and in 313 of these 573 experimental sets it statistically outperformed all the other heuristics. This is in-tune with our goals for designing the MR heuristic.

- Whereas, the prima facie evidence suggests that the BET heuristic completes the highest percentage of tasks (in 539 experimental sets), accrues the highest average quality (in 537 experimental sets) and has the lowest response-time (428), runtime (461) and turnaround time (486) amongst all the other heuristics, when we look at the breakup of these numbers in Table 4.2, a slightly different conclusion can be reached.

  For these performance criteria, in the largest number of experimental sets, the three heuristics and a random task allocation scheme are all statistically equivalent. For example, in 342 of the 539 experimental sets in which the BET heuristic completes the largest percentage of tasks, all the heuristics and the random allocation scheme are statistically equivalent. (The BET heuristic does beat all the other heuristics in 135 experimental sets for the percentage of tasks completed.)

The equivalence of these heuristics (and the random task allocation scheme) in a large number of experimental sets suggests that our reorganization algorithm (described in Section 3.4.6) makes the correct decision about when to spawn off agents and when to re-compose them. Since this decision about the need for reorganization is made based on the amount of overload and underload and hence, indirectly on the probability of completing tasks before their deadlines, this algorithm tries to ensure that all of the three heuristics have the same task execution characteristics (such as the quality and turnaround times). Note that these three heuristics still result in different organizations with different organizational characteristics (such as the number of agents spawned and the resource costs).

These results seem to suggest that, in the absence of any other information about the environmental conditions, the BET heuristic should be employed since it employs the lowest number of agents to complete the largest percentage of tasks and outperforms all the other heuristics for a number of performance criteria. (The total number of messages sent and the total resource cost being notable exceptions.)

However, given more information about the possible environmental conditions at runtime, such as the arrival rate and deadline, we can make more informed decisions and tradeoffs about these three heuristics. The graphs in Figures 4.6, 4.7, 4.8, 4.9 and 4.10 show how the number of agents, the percentage of tasks completed, the total number of messages sent, the total resource cost and the average turnaround time varies for different values of $a_{sp\text{-}diff}$ and $d_{sp\text{-}diff}$. (Note these graphs show the results that have been averaged over all the experiments that have the same values of $a_{sp\text{-}diff}$ and $d_{sp\text{-}diff}$. Standard error bars have been omitted to make the graphs clearer.)

These graphs mostly reinforce the lessons learned when looking at the statistical tests:

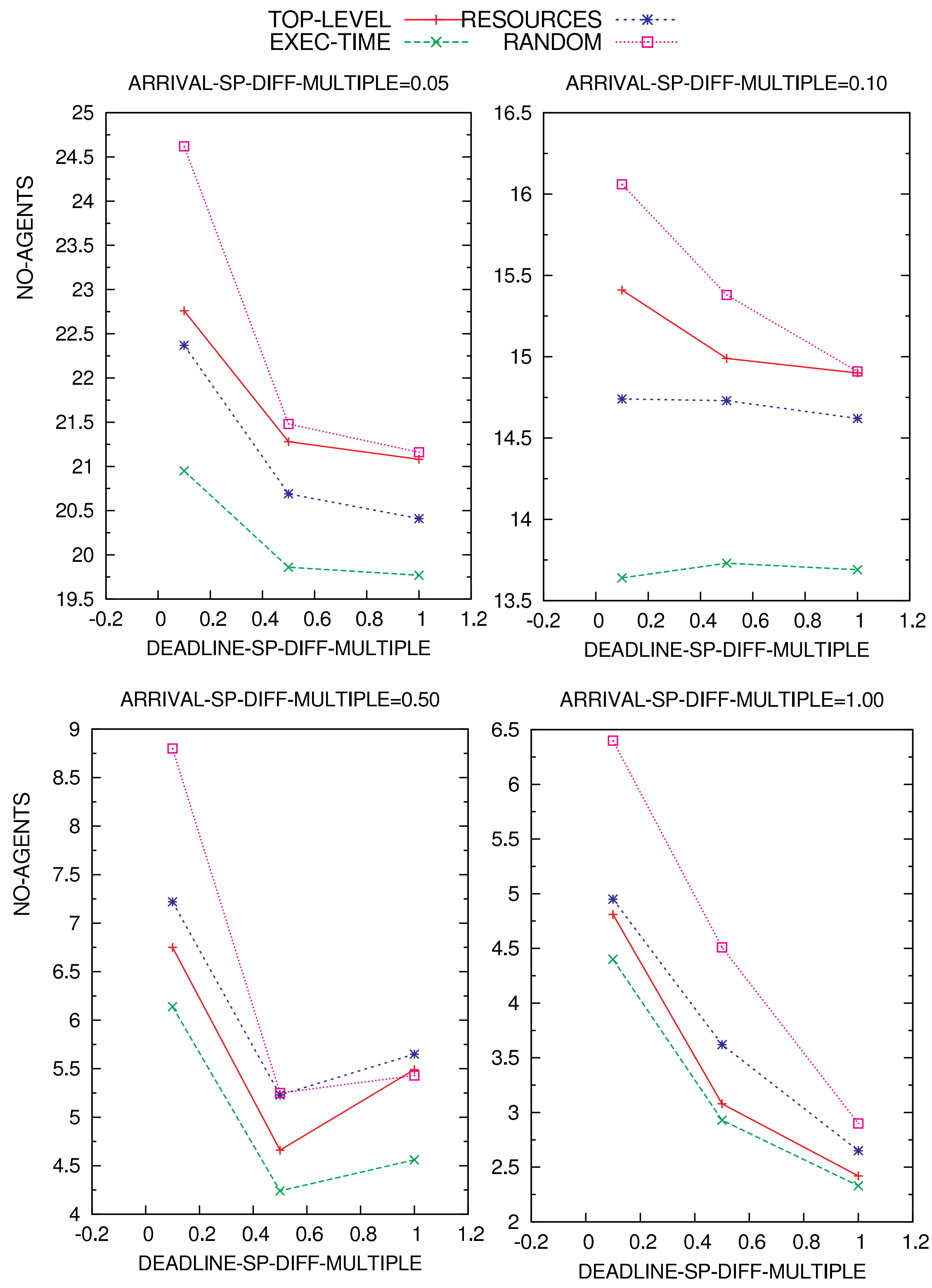


Figure 4.6: Graph showing how the number of agents varies with the arrival and deadline *sp-diff* multiples.

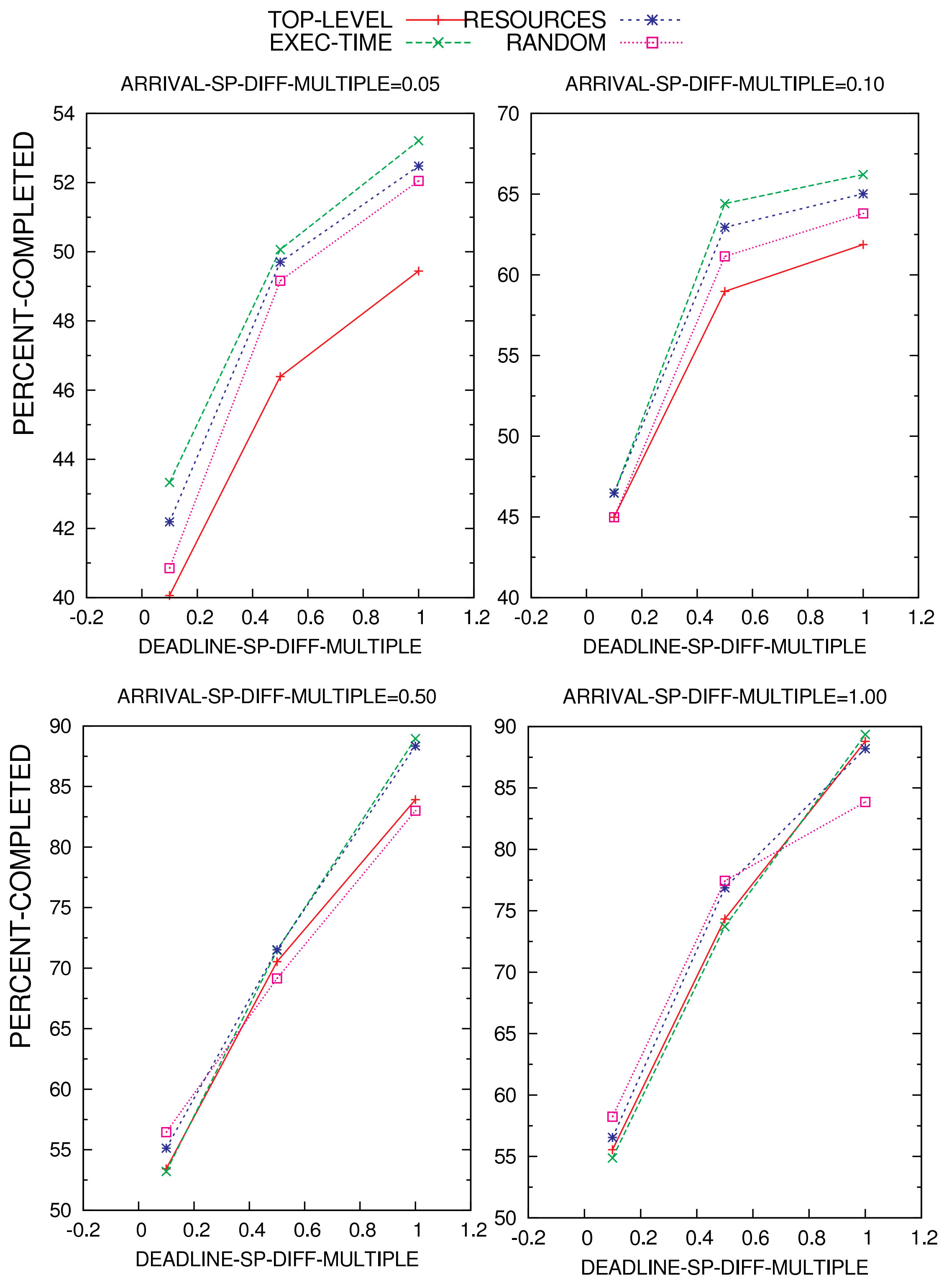


Figure 4.7: Graph showing how the percentage of tasks completed varies with the arrival and deadline *sp-diff* multiples.

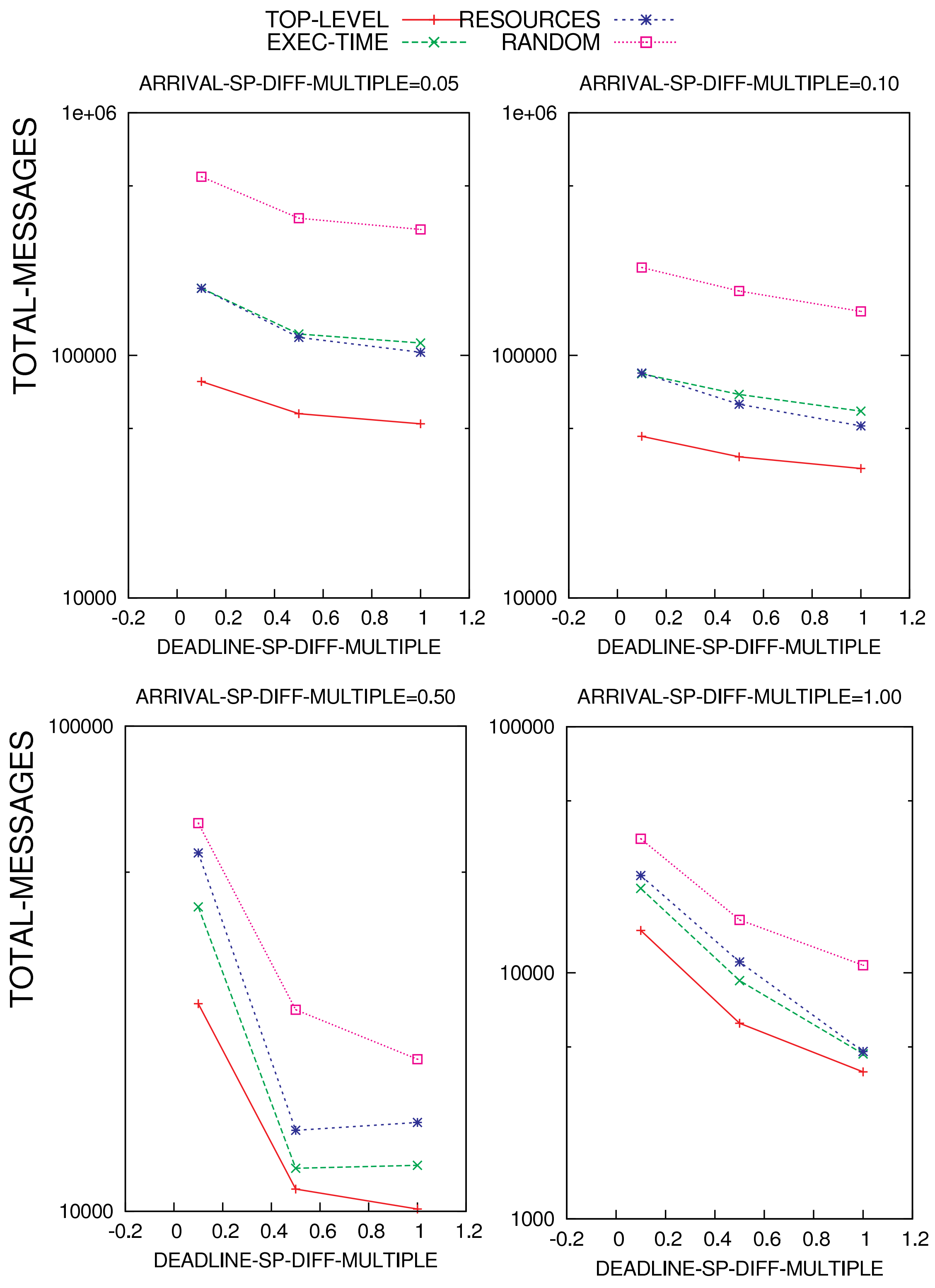


Figure 4.8: Graph showing how the total number of messages sent varies with the arrival and deadline *sp-diff* multiples.

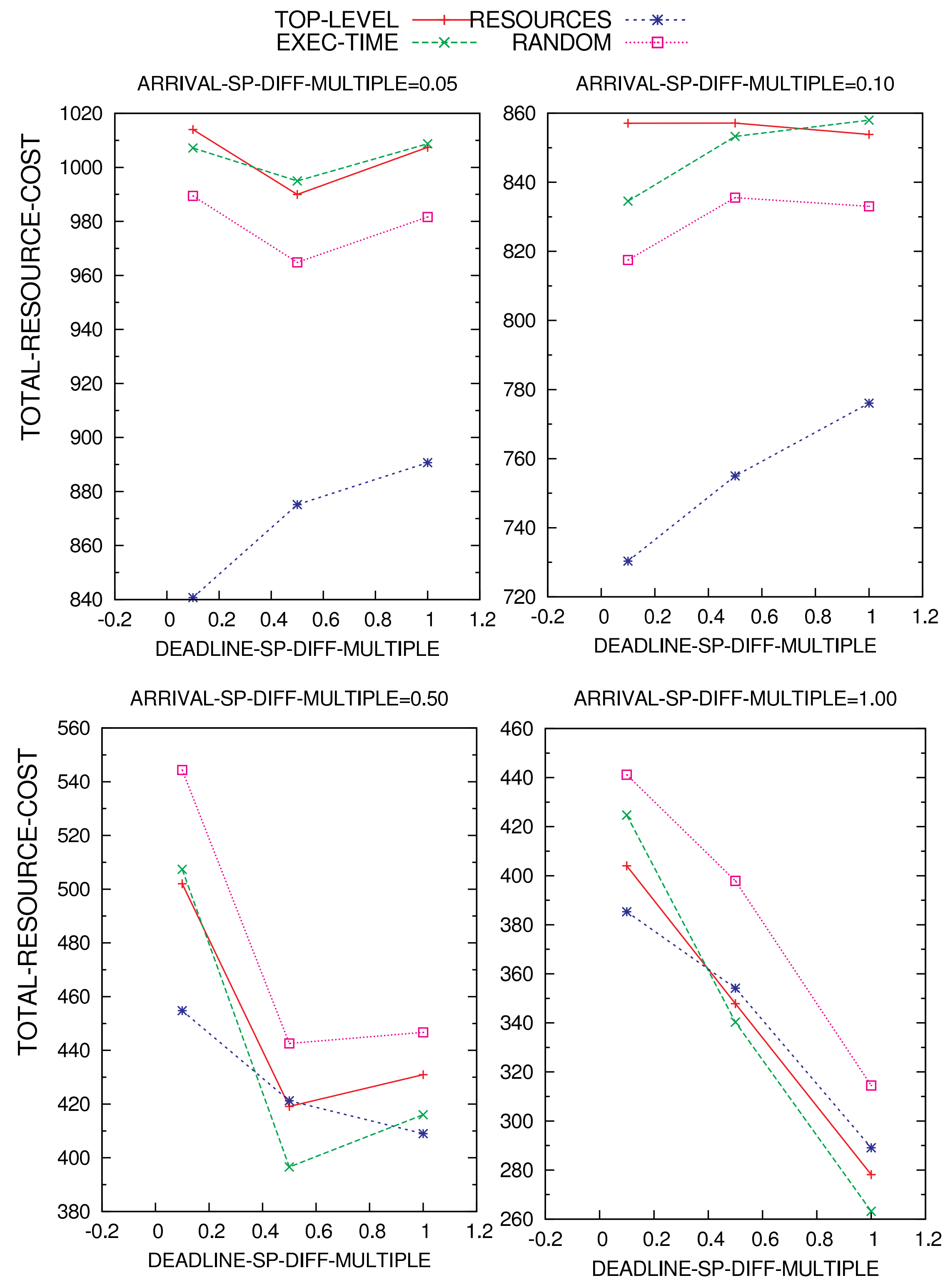


Figure 4.9: Graph showing how the total resource cost varies with the arrival and deadline *sp-diff* multiples.

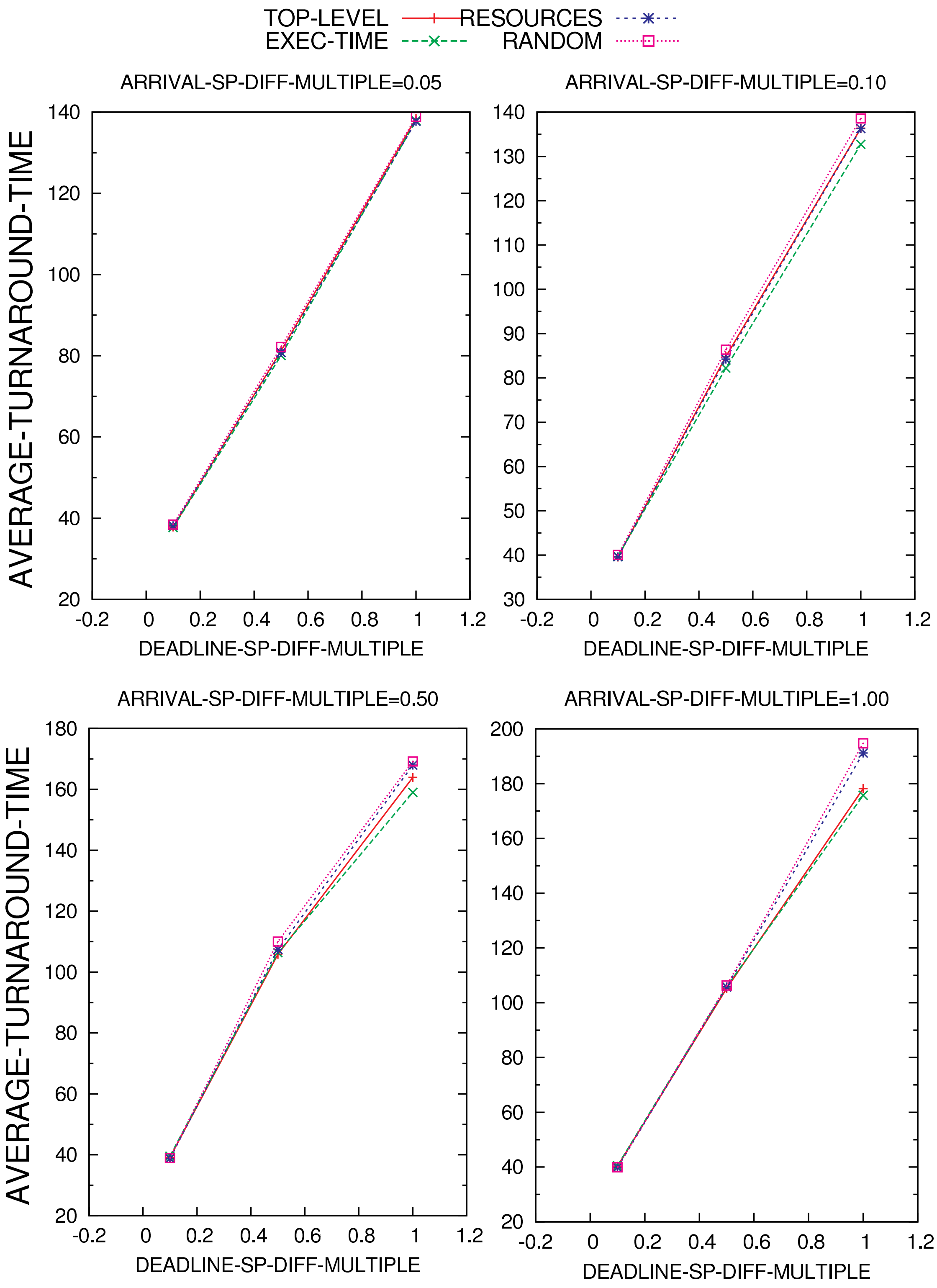


Figure 4.10: Graph showing how the average turnaround time varies with the arrival and deadline *sp-diff* multiples.

- Figure 4.6 shows that the BET heuristic uses fewer agents than the other heuristics, on average, for all the environmental conditions.

  For all three heuristics, as the *sp-diff* multiples increase, the amount of time available to complete a task increases and hence fewer agents are spawned.

- Similarly, Figure 4.8 shows that the TF heuristic sends fewer messages than the other heuristics for all the environmental conditions.

  For all three heuristics: As the arrival *sp-diff* multiple increases, fewer tasks are being generated and hence, fewer messages are exchanged between the agents. Also as the deadline *sp-diff* multiple increases, the deadline pressure on the agents is being reduced resulting in fewer agents being generated. This, in turn, results in fewer messages being exchanged between the agents.

- Figures 4.7 and 4.10 show that the three heuristics, on average, have very similar task execution characteristics (the percentage of tasks completed and the turnaround time.)

  As expected, for all the heuristics, as the arrival *sp-diff* multiple and deadline *sp-diff* multiple increases, the agents have more time to complete tasks and the percentage of tasks completed increases.[5]

  More interestingly, as the deadline *sp-diff* multiple increases, the average turnaround time also increases. This is because as the deadline pressure on the organization decreases, the organization uses fewer agents. Since fewer agents are working in parallel, the turnaround time of the organization increases. Also the average turnaround time seems to be independent of the arrival *sp-diff* multiple.

[5] Note that for extremely low values of the *sp-diff* multiples, completing the tasks is infeasible given the arrival rate and the deadlines of the tasks, which is why the percentage of tasks completed is in the low 40s for such experiments.

- The most surprising results are those of the graph in Figure 4.9. This graph shows that the *Minimizing Resources* (MR) heuristic performs the best for low values of the arrival *sp-diff* multiple but fails to perform as well for high-values of the arrival *sp-diff* multiple.

  We suspect this is the case because when the arrival *sp-diff* multiple is low, a significantly larger number of agents are spawned to deal with the high task arrival rate. This gives the MR heuristic more opportunity to lower the resource usage since the larger number of agents are spawned in a way that minimizes the total resource usage.

  However, for higher values of the arrival *sp-diff* multiple, fewer agents are generated in which case the extremely greedy nature of this heuristic ends up hurting the performance of this algorithm. This is because the MR heuristic prefers "smaller" agents [6] that have a lower resource footprint. However, we suspect that an equivalent count of small agents is not sufficient for completing the tasks by their deadlines, which causes the MR heuristic to generate more agents than the other heuristics. This in turn drives up the total resource cost for the MR heuristic for higher values of the *sp-diff* multiple.

4.4 Evaluation of The Spawning Strategies

4.4.1 Experimental Design

To measure the performance of our strategies, we used the following input variables to control the task structures generated in an experimental set:

1. The arrival *sp-diff* multiple, $a_{sp\text{-}diff}$, was set to the following values: 0.01, 0.1, 0.5 and 1.0.

[6] For this discussion "small" agents are agents that have a low expected duration for their local roles (as calculated by Algorithm 7).

2. The deadline *sp-diff* multiple, $d_{\textit{sp-diff}}$, was set to values ranging from 0.5 to 2.0 in increments of 0.5.

3. The probability of a *MIN* CAF was set to 0.1, 0.5 and 0.9. The probability of a *SUM* CAF was also varied accordingly[7].

4. The maximum number of NLEs ranged from 10 to 20.

We were interested in measuring the following performance criteria:

1. The number of agents spawned by the organization. (*Lower* is better.)

2. The percentage of tasks completed. (*Higher* is better.)

3. The total number of messages sent by the agents in the organization. (*Lower* is better.)

4. The average quality accrued by the organization. (*Higher* is better.)

5. The total resource cost of the organization. (*Lower* is better.)

6. The average turnaround time of the tasks in the organization. (*Lower* is better.)

We ran a total of 96 experimental sets or 1440 experiments. Again, to test for statistically significant differences between the performance of strategies, we ran the *Wilcoxon Matched-Pair Signed-Rank* tests.

4.4.2 Actual Results

The results of these significance tests are shown in Figure 4.11. The vertical bars show the number of times (out of 96, for the 96 experimental sets) that each

[7] Again, we did not consider *MAX* and *EXACTLY_ONE* CAFs for the purposes of these experiments because some preliminary experiments determined that they were not significant contributers to the performance of the strategies.

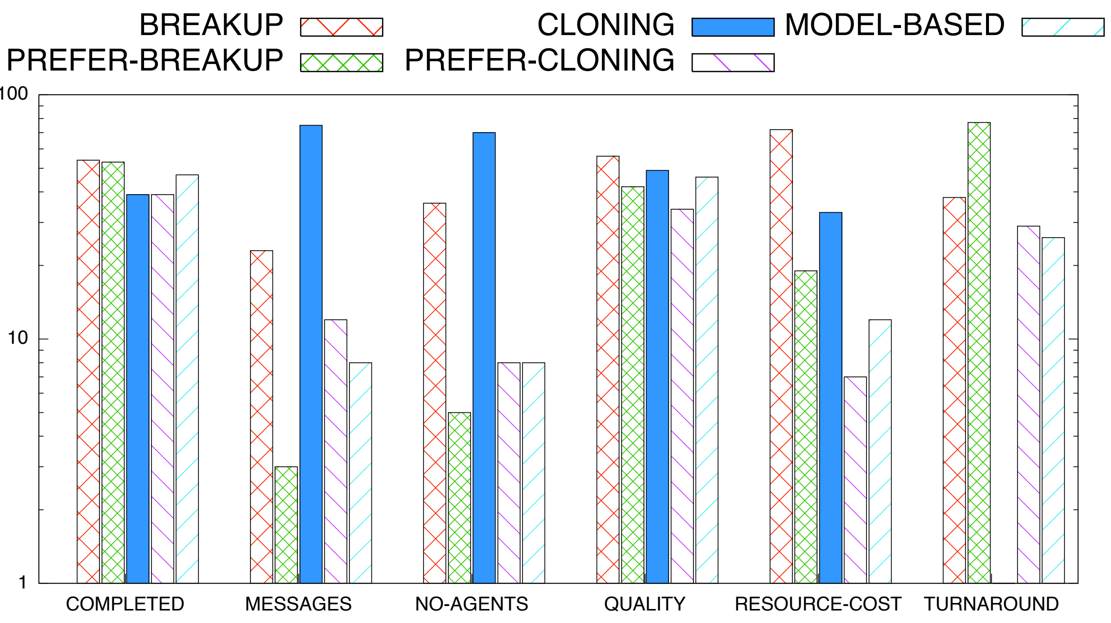


Figure 4.11: **Evaluation of The Spawning Strategies** — Graph showing the number of times each strategy performed the best or was in a group that performed statistically equivalent to the best. The y-axis uses a logarithmic scale.

strategy either performed the best or was statistically equivalent to a strategy that performed the best.

Note that this graph is a summary that shows the number of times a strategy performs as well statistically as the best strategy. Given no other information about the possible environmental conditions at run-time, for a particular performance criteria, we would implement the strategy with the longest bar.

However, to understand the task and environmental conditions that favor a particular spawning strategy, we have to look at the performance of the strategies under different environmental conditions. These performance results are shown in Figure 4.12. These two graphs show the average number of agents and average percentage of tasks completed for different values of $a_{sp\text{-}diff}$ and $d_{sp\text{-}diff}$.

Some interesting observations follow:

- From Fig. 4.11 we can see that the *Breakup* strategy performs well and outperforms most of the other strategies in that it either performs the best or performs statistically equivalent to the best in the percentage of tasks completed, average quality and total resource-cost criteria.

  However, from Fig. 4.12, we can see that *Breakup* performs poorly for extremely low values of $a_{sp\text{-}diff}$ (i.e. 0.01), completing less than 25% of the tasks on average. This is because for such high values of $a_{sp\text{-}diff}$, the number of outstanding tasks is so large that the agents have been maximally broken up. (That is, each agent is performing a single executable method). Since further breakup is infeasible, the agent population is easily overwhelmed and, hence, performs poorly.

- As can be seen from Fig. 4.12, when $d_{sp\text{-}diff} \leq 1$, the *Cloning* strategy performs significantly worse than than the other strategies. This is because when the $d_{sp\text{-}diff} < 1$, the deadline window is shorter than the *SET*, or the amount of time needed by an agent to perform a task on its own[8]. As the $d_{\textit{sp-diff}}$ value increases beyond 1, *Cloning* performs as good as if not better than other strategies.

- The *Cloning* strategy performed the best in the number of agents spawned. This was expected because (a) *Cloning* prefers to create "bigger", yet fewer, agents to perform the same task. The agents are fewer but have more responsibilities; and (b) There were experiments in which the number of agents that could be spawned by *Cloning* was limited to the number of outstanding task instances (i.e. further cloning was infeasible in such experiments.)

  However, despite *Cloning* spawning a fewer number of agents, *Breakup* outperforms *Cloning* in the total resource cost metric. Again this reflects the fact that *Cloning* creates fewer "bigger" agents that use more resources.

- The *Prefer Breakup* and *Prefer Cloning* strategies perform almost as well as the *Breakup* and *Cloning* strategies respectively under environmental conditions where *Breakup* and *Cloning* are feasible and perform well. However, in situations where either *Breakup* or *Cloning* is infeasible, *Prefer Breakup* and *Prefer Cloning* perform better than the approaches on which they are based. This is expected because the *Prefer* strategies are the same as the non-*Prefer* strategies under feasible conditions.

Finally we were very pleased and encouraged by the performance of our *Hybrid Model* based strategy. As can be seen from Fig. 4.11, our *Hybrid Model* performs statistically equivalent to the best strategy in 47 out of the 96 experimental sets (as opposed to 54/96 for the best strategy - *Breakup*.) Also, our *Hybrid Model* performs as well as *Breakup* in conditions of low load ($a_{\textit{sp-diff}} \geq 0.5$)and midway

[8] Note that the percentage of tasks completed is not 0 in such cases. This is because of the presence of *SUM* CAFs in the task structure which allow the organization to accrue some quality.

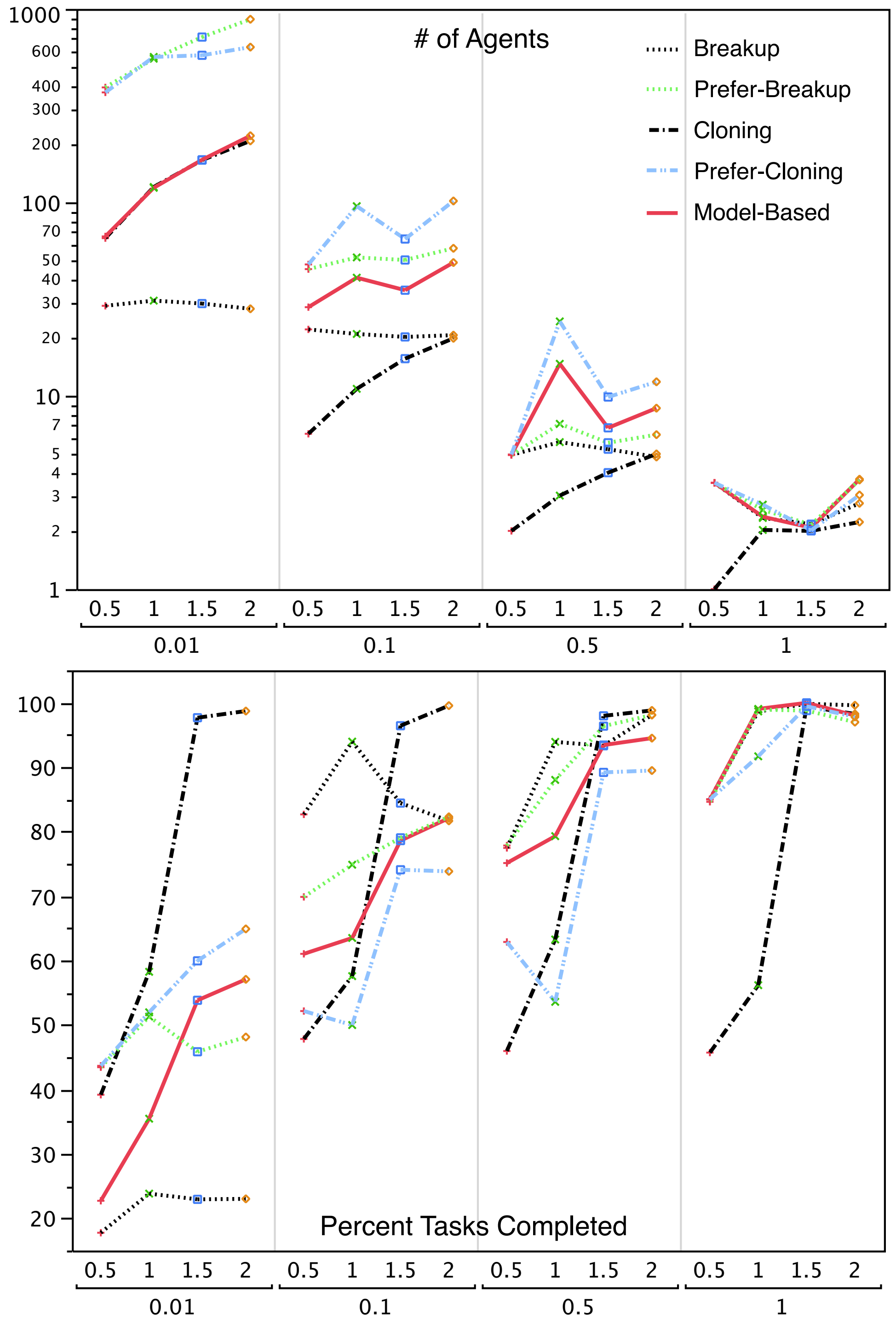


Figure 4.12: **Evaluation of The Spawning Strategies** — Graph showing the *Number of agents* (lower is better) and *Percent Tasks Completed* (higher is better). The x-axis shows the $d_{sp\text{-}diff}$ values (the top numbers) within the outer $a_{sp\text{-}diff}$, values (the bottom numbers).

between *Breakup* and *Cloning* for situations where $a_{\textit{sp-diff}} \leq 0.1$ and $d_{\textit{sp-diff}} \leq 0.5$. In particular, for $a_{\textit{sp-diff}} = 0.01$ and $d_{\textit{sp-diff}} \leq 1$, our *Hybrid Model* completes almost twice as many tasks as *Breakup*, on average. Whereas *Prefer Breakup* performs better than *Hybrid Model* under these conditions, the *Hybrid Model* uses around one-seventh the number of agents. We strongly believe that our hybrid model can be improved by using different values of the constant and thresholds, something that we hope to investigate in our future work.

4.5 Robustness Evaluation

4.5.1 Experimental Design

To evaluate the two robustness mechanisms, we ran a series of experiments that simulated the operation of the OSD organization when those mechanisms were employed. We tested the performance of the *Survivalist* approach against the *Citizens* approach with the following (per agent/per cycle) probabilities of agent failures: 0.000, 0.002, 0.006 and 0.010. Here a probability of 0.006 means that on every clock cycle, each agent has a 0.6 % chance of failing. Note that despite these seemingly low probabilities of failure, the rate of failure is actually greater than can be expected for any real world application. For example, a probability of failure of 0.006 implies that *every* agent can be expected to fail 15 times during a 2500 cycle run.

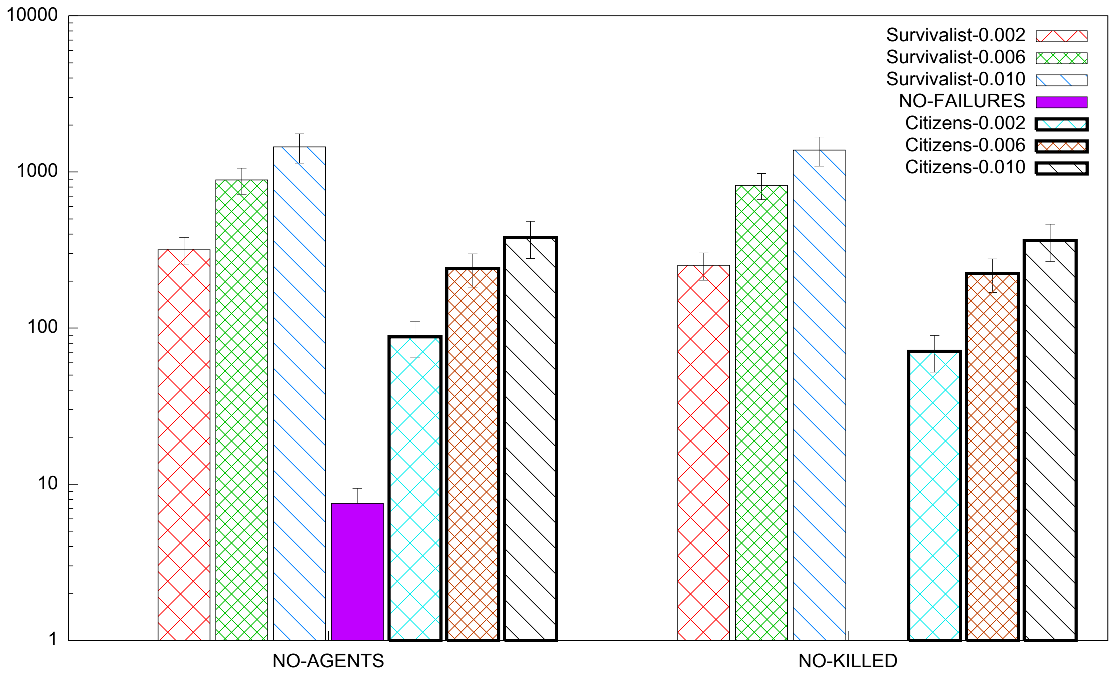
We used a randomly generated TÆMS task structure with a maximum depth of 4, branching factor of 4, and NLE count of 10 to seed the experiments. We were careful to use the same task structure, task arrival times, task deadlines and random numbers for each of the (robustness-mechanism, failure probability) pairs. Each experiment was repeated 20 times using a different randomly generated task structure. The experiments were run for 2500 clock cycles. For the *Survivalist* approach we used 4 as the value of the kill-count, K, i.e. the root node is cloned *thrice*. The minimum poll interval (for the *Are-You-Alive* messages) was set to 4 cycles and the maximum poll interval was set to 8 cycles. We used the following performance criteria to evaluate our two approaches to robustness:

1. The average number of agents used. (*Lower* is better.)
2. The percentage of tasks completed. (*Higher* is better.)
3. The resurrection interval. The resurrection interval is the amount of time an agent is "out-of-service" and is defined as the difference between the time an agent is killed and the time when it is restarted. (*Lower* is better.)
4. The average turnaround time. The turnaround time is defined as the difference between the time at which a task is either completed or failed and the time at which the task was generated (the generation time). The average turnaround time is the turnaround time divided by the total number of tasks. (*Lower* is better.)
5. The average quality accrued. The average quality is defined as the total quality accrued during the experimental run divided by the sum of the number of tasks completed and the number of tasks failed. (*Higher* is better.)
6. The total number of messages sent by all the agents. (*Lower* is better.)
7. The total resource cost of the organization. (*Lower* is better.)

4.5.2 Actual Results

The average results for these measured performance criteria are shown in Figure 4.13.

Again, we tested the statistical significance of the obtained results using the *Wilcoxon Matched-Pair Signed-Rank* tests with $p < 0.05$. *Matched-Pair* signifies that we are comparing the performance of each robustness approach on precisely



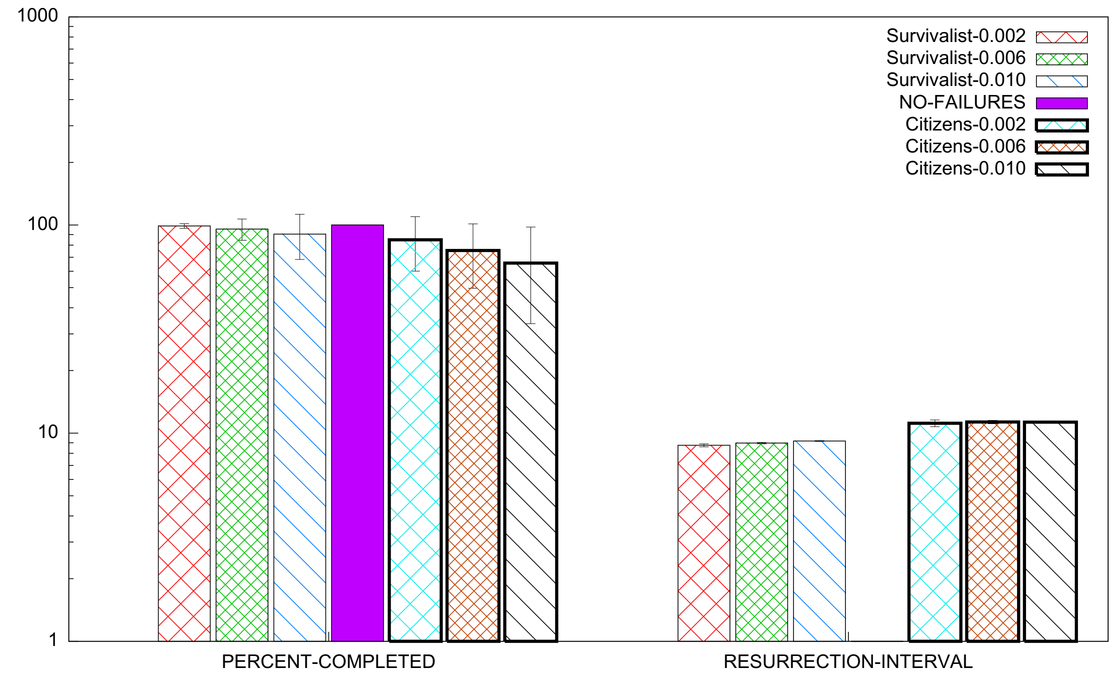


Figure 4.13: **Evaluation of the Robustness Mechanisms** — Graph showing the various measured parameters for the different robustness mechanisms.

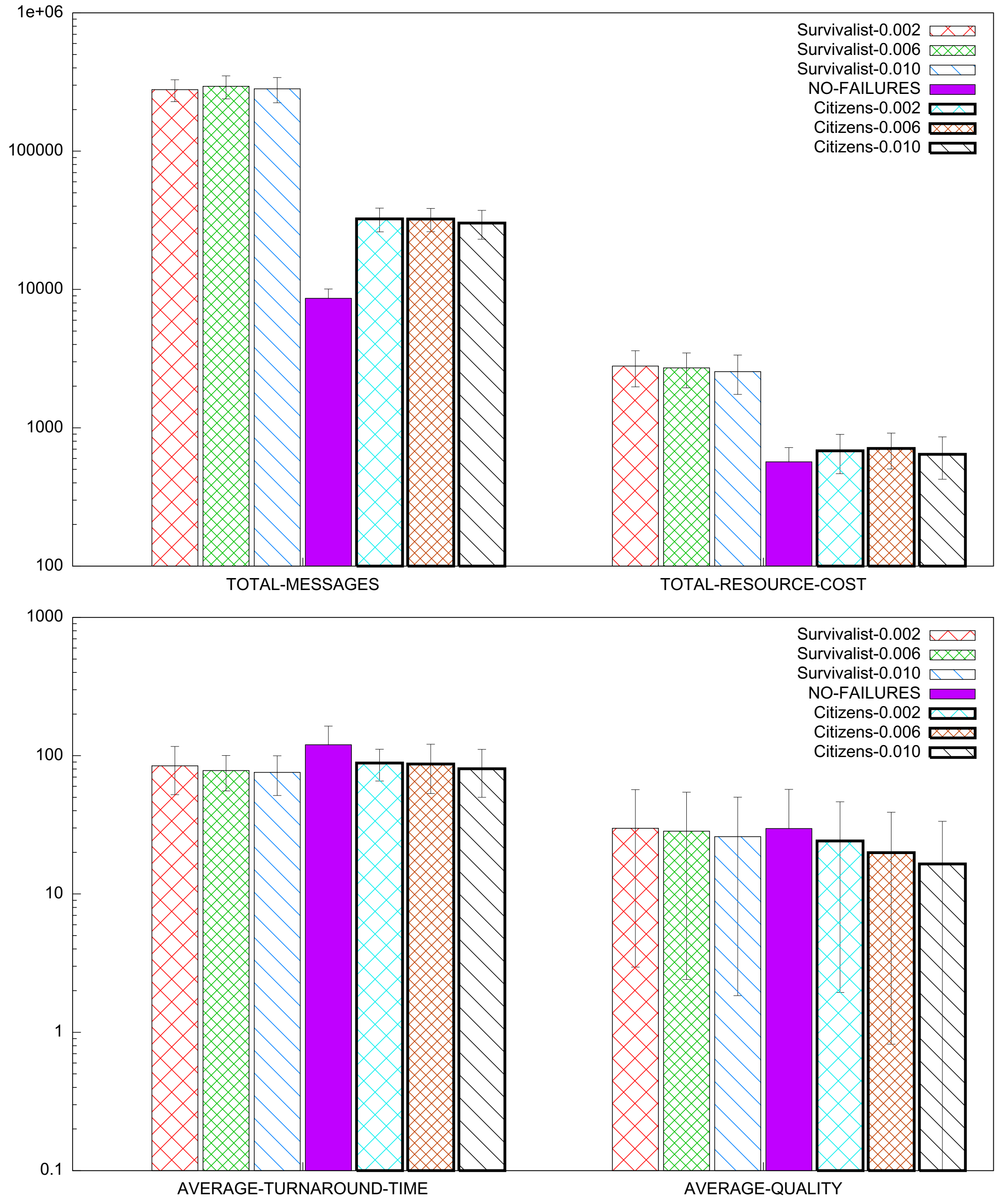


Figure 4.13: **Evaluation of the Robustness Mechanisms** — Graph showing the various measured parameters for the different robustness mechanisms.

the same randomized task set, environmental conditions and failure probabilities within each separate experiment. Some interesting observations are:

- As expected, in the absence of failures, the organization uses the fewest number of agents to complete 100 percent of the tasks. As the probability of failure increased, the number of agents increased (because agents were being killed and restarted — restarted agents were counted as new agents) and the percentage of tasks completed decreased. Furthermore, the difference in the percentage of tasks completed and number of agents spawned between the no failure case and the failure cases was statistically significant.

- We were, however, pleased with the overall performance of these two approaches to robustness. Even in the presence of a high number of agent failures, the *Survivalist* approach still managed to complete 90.44 % of the tasks on average (s.d. ±22.08). The *Citizens* approach did not perform as well and only completed 65.63 % of the tasks, on average (s.d. ±32.06). In fact, the difference in percentage of tasks completed between the two approaches was statistically significant for each of the probabilities of failure — the *Survivalist* approach consistently outperformed the *Citizens* approach at the same probability of failure. (In fact the *Survivalist* approach at probabilities of failure of 0.010 was statistically equivalent to the *Citizens* approach at probabilities of failure of 0.002.)

  We were really surprised to see the *Survivalist* approach out-performing the *Citizens* approach, especially since the *Survivalist* approach is completely decentralized and does not assume the presence of immortal monitor agents.

  The reason behind these results become clear once we look at the variation of the resurrection interval with the minimum poll interval as seen in Figure

4.14a[9]. For both these approaches, as the minimum poll interval increases, the resurrection interval also increases and the percentage of tasks completed decreases.

As can be seen in Figure 4.14a, the resurrection interval of the *Citizens* approach is higher than that of the *Survivalist* approach. This difference in resurrection interval is due to the fact that in the *Survivalist* approach, every agent is monitoring some set of other agents. Furthermore, all the agents randomize the time at which they send the *Are-You-Alive* messages[10]. These two factors result in the *Survivalist* approach detecting failures faster than the *Citizens* approach.

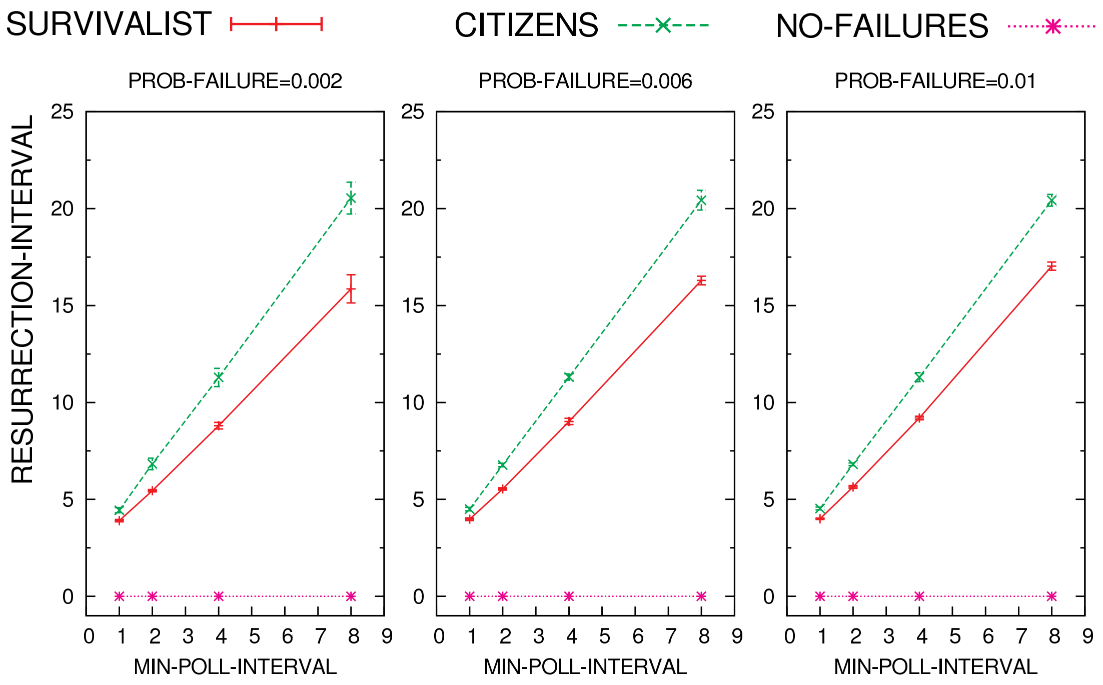
This, in turn, means that the agents stay "dead" for longer in the *Citizens* approach. Since, dead agents can not do any work, this causes the *Citizens* approach to miss more deadlines and complete fewer tasks than the *Survivalist* approach.

- The increased robustness performance of the *Survivalist* approach comes at the price of a statistically significant increase in the number of agents spawned, the number of messages sent and the total resource cost of the organization. This result was expected because the *Survivalist* approach pro-actively replicates agents to increase the robustness capacity of the organization.

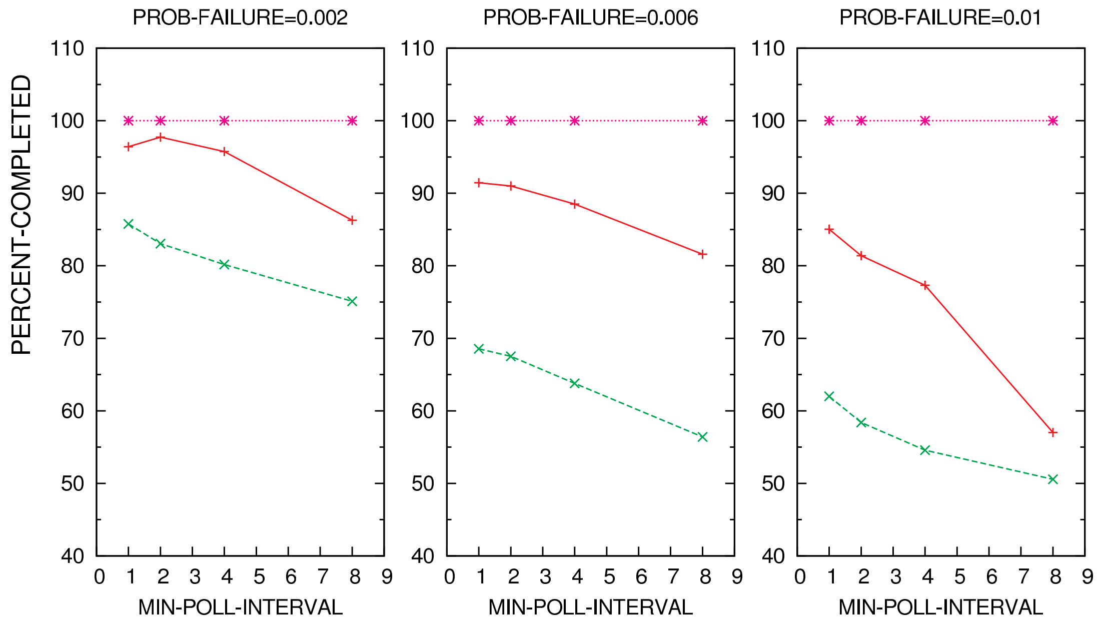
  Since, the number of extra agents replicated by the *Survivalist* approach depends on the value of the kill count, we decided to measure how the tested performance criteria varied with a change in the value of the kill-count or

[9] In these graphs, the maximum poll interval is set at twice the value of the minimum poll interval.

[10] If an agent in the *Survivalist* approach, has three other agents in its monitoring set, the interval at which it receives *Are-You-Alive* messages from the other agents is effectively one-third of the interval of the *Are-You-Alive* messages in the *Citizens* approach.

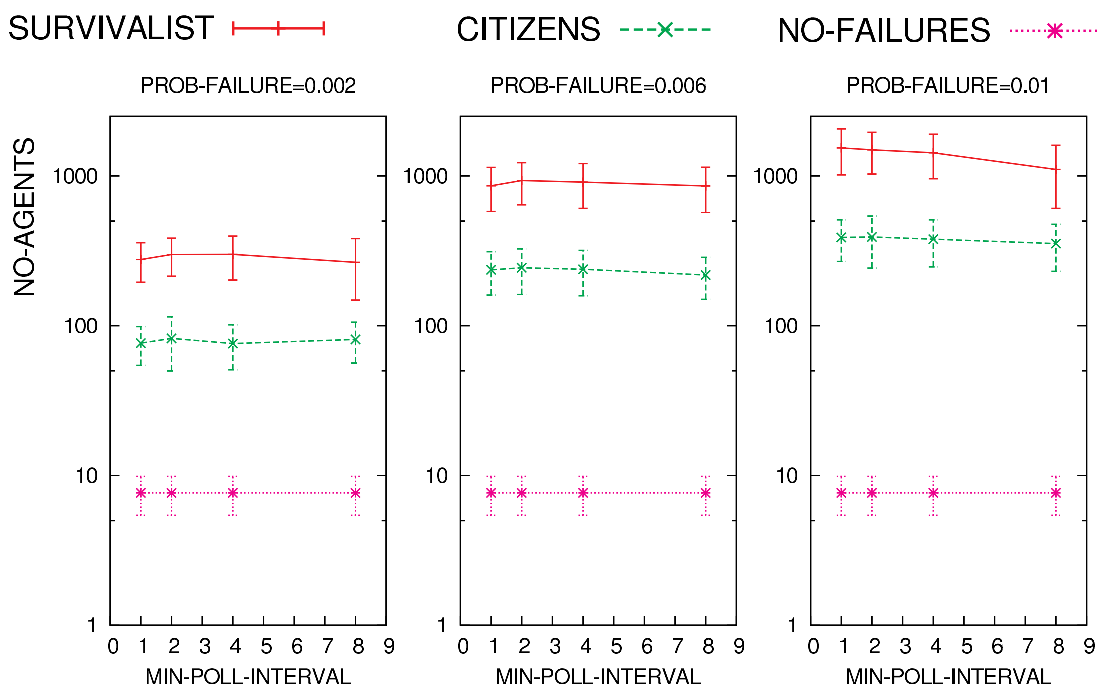


(a) Resurrection Interval

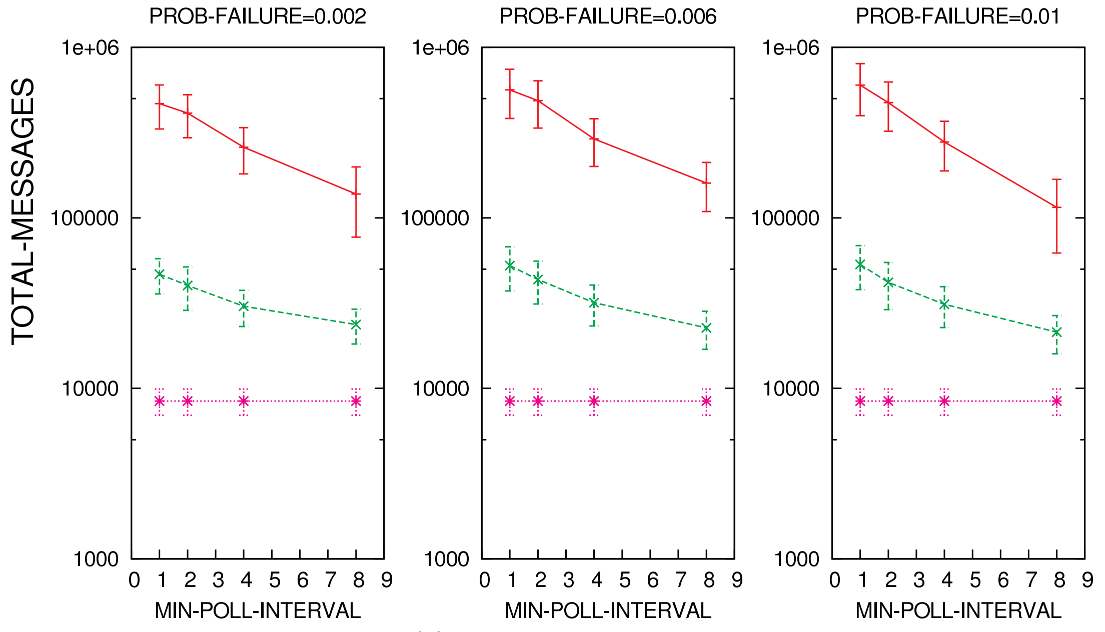


(b) Percentage of Tasks Completed (Error bars omitted)

Figure 4.14: Graph showing how the *Resurrection Interval* and *Percentage of Tasks Completed* varies with the minimum poll -interval and the probability of failure (averaged over all the experiments).



(c) Number of Agents



(d) Total Messages Sent

Figure 4.14: Graph showing how the *Number of Agents* and *Total Number of Messages Sent* varies with the varies with the minimum poll-interval and the probability of failure (averaged over all the experiments).

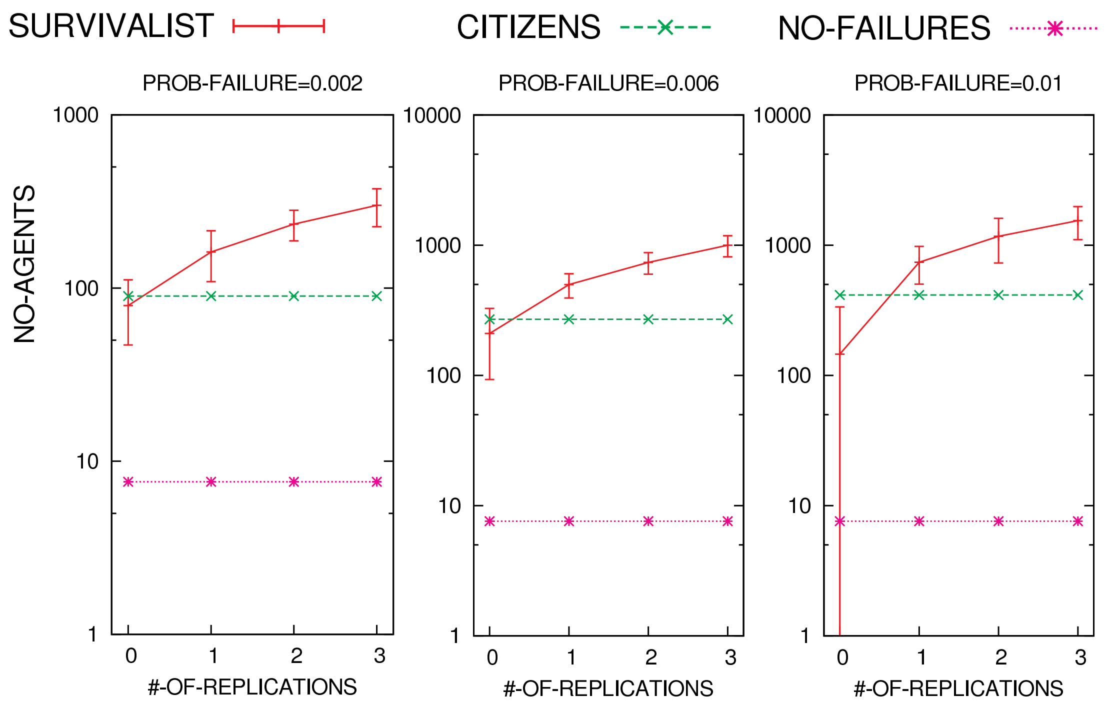
K. The results are show in Figure 4.15. Note that these graphs shown the number-of-replications[11] instead of the kill count.

As can be seen, both the number of agents and percentage of tasks completed increases with an increase in the kill count. The no replication case (number-of-replications = 0) uses fewer agents than the *Citizens* approach and also completes a fewer percentage of tasks than the *Citizens* approach. Furthermore, this result is statistically significant for probabilities of failure greater than 0.002 — showing that the centralized *Citizens* approach out-performs the distributed *Survivalist* approach in the absence of replication. However, with replications, the *Survivalist* approach uses more agents but outperforms the *Citizens* approach in the percentage of tasks completed.

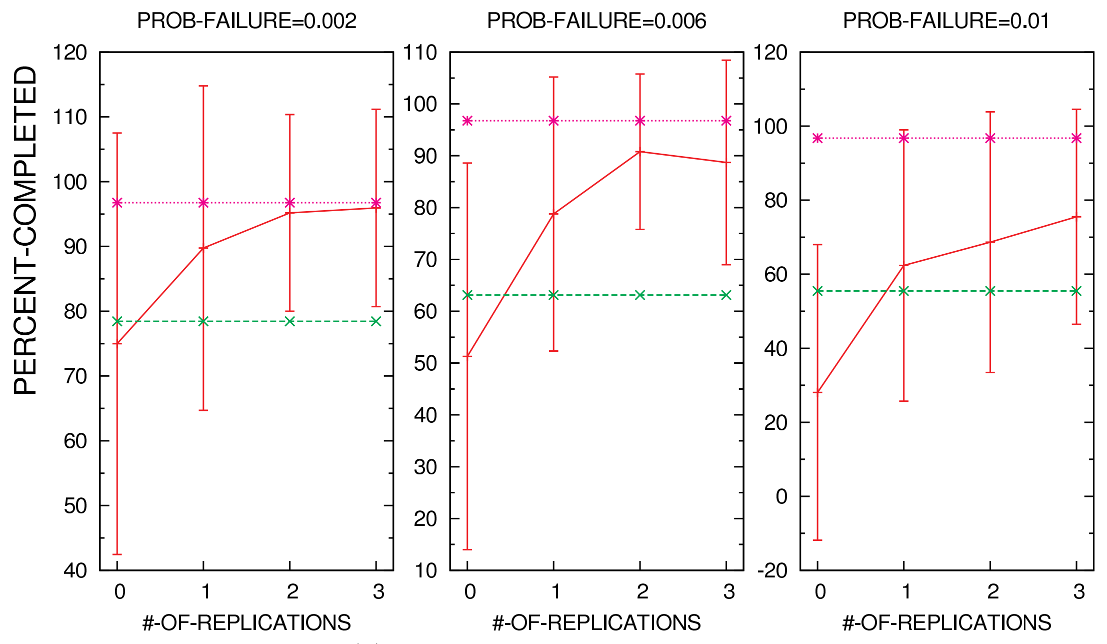
- Finally, the total resource cost of the organization is lower for the *Citizens* approach when compared to the *Survivalist* approach. This is as expected because the resource cost is directly dependent on the number of agents used — since the *Citizens* approach uses fewer agents, it has a lower total organizational resource cost.

Note that despite prima facie evidence to the contrary, our results do not directly contradict the results presented in Dellarocas & Klein [2000]. This is because in their work, Dellarocas & Klein [2000] do not consider replication in their discussion of the *Survivalist* approach (that is, they only consider the no replication case, where number-of-replications = 0). In our results, for the no replication case, the *Citizens* approach outperforms the *Survivalist* approach, which is in-tune with their results.

[11] The number of replications is the number of times the root node was replicated. Recall that the number of replications is defined as $K - 1$.

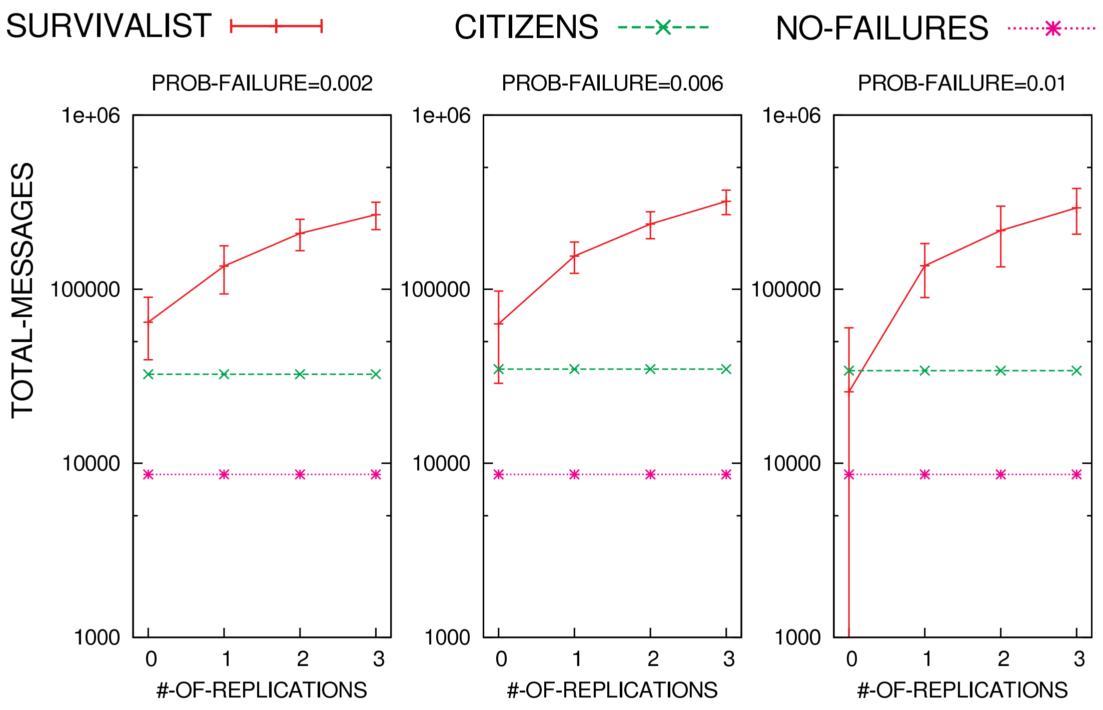


(a) Number of Agents

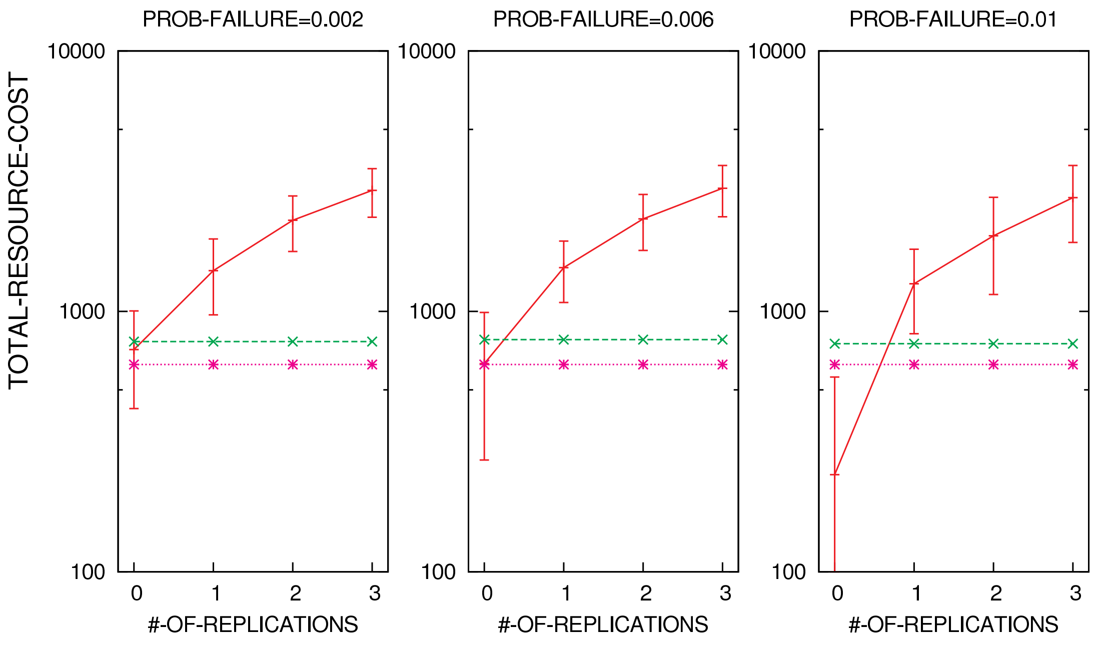


(b) Percentage of Tasks Completed

Figure 4.15: Graph showing how the *Number of Agents* and *Percentage of Tasks Completed* varies with the desired robustness level and the probability of failure (averaged over all the experiments). Note that standard error bars are only shown for the *Survivalist* approach.



(c) Total Messages Sent



(d) Total Resource Cost

Figure 4.15: Graph showing how the *Total Number of Messages Sent* and *Total Resource Cost* varies with the desired robustness level and the probability of failure (averaged over all the experiments). Note that standard error bars are only shown for the *Survivalist* approach.

4.6 Chapter Summary

In this chapter we described a series of experiments that test the use of OSD to allow the agents to generate their own organizational structures at run-time.

Section 4.1 compared the OSD approach to the Contract Net Protocol and showed that our OSD approach performed significantly better than the contract net protocol — our OSD approach completed a larger number of tasks, accrued more quality and also had a lower response time and communication overhead when compared to the contract net protocol.

Section 4.2 defined the *sp-diff* value — a value that is used to control the arrival rates and deadlines of the tasks in later experiments. This value allows easy comparison of randomly generated tasks with vastly different task characteristics and allows us to see the effect of the environmental conditions on the task structures.

Section 4.3 describes an evaluation of the three different task allocation heuristics outlined in Section 3.4.5.1 against a random task allocation strategy. These results suggest that the *Balancing Execution Time* (BET) heuristic outperforms the other heuristics for a majority of the performance criteria.

Section 4.4 describes an evaluation of of the five spawning strategies presented in Section 3.4.5.2. In this section, we show how the choice of the spawning strategy depends on the environmental conditions under which the tasks are being solved. We also show that our *Hybrid Model* presents a good tradeoff between the *Breakup* and the *Cloning* strategies.

Finally, Section 4.5 discusses the performance of the two commonly employed approaches to robustness — the *Survivalist* approach and the *Citizens* approach when applied to OSD. We show that the *Survivalist* approach achieves a higher level of robustness than the *Citizens* approach. However, the *Survivalist* approach uses more agents and has a much higher communication overhead than the *Citizens* approach. Hence, there is a tradeoff between using these two approaches.

Chapter 5

APPLYING ORGANIZATIONAL SELF-DESIGN TO REAL WORLD DISTRIBUTED COMPUTING SYSTEMS

Basic research is very useful, but it should be more geared toward application than it was before. (Luc Montagnier)

The pursuit of pretty formulas and neat theorems can no doubt quickly degenerate into a silly vice, but so can the quest for austere generalities which are so very general indeed that they are incapable of application to any particular. (E. T. Bell)

One of the primary hypothesis behind this dissertation is that multiagent organizations, in general, and OSD, in particular, are especially suited to the problem of generating virtual organizations for grid-, volunteer-, and cloud- computing systems. This is because (a) such systems are often used to solve complex problems in worth oriented domains and would benefit from having a more flexible workflow representation language that allows quality, cost and duration tradeoffs to be made [Atlas *et al.* , 2005]; and (b) such systems are typically deployed in dynamic and semi-dynamic environments [Taufer *et al.* , 2005a].

In this chapter, we discuss the ways in which OSD can be applied to real-world computing systems. We primarily focus on applying OSD to a to a real-world volunteer computing system, specifically the docking@home project.

5.1 Volunteer Computing

Volunteer computing [Anderson, 2004; Shirts & Pande, 2000] is a form of distributed computing in which a group of volunteers donate their computing resources to a cause, such as folding proteins, predicting climate change, etc. Currently volunteer computing has been implemented using a master-slave (client-server) architecture. Volunteers download a client that connects to one or more centralized servers and requests jobs that make use of the volunteer's computing resources. The centralized servers, in turn, need to figure out a scheduling policy that tries to perform an optimal allocation of jobs to the clients (for some definition of optimality).

The clients running on the volunteer machines can be thought of as agents. This leads to a direct mapping from the problem of determining a suitable scheduling policy for the clients to the problem of determining a suitable organization for the agents. Hence, the solution to the organizational issues, such as the allocation of agents to the subtasks of the problem being solved and the coordination of inter-agent activities, will generate a scheduling policy that can be used to allocate jobs to the agents.

We focus on applying OSD to the problem of studying protein-ligand docking [Taufer *et al.* , 2005b]. Ligands are small molecules that bind to proteins and can be used to regulate their function[Vera *et al.* , 2007]. Inhibiting the activity of key enzymes (proteins) may result in entire biochemical pathways being turned on or off. Indeed, many small molecule drugs marketed today function by inhibiting enzymes. Hence, protein-ligand docking can be the first step towards discovering new drugs.

In this chapter, we describe how the problem of job-allocation in volunteer computing systems can be mapped onto the problem of solving a set of problem instances represented in TÆMS through the use of organizational self-design. Hence,

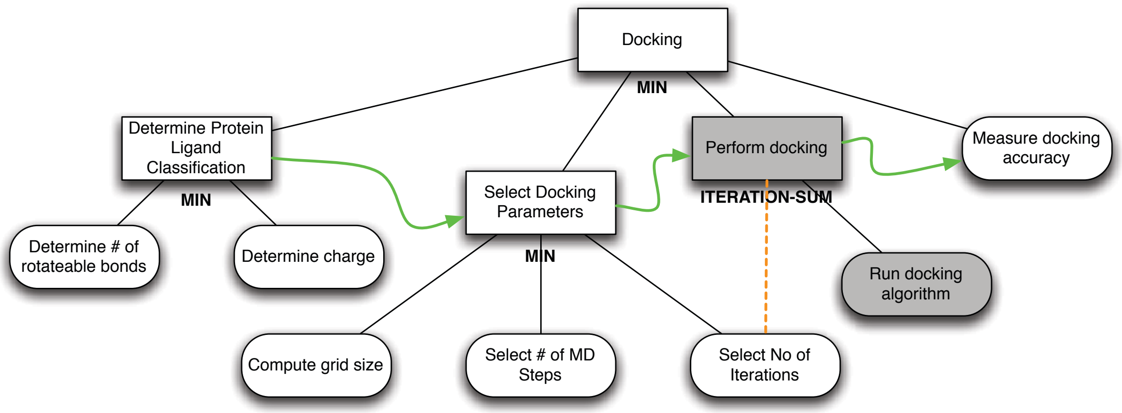


Figure 5.1: A simplified TÆMS task structure representing protein-ligand docking. The white rectangles represent tasks, the shaded rectangles represent templates the rounded rectangles represent executable methods. The arrows represent non-local effects. *Method outcomes and characteristics are not shown.*

we try to bridge the gap between theoretical OSD research and a practical application of such research to volunteer computing systems. Furthermore, we advocate moving volunteer computing from a strictly master-slave paradigm to a more distributed peer-to-peer model. We show that such a move allows for increased throughput of such systems, while at the same time minimizing the load on the centralized servers.

5.2 Current Approaches to Docking

The use of molecular dynamics for protein-ligand docking[1] was first explored in [Taufer *et al.* , 2005b]. In this work, the researchers used a traditional cluster to run their docking attempts. Subsequent work lead to the creation of the docking@home project, a volunteer computing system based on the BOINC framework [Anderson, 2004].

BOINC (Berkeley Open Infrastructure for Network Computing) is an open-source framework that allows researchers to harness the computing resources of a

[1] Henceforth, referred to as simply docking

large and heterogeneous group of volunteers. Task and resource allocation amongst the volunteers is performed by the BOINC middle-ware, which allocates jobs (tasks) to the volunteers based on a *scheduling policy*.

Current approaches to task allocation in BOINC [Anderson & Reed, 2009] are based on a naive and greedy algorithm. In the current system, volunteers (agents) periodically issue a *scheduler request* to a centralized server for a certain number of jobs. The centralized server then assigns a group of jobs to the requesting volunteer. To select the assigned jobs, the server randomly scans its cache of outstanding jobs for a set of jobs that can feasibly be run on the volunteer's machine. This set of jobs is assigned to the user.

5.3 The OSD Approach

The OSD approach to docking is based on two intuitions:

1. The agents (volunteers) have the most information about the usage patterns of the volunteers' computers. That is the volunteer agents can easily monitor the computers on which they are running and can, hence, learn if a computer is lightly used (for example, if the owner of the computer only uses it to check her/his email) or if it is heavily used (for example, if the owner is an avid gamer who loves to play CPU intensive games).

   Hence, the volunteer agents can easily determine if the computer is overloaded and will not be able to complete the jobs assigned to it by their deadlines. In such situations, it can pro-actively spawn off new agents before the time-out/deadline of the jobs.

   Our hypothesis is that our OSD approach should work much better than the current approach to docking in which the BOINC server has no clue if the jobs assigned to a volunteer will be completed by the deadlines assigned to that volunteer. The BOINC server has to wait for jobs to timeout before assigning them to other volunteers.

2. By implementing a semi-hierarchical organization, the load on the BOINC server can be reduced. Specifically, when using OSD, the BOINC server does not need to maintain detailed information about the millions of volunteer agents. Instead it can simply maintain information about its immediate children sub-organizations and allocate jobs based on this information.

   Our hypothesis is that the OSD approach should make better scheduling decisions because it can make these decisions based on more detailed observations of the sub-organizations. Furthermore, the BOINC server should be able to make better predictions about the agents that form these sub-organizations.

In the OSD approach to docking, the task structure presented in Fig. 5.1 is provided as an input to the system. As with regular OSD, the organization starts of with a single agent responsible for all the activities of the organization. This single agent is equivalent to the BOINC server in regular docking and performs the same functions as the BOINC server — it accepts protein-ligand pairs for docking, from the user, and "generates" a task instance for the input pair. It then applies the standard OSD approach to this task instance, with the exception that (a) spawning a new agent involves removing a volunteer from the volunteer pool and creating an agent (a wrapper or plug-in for the standard BOINC docking client) for that volunteer; and (b) composition involves destroying the agent and re-adding the volunteer to the volunteer pool[2].

Two key departures from regular BOINC docking are in the way in which we handle the clients:

[2] For the purposes of this chapter, we assume that the volunteer pool is infinite. In our future work, we would like to investigate OSD approaches that work with a finite albeit changing set of volunteers.

1. Instead of a regular master-slave paradigm (i.e. a hierarchical organization that has a depth of 1), the volunteer clients in our approach form a peer-to-peer sub-organization with a much deeper hierarchical structure[3].

   A volunteer that detects an overload while trying to complete the jobs assigned to it will spawn off a new agent[4]. The spawning volunteer will, in turn, be responsible for assigning a subset of its jobs to the newly spawned volunteer. The spawned volunteer may itself spawn-off new agents, hence forming a multi-level hierarchy.

2. As in regular docking, the BOINC server is responsible for assigning jobs to the volunteer agents. However, (a) the volunteers may reassign jobs to other volunteers at lower levels of the hierarchy; and (b) instead of the naive BOINC scheduling algorithm, the assignment of jobs to the lower level sub-organizations is based on the past performance of the sub-organization. Specifically we maintain two pieces of information about each sub-organization, i:

   (a) $\text{T-S{\scriptsize INGLE}}_i(n)$ or the n^{th} *estimate* of the amount of time needed by sub-organization i to complete a *single* assigned job. For every result, j, returned, a new estimate is computed using according to the formula:

   $$\text{T-S{\scriptsize INGLE}}_i(n+1) = \alpha * \text{D{\scriptsize URATION}}(j) + (1-\alpha) * \text{T-S{\scriptsize INGLE}}_i(n) \tag{5.1}$$

   where $\text{T-S{\scriptsize INGLE}}_i(n+1)$ is the next (that is, $(n+1)^{th}$) estimate of the amount of time needed to complete a single job, $\text{D{\scriptsize URATION}}(j)$ is the

[3] The volunteers only form hierarchical structure if composition is disabled. If composition is enabled, the hierarchy would break down to form an interconnected mesh structure because we allow any agent to compose with any other agent

[4] Spawning an agent still involves asking the BOINC server for a “new” agent created from its pool of volunteers. The BOINC server would provide a service similar to the centralized trackers in the bit-torrent protocol.

amount of time it took for job j to complete, and α is a constant between 0 and 1.

(b) $\textsc{Outstanding}_i$ or the estimated completion time for *all* the outstanding jobs assigned to the sub-organization, i.

Based on these values, the jobs are assigned to the sub-organization based on the equation:

$$\underset{i \in \textsc{Sub-Organizations}(\kappa)}{\arg\min} \{\text{T-}\textsc{Outstanding}_i + \text{T-}\textsc{Single}_i\} \tag{5.2}$$

Hence, a new job is assigned to the sub-organization that is expected to complete the new job in the fastest time possible.

A slightly complicating issue, in the assignment of jobs to the volunteers, is the fact that each job has to be replicated a certain number of times (say r) to ensure the validity of that job[5]. This is handled in a way that is similar to the survivalist approach to robustness (See Section 3.4.7.2) — That is, the *Run Docking Algorithm* method in Figure 5.1 is explicitly cloned r times at startup and DIFFERENT norms (See Section 3.4.3) are inserted between the r clones of this method. This ensures that each job is replicated the correct number of times and that no volunteer works on two replicas of the same job. Finally, a verification method is added to verify the results obtained from the volunteers.

5.4 Evaluation

As a first step towards evaluating our approach, we ran a series of experiments comparing our OSD approach to the BOINC approach with 25, 50 and 100 volunteers[6].

[5] The results of a job are only accepted if a certain number of these replicas agree on the results.

[6] Our OSD approach automatically selects an appropriate number of agents through spawning and composition

To simulate our approach we needed a volunteer population. To keep the runs realistic, we used the statistical models reported in [Estrada *et al.* , 2009] to compute the *work-in-progress* time or the time needed by a volunteer to generate a result for a job submitted to it (henceforth referred to as the *runtime*). These models were generated from actual BOINC traces. In this chapter, we report the results obtained by assuming two different volunteer populations: Charmm and MFold from the Prediction@Home BOINC project.

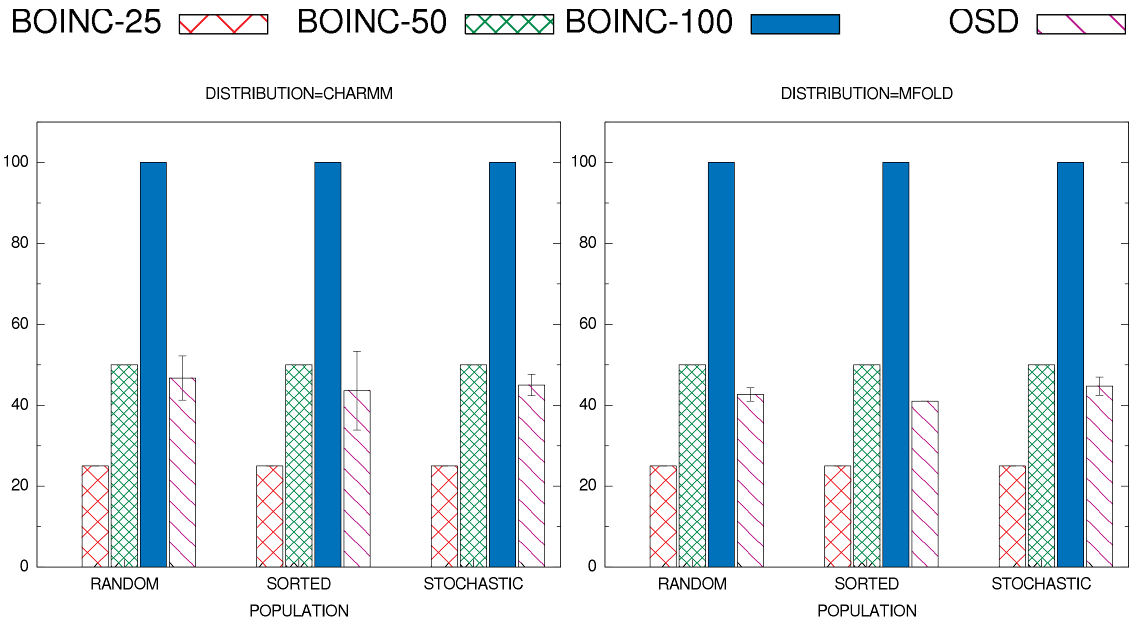
To generate a volunteer population consisting of n volunteers, we used the statistical models to generate an array of $n \cdot m$ random run-times, where m is the maximum number of jobs generated during a simulation run. We then partitioned this array into n non-overlapping lists — each list was used to generate the runtime behavior of a specific volunteer. This partitioning was done in three different ways:

1. **Random**, the generated array of run-times was partitioned into n equal parts. This population would represent volunteers with no fixed pattern.

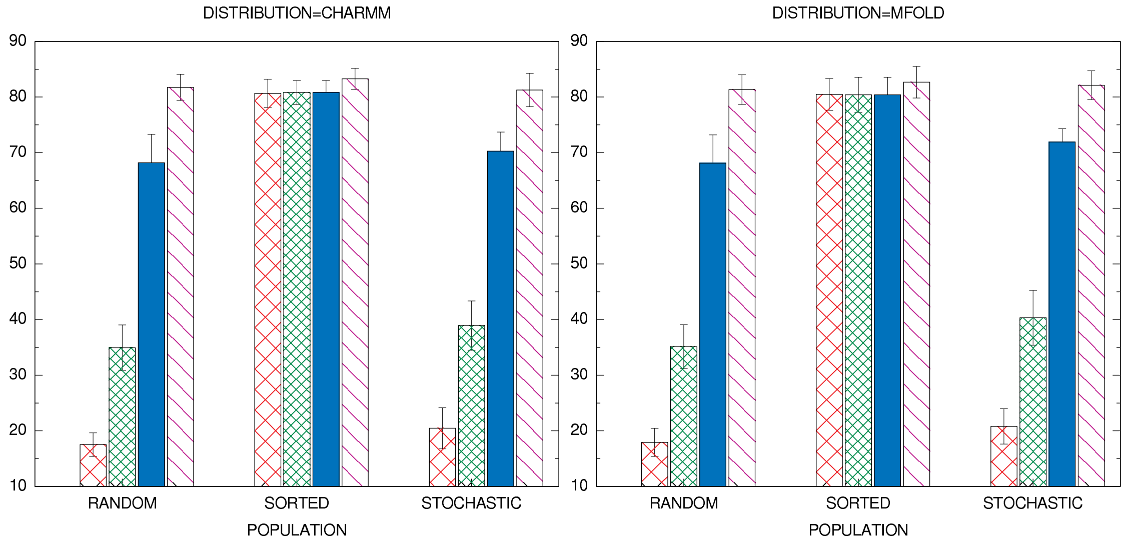
2. **Sorted**, the generated array of run-times was first sorted and then partitioned into n equal parts. The sorted volunteer population would simulate volunteers with a fixed and predictable runtime behavior.

3. **Stochastic**, the generated array of run-times was first sorted. However, instead of performing a simple partition, the sorted array was sampled stochastically to generate the volunteer population. That is, for each volunteer, in turn, the next runtime was selected with probability p and randomly with probability $(1 - p)$.

Each experiment was repeated 15 times with a new runtime array and a new volunteer population for each run. The results are show in Fig. 5.2. As can be seen, the initial results are extremely promising. Not only does our approach complete a larger number of tasks than the BOINC approaches, the result is also

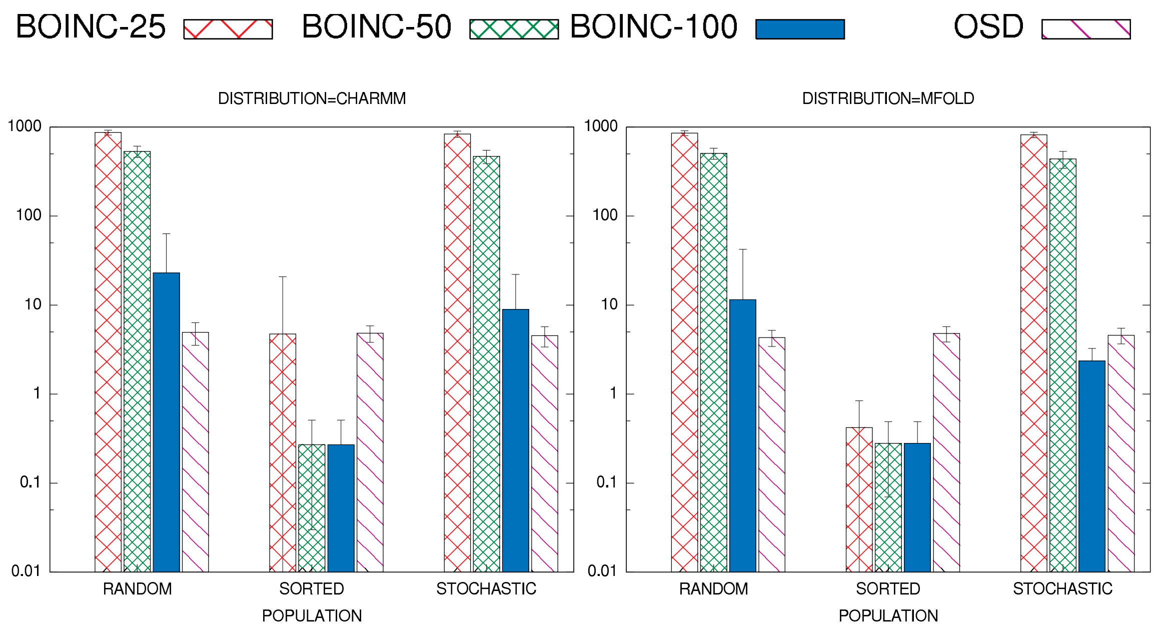


(a) Number of Agents

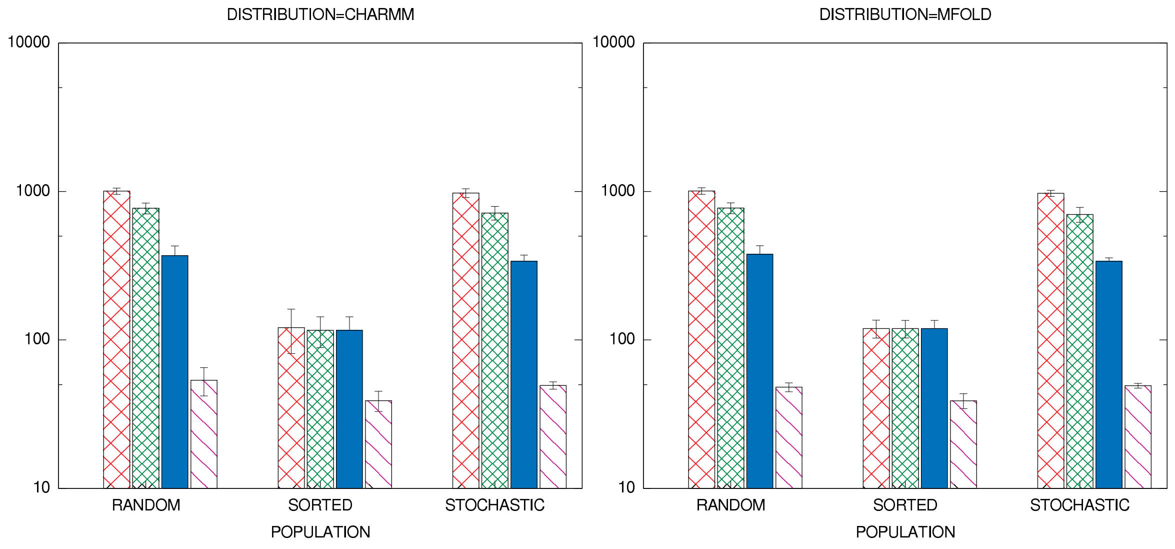


(b) Number of Tasks Completed

Figure 5.2: Graphs showing the average performance of our OSD approach against the BOINC approach with 25, 50 and 100 volunteers.



(c) Average Response Time



(d) Average Turnaround Time

Figure 5.2: Graphs showing the average performance of our OSD approach against the BOINC approach with 25, 50 and 100 volunteers.

statistically significant. Furthermore, our approach has a lower response-time and turnaround-time while using fewer than 50 agents on average.

5.5 Chapter Summary

This chapter presented a first attempt at applying our research on OSD to a real-world volunteer computing system. Section 5.1 described how the problem of determining an appropriate scheduling policy can be mapped onto the problem of generating an organization for the agents. This was followed by a description of the current approaches to docking in Section 5.2. Section 5.3 presented a novel approach to docking based on our OSD research. Finally, our OSD approach to docking was tested against the current naive approach to docking in Section 5.4.

However, our OSD approach would require a significant change to the current BOINC approach of allocating jobs to volunteers. Specifically, as explained in Section 5.3, we need to (1) move from the current master-slave paradigm to a more peer-to-peer based organizational structure; and (2) we need to the maintain a history of the past performance of the volunteers. We expect the former to be a more controversial and radical change than the latter — Already, a few researchers (for example, see [Sonnek *et al.* , 2006] and [Budati *et al.* , 2007]) have proposed using a reputation-based system for allocating jobs to the volunteers, where the reputation of a volunteer is a measure of the reliability of that volunteer in delivering correct results to the server. Whereas, in our system we use the turnaround time to allocate jobs to the volunteers, our system can be trivially extended to allocate jobs based on the reliability in addition to the turnaround time.

So the question is: How reasonable is a move from the current master-slave paradigm to our proposed OSD based peer-to-peer system and are there justifiable advantages in making the move? We believe that there aren't many technical limitations to such a move — for example, see [Costa *et al.* , 2008] for a discussion of how peer-to-peer data distribution techniques can be adapted to the BOINC platform[7]. We also believe there are significant advantages of making such a move that will outweigh the technical difficulties. Specifically, our system can significantly reduce the load on the central BOINC servers, a problem that is likely to be exacerbated once a reputation-based system in implemented on top of BOINC, while still increasing the throughput of the whole system. Furthermore, our approach increases the robustness of the whole system since there is no central point of failure. However, we believe that the most significant advantage of moving to our OSD-based system would be gleaned from the TÆMS based representation of volunteer-computing workflows. Using TÆMS would allow volunteer-computing customers (i.e. the researchers that use volunteer computing systems like BOINC and folding@home) to use a more flexible workflow representation language — one that would allow them to represent alternative ways of achieving a goal (algorithms for solving a problem) with different quality/time/resource tradeoffs. Our approach would then provide these researchers with custom solutions based on the characteristics of the problem and their desired optimizing criteria. So, for example, there are alternative docking algorithms based on implicit and explicit representations of the protein/ligand and there are different scoring techniques based on Cartesian Molecular Dynamics (MD) and Torsion Angle Molecular Dynamics (TAMD). By representing the larger workflow in TÆMS (see Figure 5.3), our OSD approach can select different algorithms and try different alternatives based on the quality achieved and the time available for coming up with a solution.

[7] Note that this paper does not propose any specific algorithms and does not discuss an existing implementation.

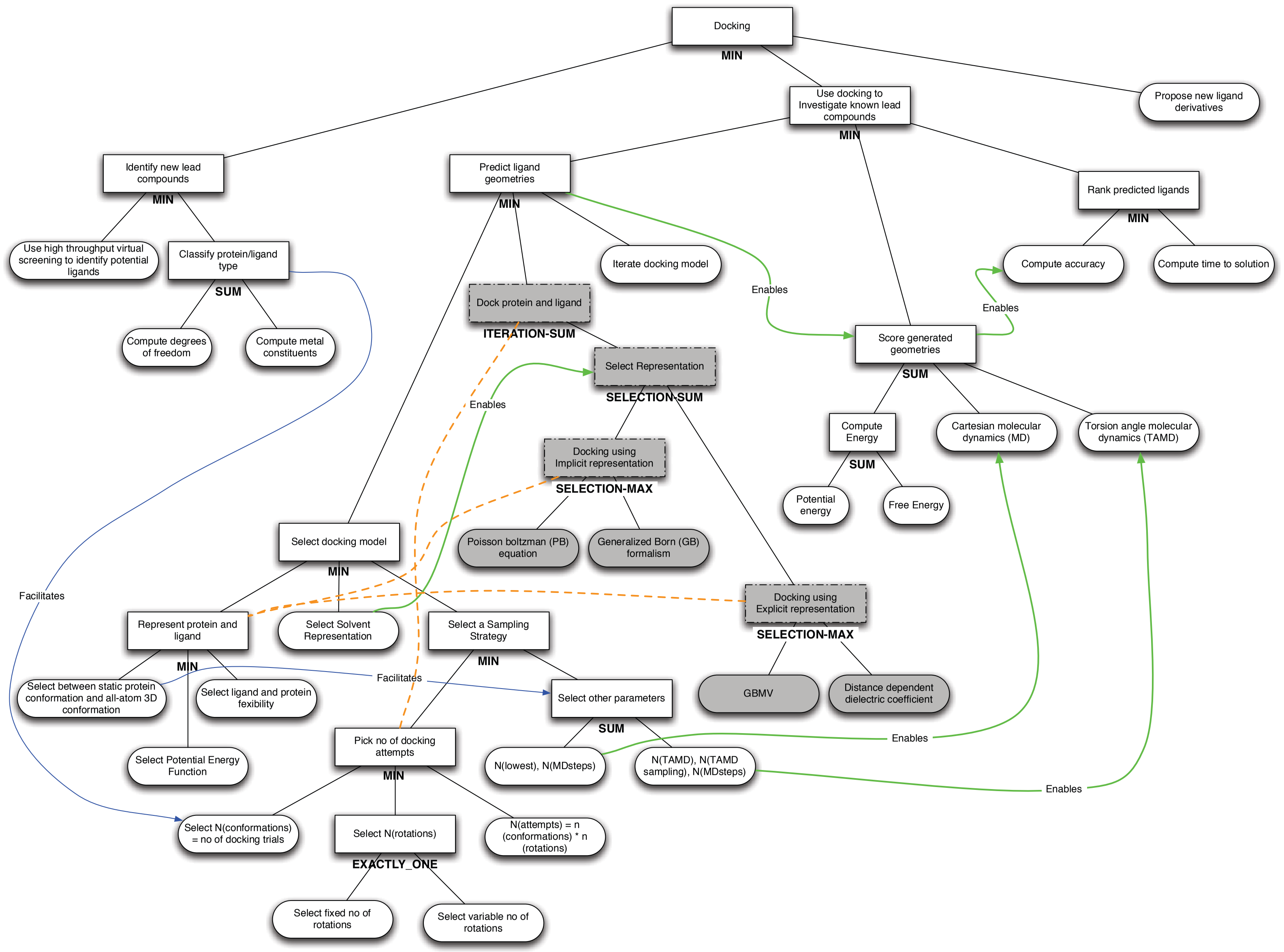

Docking
MIN
Identify new lead compounds
MIN
Use high throughput virtual screening to identify potential ligands
Classify protein/ligand type
SUM
Compute degrees of freedom
Compute metal constituents
Use docking to Investigate known lead compounds
MIN
Propose new ligand derivatives
Predict ligand geometries
MIN
Iterate docking model
Dock protein and ligand
ITERATION-SUM
Select Representation
SELECTION-SUM
Docking using Implicit representation
SELECTION-MAX
Poisson boltzman (PB) equation
Generalized Born (GB) formalism
Docking using Explicit representation
SELECTION-MAX
GBMV
Distance dependent dielectric coefficient
Rank predicted ligands
MIN
Compute accuracy
Compute time to solution
Score generated geometries
SUM
Compute Energy
SUM
Potential energy
Free Energy
Cartesian molecular dynamics (MD)
Torsion angle molecular dynamics (TAMD)
Enables
Enables
Enables
Enables
Enables
Select docking model
MIN
Represent protein and ligand
MIN
Select between static protein conformation and all-atom 3D conformation
Select ligand and protein fexibility
Select Potential Energy Function
Select Solvent Representation
Select a Sampling Strategy
MIN
Select other parameters
SUM
N(lowest), N(MDsteps)
N(TAMD), N(TAMD sampling), N(MDsteps)
Pick no of docking attempts
MIN
Select N(conformations) = no of docking trials
Select N(rotations)
EXACTLY_ONE
Select fixed no of rotations
Select variable no of rotations
N(attempts) = n (conformations) * n (rotations)
Facilitates
Facilitates

Figure 5.3: An extended TÆMS task structure representing protein-ligand docking

Chapter 6

CONCLUSION

A conclusion is the place where you got tired thinking.
(Martin Henry Fischer)

I have but one lamp by which my feet are guided, and that is the lamp of experience. I know no way of judging of the future but by the past.
(Edward Gibbon)

6.1 Conclusion

In this dissertation, we focus on the organizational design of a subset of multi-agent systems in which the environment is semi-dynamic. Our primary hypothesis is that most current approaches to organizational design either model the organization at design time, assuming a static environment, or generate a new organization on the fly, at run time, for each new instance of the problem and that such approaches are inefficient and fail to correctly model the dynamics of a slowly changing environment. Hence, Organizational Self-Design, in which the agents are responsible for designing their own organization at run-time is particularly suited to organizational design in such environments.

In this dissertation, we have described how Organizational Self-Design can be applied to worth-oriented domains in which problems are represented using the TÆMS task representation language.

In Chapter 3, we developed a formal model for OSD and extended the TÆMS language to (a) represent information about the local task view of the agent vis-a-vis its role in the organization and its relationships to the other agents; and (b) represent iteration nodes in which the number of iterations of a task depend on the results of executing another task. We also showed how task structure rewriting can be used to change the organizational structure of the agents at run-time. In Section 3.4.5.1, we have provided a description of three task allocation heuristics that can be used to select a role for allocation to the newly spawned agent. We follow this up with Section 3.4.5.2, which discusses different spawning strategies for choosing between the generalization and specialization of an agent. Section 3.4.6 describes how the agents might detect a need for reorganization. Finally, we present various robustness mechanisms in Section 3.4.7.

In Chapter 4, we ran a series of experiments to test the performance and robustness of our approach and the different algorithms and heuristics that we developed. Section 4.1, compared our OSD approach to the Contract Net Protocol. Then in Section 4.3, we compared the three task allocation heuristics to a random task allocation approach. Section 4.4 evaluates the five spawning strategies presented in the previous chapter. Finally, this chapter ends with an evaluation of the two robustness mechanisms in Section 4.5.

Chapter 5 presented a first attempt at applying OSD to a real world volunteer-computing application, specifically protein-ligand docking. We show that applying OSD to the job-allocation problem in volunteer computing gave some promising results — our approach completed a larger number of tasks and had a better turnaround time compared to the existing BOINC scheduling approach.

In summary, we have developed a set of algorithms, heuristics and models that mark a significant step forward in the application of Organizational Self-Design techniques to problems in dynamic, uncertain, worth-oriented domains. We believe these techniques will be useful to not just the multiagent systems community but also to the fields of high-performance grid, cloud and volunteer computing. In particular we believe that the grid, cloud and volunteer computing communities have focused on the low-level middle-ware issues such as addressing, discovering and monitoring individual resources and have not focused on the high level issues, such as generating an explicit organization that is able to automatically manage and schedule these resources in an efficient and predictable manner. This is where our research comes in — we provide a high level framework that can be used to effectively allocate and manage the resources in such distributed computing environments. Hence, we hope that our research can help bridge the gap between the focus on individual resources in distributed computing community and focus on the collective coordination of the agents in the multiagent community.

6.2 Lessons Learned

We started this thesis based on the postulations of Contingency Theory — that there is no best way of organizing and that all ways of organizing are not equally effective. Since the optimal organizational structure depends on the problem being solved and the environmental conditions under which they are being solved, we expected a run-time approach to be more suitable for dynamic and uncertain environments as it can adapt to the changing conditions at hand. As the primary lesson learned, we would like to add a corollary to Contingency Theory "*There is no optimal set of heuristics or algorithms that are sufficient for adapting an organization to the changing environmental conditions and problem structure.*" We found that none of our heuristics or algorithms out-performed all the others for all the *randomly* generated task structures. Indeed, for every heuristic it was possible to generate a counter-example in which that heuristic wouldn't work as well.

This is not a negative result, per se — due to the inherent difficulty and computational intractability of the problem of organizing, we would have been surprised if the opposite was true. However, this work led us to strongly believe that using heuristics developed on completely random task structures is the wrong thing to do for real-world applications. Instead, use the algorithms and heuristics presented here as a baseline and systematically vary the constants and the algorithms to whatever works best for the application at hand. This might require a few bootstrapping steps before OSD can be deployed "in the field" but should provide consistent results for a group of similar problems.

6.3 Future Work

We will divide our future work into two parts — (1) work on the further development of OSD; and (2) applying OSD to future applications in volunteer/grid/cloud computing systems.

6.3.1 Further Development of OSD

There are several ways in which our OSD work can be further extended:

1. The performance of the algorithms and heuristics presented in this work depend on the values of a lot of "constants"[1]. (For example, six constants are used to select the choice of an agent's spawning strategy.) However, we haven't yet discovered the best values for these constants or whether these constants even have an optimal value. In our future work, we would like to run some experiments that determine the sensitivity of our algorithms to the value of these constants. Furthermore, we would like to determine how these constants interact with each other.

[1] There are at least two dozen such constants in our OSD code. Not all of them have been mentioned in this thesis.

2. One of the open questions in this dissertation is the consideration of the interplay between coordination and organization. Do certain coordination mechanisms preclude the use of certain organizational structures, or vice versa? What coordination protocols are suitable for an arbitrary type of organizational structure? In our future work, we would like to evaluate a family of coordination mechanisms to see which ones perform best under a given set of conditions. Towards this end, we will be looking at both commitments made during spawning and mechanisms available during execution of a task instance.

3. In this work, we have only looked at the allocation of exclusive, non-consumable resources. We would like to extend this work to look at both shared and consumable resources. We would like to answer questions like: How does the resource model constrain the kinds of organizational structures that can be generated? How do the algorithms change in an environment with heterogeneous resources?

6.3.2 Applying OSD to Applications in Volunteer/Grid/Cloud Computing Systems

Chapter 5 only scratched the surface of what's possible when OSD is applied to real world volunteer/grid/cloud computing applications. With reference to volunteer computing, in our future work, we would like to:

1. Scale and test our OSD approach with several hundreds of thousands of volunteers. Specifically we would like to derive bounds for the load on the server and measure how the load varies depending on (a) the number of volunteers; and (b) the depth and branching factor of the organization.

2. Maintain a more accurate history model for the volunteers in our organization. We can anticipate a situation where the time required to complete a job would

depend on the time of the day when the job is being run. For example, jobs run during the day when a volunteer's machine is being heavily used would take significantly longer than jobs run during the night when the volunteer is idle. We should be able to vary the number of jobs allocated to an agent depending on how its execution profile varies according to the time of the day.

3. Implement our approach on the standard BOINC server and clients.

Finally, we would like to show how existing workflow languages might benefit from the use of organizational modeling, especially the kind of modeling being done by our approach. Most existing workflow languages (like Business Process Execution Language for Web Services (BPEL4WS) [Juric, 2006], Grid-Flow [Guan *et al.* , 2005], and Grid Services Flow Language [Krishnan *et al.* , 2002]) are very procedural, in that they describe exactly how workflows are composed of their component subtasks, but do not allow alternatives workflows based on environmental conditions. As argued by [Atlas *et al.* , 2005], such approaches are fairly rigid and are not suitable for dynamic environments.

We strongly believe that grids and web services would benefit from using a more declarative approach, in which the workflow is represented using TÆMS and an approach similar to mine is used for "executing" these workflows. We would like to use a detailed example to show why this is the case.

Appendix A

AN ILLUSTRATED EXAMPLE

Figures A.1, A.2, A.3, A.4, A.5 show the stages in the organizational self-design of an organization for solving the randomly generated task structure shown in Appendix B. This task structure had a maximum depth of 3 and a maximum branching factor of 4. There was an equal probability of generating a Min CAF and a Sum CAF and the generated task structure had 10 non-local-effects (NLEs).

For generating this organization, we assumed a fixed arrival rate with both the arrival *sp-diff* multiple and the deadline *sp-diff* multiple set to 0.5[1]. This translated to a task instance being generated every 31 time cycles and the organization has 53 time cycles to complete each task instance (i.e. the deadline window is 53 cycles.) This experiment was run for 2500 time cycles.

The steps in the formation of this organization are:

- **At Time 0:** (Figure A.1) The organization starts off with a single agent, *Agent-0* that is responsible for the complete task structure.

- **At Time 5:** (Figure A.2) *Agent-0* is overloaded, so it spawns off two new agents, *Agent-1* and *Agent-2* and allocates them part of its task structure[2].

[1] We chose to use a fixed arrival rate to keep this example small and still illustrate the formation of a stable organization.

[2] *Agent-0* actually detects the overload and initiates the spawning at Time 3. The spawning has been completed at Time 5. In this example, we only show the organization after the organizational change has been completed.

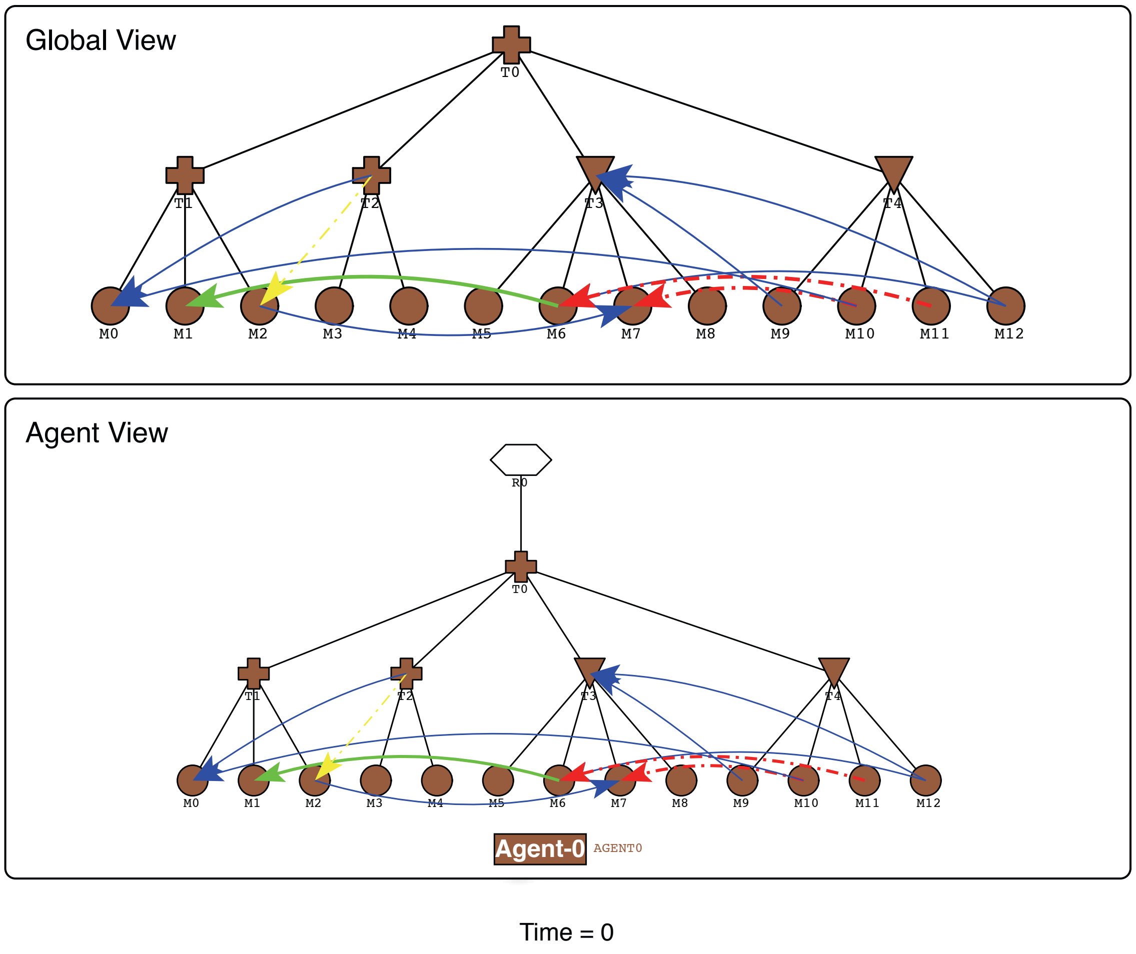


Figure A.1: Organization at **Time 0**

- **At Time 68:** (Figure A.3) *Agent-2* is overloaded, so it spawns off *Agent 3*.
- **At Time 72:** (Figure A.4) *Agent-0* is overloaded, so it spawns off *Agent 4*.
- **At Time 122:** (Figure A.5) *Agent-3* is underloaded, so it composes with *Agent-4* and ceases to exist.
- **After Time 123:** The organization was stable and no further changes to the organization were made.

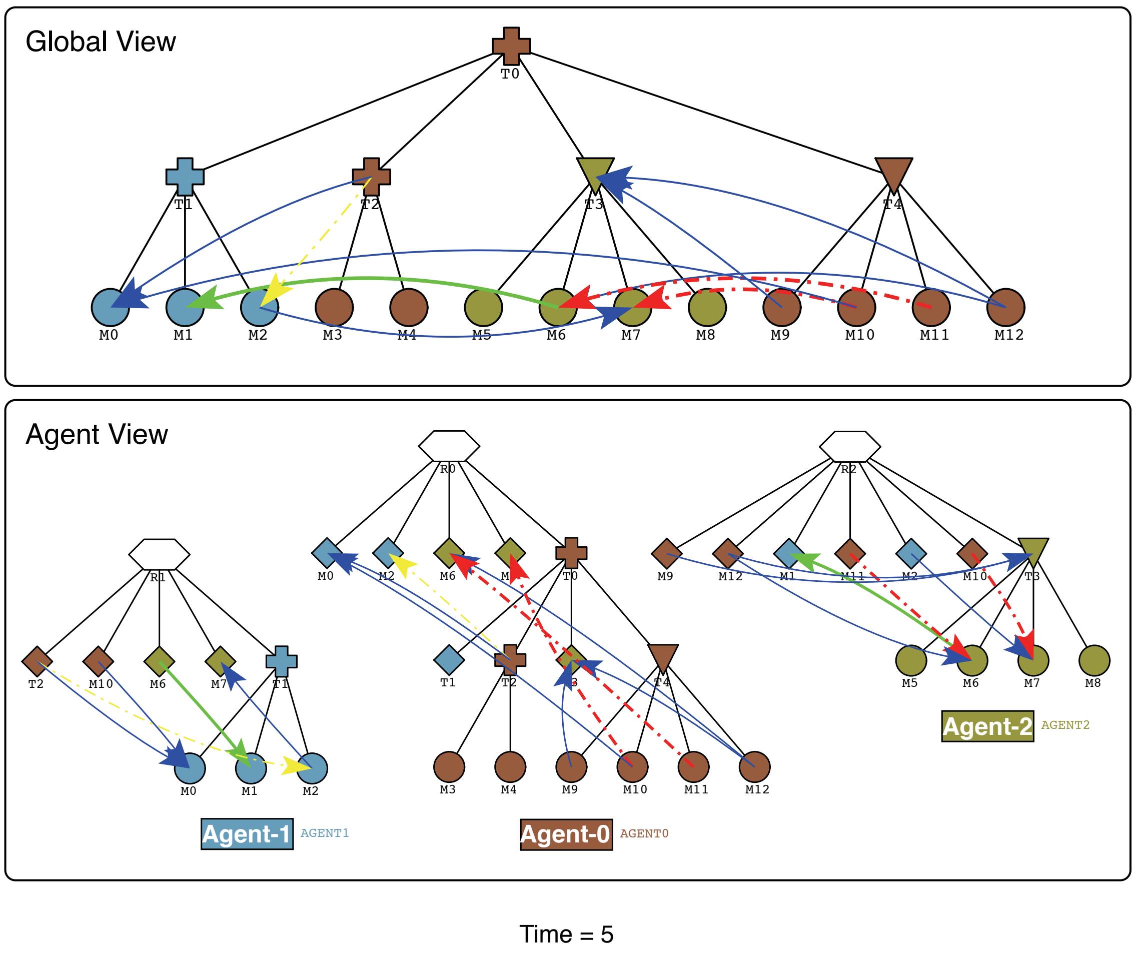
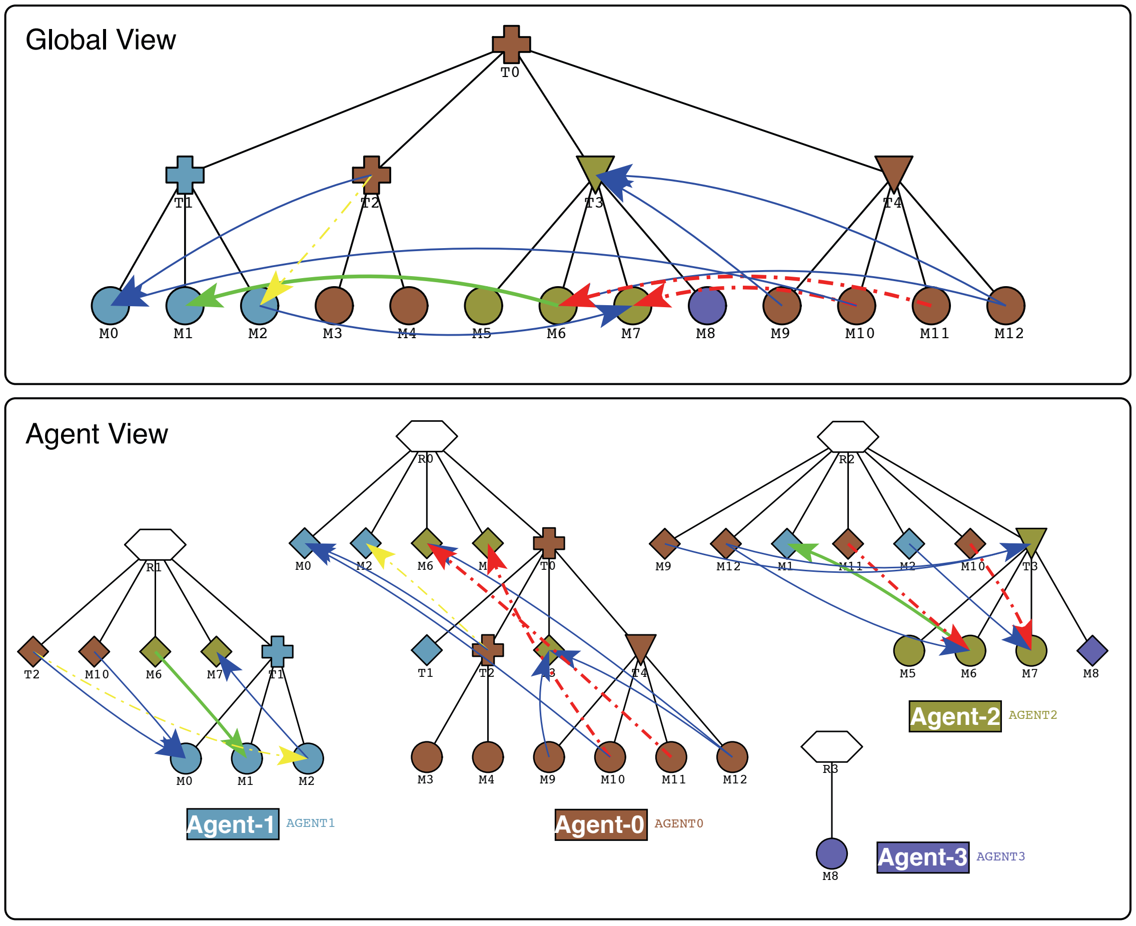


Figure A.2: Organization at **Time 5**: *Agent-0* spawns off *Agent-1* and *Agent-2*



Time = 68

Figure A.3: Organization at **Time 68**: *Agent-2* spawns *Agent-3*

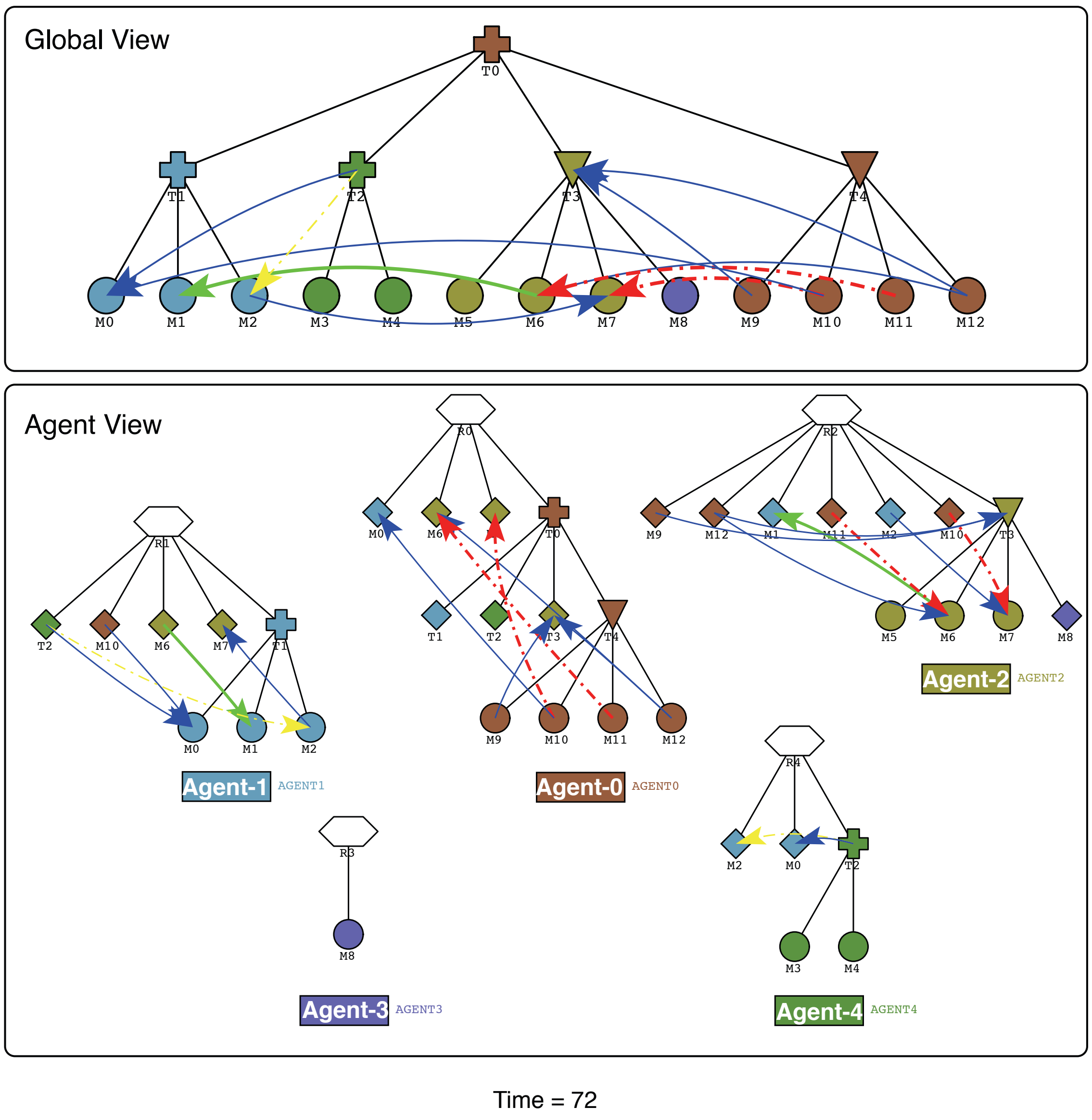


Figure A.4: Organization at **Time 72**: *Agent-0* spawns *Agent-4*

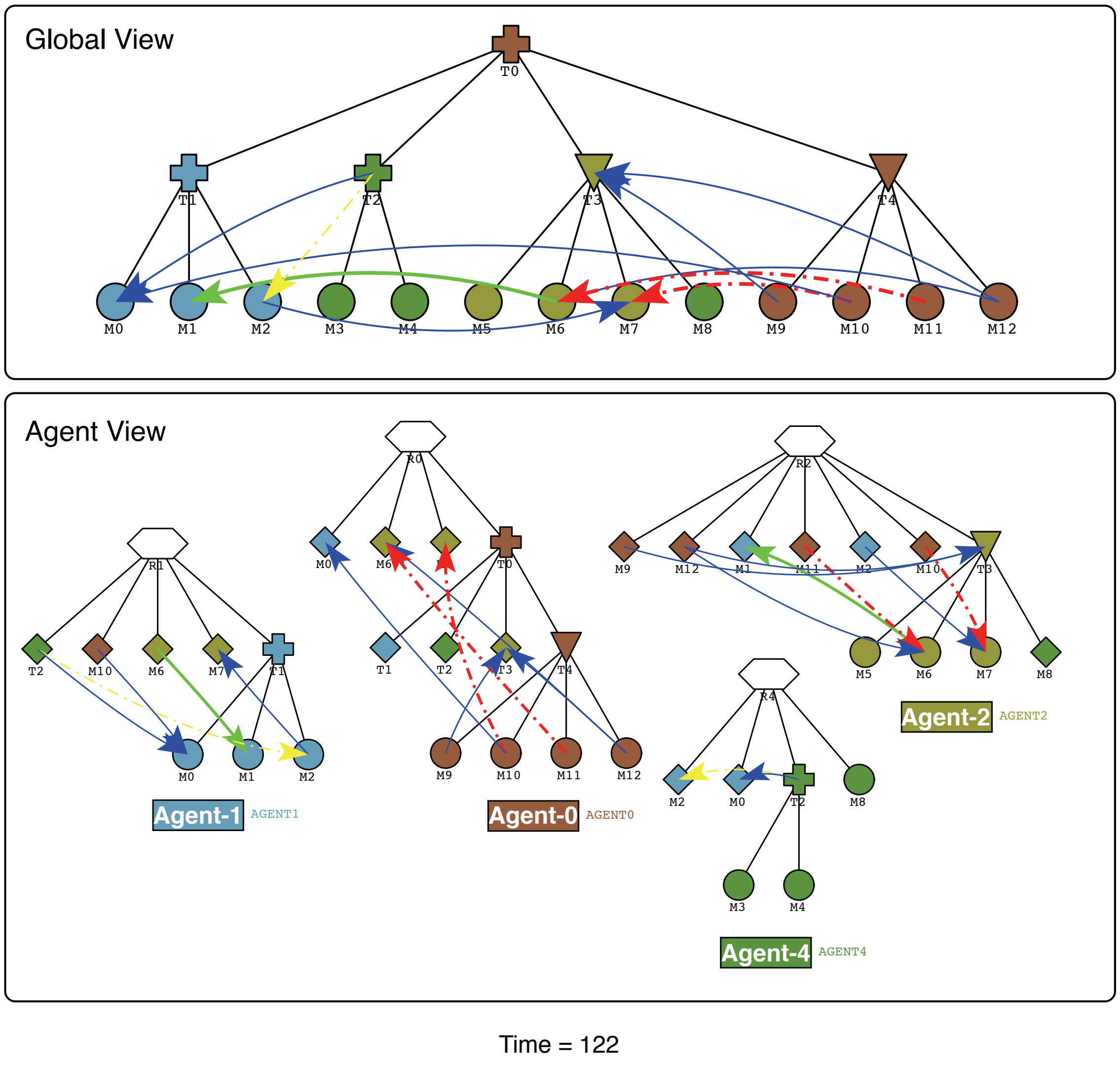


Figure A.5: Organization at **Time 122**: *Agent-3* composes with *Agent-4*

Appendix B

A RANDOM TASK STRUCTURE

This appendix describes a randomly generated task structure. The task structure shown here (see Figure B.1) was generated by specifying a maximum depth of 3 and a maximum branching factor of 4 to our simulator. There was an equal probability of generating a Min CAF and a Sum CAF and the generated task structure had 10 non-local-effects (NLEs).

The generated task structure is shown in Figure B.1 and its qualitative and quantitative characteristics are described in Tables B.1 (the tasks), B.2 (the methods) and B.6 (the non-local effects).

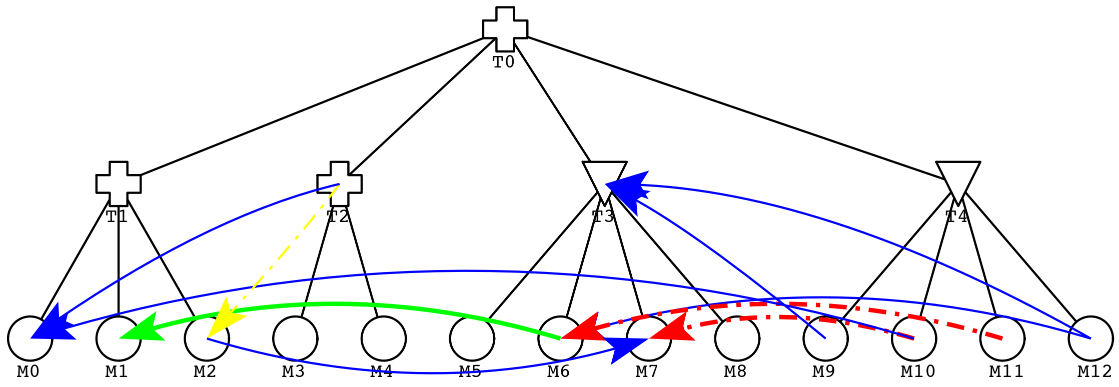


Figure B.1: Our randomly generated task structure. The polygons (T0 — T4) represent tasks and the circles (M0 — M12) represent methods. The arrows show the non-local effects (NLEs). Green represents an Enables NLE, Red represents a Disables NLE, Blue represents a Facilitates NLE and Yellow represents a Hinders NLE.

Table B.1: The set of tasks and their subtasks in our randomly generated task structure.

Task	CAF	Subtasks
T0	Sum	T1 T2 T3 T4
T1	Sum	M0 M1 M2
T2	Sum	M3 M4
T3	Min	M5 M6 M7 M8
T4	Min	M9 M10 M11 M12

Table B.2: The set of methods and their characteristics in our randomly generated task structure.

Method	Outcome	Characteristics	
M0	OUTCOME0	Probability	0.727
		Quality	((1 0.626) (9 0.01) (7 0.364))
		Cost	((7 0.873) (5 0.082) (3 0.044))
		Duration	((8 1.0))
	OUTCOME1	Probability	0.088
		Quality	((3 1.0))
		Cost	((6 1.0))
		Duration	((7 0.616) (3 0.179) (2 0.077) (1 0.127))
	OUTCOME2	Probability	0.185
		Quality	((1 1.0))
		Cost	((1 1.0))
		Duration	((10 0.706) (1 0.264) (7 0.0) (5 0.03))
M1	OUTCOME3	Probability	0.686
		Quality	((4 1.0))
		Cost	((1 0.17) (8 0.633) (10 0.19) (9 0.006))
		Duration	((2 0.226) (3 0.258) (1 0.097) (5 0.419))
	OUTCOME4	Probability	0.314
		Quality	((6 0.905) (3 0.095))
		Cost	((2 0.232) (10 0.752) (3 0.016))
		Duration	((8 1.0))
M2	OUTCOME5	Probability	0.567
		Quality	((4 0.217) (8 0.226) (5 0.244) (3 0.312))
		Cost	((9 0.112) (2 0.888))
		Duration	((9 0.875) (7 0.084) (1 0.032) (8 0.009))
	OUTCOME6	Probability	0.433
		Quality	((5 1.0))
		Cost	((8 0.602) (4 0.398))
		Duration	((8 1.0))

Table B.3: The set of methods and their characteristics in our randomly generated task structure. (Continued from the Previous Page.)

Method	Outcome	Characteristics	
M3	OUTCOME7	Probability	0.039
		Quality	((3 1.0))
		Cost	((9 1.0))
		Duration	((7 1.0))
	OUTCOME8	Probability	0.961
		Quality	((1 0.469) (6 0.191) (4 0.34))
		Cost	((5 0.025) (7 0.414) (10 0.561))
		Duration	((10 0.114) (5 0.886))
M4	OUTCOME9	Probability	1.0
		Quality	((4 1.0))
		Cost	((1 0.565) (10 0.435))
		Duration	((6 0.191) (9 0.072) (3 0.737))
M5	OUTCOME10	Probability	1.0
		Quality	((8 0.322) (7 0.039) (2 0.474) (10 0.164))
		Cost	((4 1.0))
		Duration	((6 0.578) (4 0.422))
M6	OUTCOME11	Probability	0.122
		Quality	((9 0.884) (3 0.091) (2 0.024))
		Cost	((1 0.162) (2 0.838))
		Duration	((4 0.508) (1 0.491))
	OUTCOME12	Probability	0.456
		Quality	((7 0.08) (5 0.423) (10 0.497))
		Cost	((5 0.756) (9 0.008) (4 0.236))
		Duration	((9 0.871) (10 0.129))
	OUTCOME13	Probability	0.298
		Quality	((8 0.463) (9 0.537))
		Cost	((9 1.0))
		Duration	((3 0.565) (5 0.379) (6 0.033) (1 0.022))
	OUTCOME14	Probability	0.124
		Quality	((9 0.099) (1 0.304) (3 0.597))
		Cost	((6 0.561) (7 0.075) (3 0.005) (1 0.359))
		Duration	((2 0.62) (7 0.279) (3 0.1))

Table B.4: The set of methods and their characteristics in our randomly generated task structure. (Continued from the Previous Page.)

Method	Outcome	Characteristics	
M7	OUTCOME15	Probability	0.103
		Quality	((7 0.481) (3 0.488) (9 0.022) (2 0.008))
		Cost	((5 0.932) (9 0.068))
		Duration	((6 1.0))
	OUTCOME16	Probability	0.49
		Quality	((5 0.146) (2 0.217) (3 0.087) (4 0.549))
		Cost	((8 0.978) (6 0.022))
		Duration	((9 1.0))
	OUTCOME17	Probability	0.306
		Quality	((7 1.0))
		Cost	((4 0.992) (6 0.008))
		Duration	((10 0.176) (1 0.02) (9 0.804))
	OUTCOME18	Probability	0.101
		Quality	((10 0.533) (8 0.467))
		Cost	((5 0.599) (1 0.372) (6 0.028) (2 0.001))
		Duration	((9 0.291) (6 0.088) (1 0.621))
M8	OUTCOME19	Probability	1.0
		Quality	((2 0.03) (6 0.633) (7 0.337))
		Cost	((4 0.61) (6 0.39))
		Duration	((10 0.499) (7 0.062) (4 0.438))

Table B.5: The set of methods and their characteristics in our randomly generated task structure. (Continued from the Previous Page.)

Method	Outcome	Characteristics	
M9	OUTCOME20	Probability	0.68
		Quality	((7 0.847) (5 0.02) (2 0.133))
		Cost	((7 0.702) (10 0.297))
		Duration	((8 1.0))
	OUTCOME21	Probability	0.03
		Quality	((3 0.69) (2 0.31))
		Cost	((2 0.713) (6 0.17) (10 0.015) (3 0.102))
		Duration	((6 0.546) (9 0.059) (10 0.394))
	OUTCOME22	Probability	0.143
		Quality	((3 0.07) (4 0.93))
		Cost	((7 0.361) (4 0.639))
		Duration	((4 1.0))
	OUTCOME23	Probability	0.146
		Quality	((10 0.348) (9 0.37) (1 0.282))
		Cost	((2 1.0))
		Duration	((1 1.0))
M10	OUTCOME24	Probability	1.0
		Quality	((6 0.447) (4 0.277) (3 0.086) (8 0.189))
		Cost	((7 1.0))
		Duration	((2 0.554) (3 0.445))
M11	OUTCOME25	Probability	1.0
		Quality	((6 1.0))
		Cost	((8 1.0))
		Duration	((4 1.0))
M12	OUTCOME26	Probability	0.405
		Quality	((10 0.526) (1 0.436) (5 0.023) (4 0.014))
		Cost	((5 0.131) (3 0.869))
		Duration	((8 0.145) (6 0.087) (7 0.768))
	OUTCOME27	Probability	0.277
		Quality	((9 0.893) (3 0.015) (2 0.091))
		Cost	((1 0.351) (6 0.079) (8 0.57))
		Duration	((5 1.0))
	OUTCOME28	Probability	0.318
		Quality	((8 0.823) (1 0.132) (5 0.04) (2 0.004))
		Cost	((6 0.632) (10 0.367))
		Duration	((2 0.939) (8 0.016) (10 0.044))

Table B.6: The set of Non-Local Effects (NLEs) and their characteristics in our randomly generated task structure.

NLE	Type	Source	Sink	Characteristics	
NLE0	Facilitates	M12	M6	Delay	0
				Quality-power	0.7681476
				Cost-power	0.43372154
				Duration-power	0.036467075
NLE1	Enables	M6	M1	Delay	0
NLE2	Disables	M11	M6	Delay	0
NLE3	Facilitates	M10	M0	Delay	0
				Quality-power	0.42604792
				Cost-power	0.29166973
				Duration-power	0.38953853
NLE4	Facilitates	M12	T3	Delay	0
				Quality-power	0.3149761
				Cost-power	0.6441519
				Duration-power	0.1155262
NLE5	Disables	M10	M7	Delay	0
NLE6	Facilitates	M9	T3	Delay	0
				Quality-power	0.678903
				Cost-power	0.9109988
				Duration-power	0.35027683
NLE7	Facilitates	T2	M0	Delay	0
				Quality-power	0.15140426
				Cost-power	0.079969764
				Duration-power	0.8773806
NLE8	Facilitates	M2	M7	Delay	0
				Quality-power	0.49688518
				Cost-power	0.36176348
				Duration-power	0.85883725
NLE9	Hinders	T2	M2	Delay	0
				Quality-power	0.24853396
				Cost-power	0.10359275
				Duration-power	0.08548164

B.1 TÆMS Code

The generated TÆMS code is shown below:

```
(TASK-SUM (:LABEL T0)  (:TASK-TYPE TAEMS::ROOT)
          (:SUPERTASKS NIL) (:SUBTASKS (T1 T2 T3 T4)) (:CAF-LABEL TAEMS::SUM))
(TASK-SUM (:LABEL T1)  (:TASK-TYPE TAEMS::SUBTASK)
          (:SUPERTASKS (T0)) (:SUBTASKS (M0 M1 M2)) (:CAF-LABEL TAEMS::SUM))
(EXEC-METHOD (:LABEL M0)  (:SUPERTASKS (T1))
             (:OUTCOMES
              ((OUTCOME (:LABEL OUTCOME2) (:DENSITY 0.185)
                        (:QUALITY_DISTRIBUTION ((1 1.0)))
                        (:DURATION_DISTRIBUTION
                         ((10 0.706) (1 0.264) (7 0.0) (5 0.03)))
                        (:COST_DISTRIBUTION ((1 1.0))))
               (OUTCOME (:LABEL OUTCOME1) (:DENSITY 0.088)
                        (:QUALITY_DISTRIBUTION ((3 1.0)))
                        (:DURATION_DISTRIBUTION
                         ((7 0.616) (3 0.179) (2 0.077) (1 0.127)))
                        (:COST_DISTRIBUTION ((6 1.0))))
               (OUTCOME (:LABEL OUTCOME0) (:DENSITY 0.727)
                        (:QUALITY_DISTRIBUTION ((1 0.626) (9 0.01) (7 0.364)))
                        (:DURATION_DISTRIBUTION ((8 1.0)))
                        (:COST_DISTRIBUTION ((7 0.873) (5 0.082) (3 0.044))))))
             (:EXECUTE-FN NIL) (:MEASURE-CHARACTERISTICS-FN NIL)
             (:RESOURCES ((TAEMS::RESOURCE_B ((2 0.459) (4 0.541))))))
(EXEC-METHOD (:LABEL M1)  (:SUPERTASKS (T1))
             (:OUTCOMES
              ((OUTCOME (:LABEL OUTCOME4) (:DENSITY 0.314)
                        (:QUALITY_DISTRIBUTION ((6 0.905) (3 0.095)))
                        (:DURATION_DISTRIBUTION ((8 1.0)))
                        (:COST_DISTRIBUTION ((2 0.232) (10 0.752) (3 0.016))))
               (OUTCOME (:LABEL OUTCOME3) (:DENSITY 0.686)
                        (:QUALITY_DISTRIBUTION ((4 1.0)))
                        (:DURATION_DISTRIBUTION
                         ((2 0.226) (3 0.258) (1 0.097) (5 0.419)))
                        (:COST_DISTRIBUTION
                         ((1 0.17) (8 0.633) (10 0.19) (9 0.006))))))
             (:EXECUTE-FN NIL) (:MEASURE-CHARACTERISTICS-FN NIL)
             (:RESOURCES
               ((TAEMS::CPU ((5 0.909) (3 0.091)))
                (TAEMS::RESOURCE_D ((5 0.275) (1 0.725))))))
(EXEC-METHOD (:LABEL M2)  (:SUPERTASKS (T1))
             (:OUTCOMES
              ((OUTCOME (:LABEL OUTCOME6) (:DENSITY 0.433)
                        (:QUALITY_DISTRIBUTION ((5 1.0)))
                        (:DURATION_DISTRIBUTION ((8 1.0)))
                        (:COST_DISTRIBUTION ((8 0.602) (4 0.398))))
               (OUTCOME (:LABEL OUTCOME5) (:DENSITY 0.567)
                        (:QUALITY_DISTRIBUTION
                         ((4 0.217) (8 0.226) (5 0.244) (3 0.312)))
                        (:DURATION_DISTRIBUTION
                         ((9 0.875) (7 0.084) (1 0.032) (8 0.009)))
                        (:COST_DISTRIBUTION ((9 0.112) (2 0.888))))))
             (:EXECUTE-FN NIL) (:MEASURE-CHARACTERISTICS-FN NIL)
             (:RESOURCES ((TAEMS::RESOURCE_B ((5 0.448) (3 0.552))))))
(NLE-FACILITATES (:LABEL NLE8) (:SOURCE M2) (:SINK M7) (:DELAY 0)
                 (:QUALITY_POWER 0.49688518)
                 (:COST_POWER 0.36176348)
                 (:DURATION_POWER 0.85883725) (:MAX-SOURCE-QUALITY 8))
(TASK-SUM (:LABEL T2)  (:TASK-TYPE TAEMS::SUBTASK)
          (:SUPERTASKS (T0)) (:SUBTASKS (M3 M4)) (:CAF-LABEL TAEMS::SUM))
(EXEC-METHOD (:LABEL M3)  (:SUPERTASKS (T2))
             (:OUTCOMES
              ((OUTCOME (:LABEL OUTCOME8) (:DENSITY 0.961)
                        (:QUALITY_DISTRIBUTION ((1 0.469) (6 0.191) (4 0.34)))
                        (:DURATION_DISTRIBUTION ((10 0.114) (5 0.886)))
                        (:COST_DISTRIBUTION ((5 0.025) (7 0.414) (10 0.561))))
               (OUTCOME (:LABEL OUTCOME7) (:DENSITY 0.039)
                        (:QUALITY_DISTRIBUTION ((3 1.0)))
                        (:DURATION_DISTRIBUTION ((7 1.0)))
                        (:COST_DISTRIBUTION ((9 1.0))))))
             (:EXECUTE-FN NIL) (:MEASURE-CHARACTERISTICS-FN NIL)
             (:RESOURCES ((TAEMS::RESOURCE_A ((1 0.78) (2 0.22))))))
(EXEC-METHOD (:LABEL M4)  (:SUPERTASKS (T2))
             (:OUTCOMES
              ((OUTCOME (:LABEL OUTCOME9) (:DENSITY 1.0)
                        (:QUALITY_DISTRIBUTION ((4 1.0)))
                        (:DURATION_DISTRIBUTION ((6 0.191) (9 0.072) (3 0.737)))
                        (:COST_DISTRIBUTION ((1 0.565) (10 0.435))))))
             (:EXECUTE-FN NIL) (:MEASURE-CHARACTERISTICS-FN NIL)
             (:RESOURCES
              ((TAEMS::RESOURCE_A ((5 0.342) (2 0.658)))
               (TAEMS::RESOURCE_C ((3 0.562) (1 0.438))))))
(NLE-HINDERS (:LABEL NLE9) (:SOURCE T2) (:SINK M2) (:DELAY 0)
             (:QUALITY_POWER 0.24853396)
             (:COST_POWER 0.10359275)
             (:DURATION_POWER 0.08548164)
             (:MAX-SOURCE-QUALITY 10))
(NLE-FACILITATES (:LABEL NLE7) (:SOURCE T2) (:SINK M0) (:DELAY 0)
                 (:QUALITY_POWER 0.15140426)
                 (:COST_POWER 0.079969764)
                 (:DURATION_POWER 0.8773806)
                 (:MAX-SOURCE-QUALITY 10))
(TASK-MIN (:LABEL T3)  (:TASK-TYPE TAEMS::SUBTASK)
          (:SUPERTASKS (T0)) (:SUBTASKS (M5 M6 M7 M8)) (:CAF-LABEL MIN))
(EXEC-METHOD (:LABEL M5)  (:SUPERTASKS (T3))
             (:OUTCOMES
              ((OUTCOME (:LABEL OUTCOME10) (:DENSITY 1.0)
                        (:QUALITY_DISTRIBUTION
                         ((8 0.322) (7 0.039) (2 0.474) (10 0.164)))
                        (:DURATION_DISTRIBUTION ((6 0.578) (4 0.422)))
                        (:COST_DISTRIBUTION ((4 1.0))))))
             (:EXECUTE-FN NIL) (:MEASURE-CHARACTERISTICS-FN NIL)
             (:RESOURCES ((TAEMS::RESOURCE_D ((3 0.139) (1 0.861))))))
(EXEC-METHOD (:LABEL M6)  (:SUPERTASKS (T3))
             (:OUTCOMES
              ((OUTCOME (:LABEL OUTCOME14) (:DENSITY 0.124)
                        (:QUALITY_DISTRIBUTION
                         ((9 0.099) (1 0.304) (3 0.597)))
                        (:DURATION_DISTRIBUTION
                         ((2 0.62) (7 0.279) (3 0.1)))
                        (:COST_DISTRIBUTION
                         ((6 0.561) (7 0.075) (3 0.005) (1 0.359))))
               (OUTCOME (:LABEL OUTCOME13) (:DENSITY 0.298)
                        (:QUALITY_DISTRIBUTION ((8 0.463) (9 0.537)))
                        (:DURATION_DISTRIBUTION
                         ((3 0.565) (5 0.379) (6 0.033) (1 0.022)))
                        (:COST_DISTRIBUTION ((9 1.0))))
               (OUTCOME (:LABEL OUTCOME12) (:DENSITY 0.456)
                        (:QUALITY_DISTRIBUTION
                         ((7 0.08) (5 0.423) (10 0.497)))
                        (:DURATION_DISTRIBUTION ((9 0.871) (10 0.129)))
                        (:COST_DISTRIBUTION
                         ((5 0.756) (9 0.008) (4 0.236))))
               (OUTCOME (:LABEL OUTCOME11) (:DENSITY 0.122)
                        (:QUALITY_DISTRIBUTION
                         ((9 0.884) (3 0.091) (2 0.024)))
                        (:DURATION_DISTRIBUTION ((4 0.508) (1 0.491)))
                        (:COST_DISTRIBUTION ((1 0.162) (2 0.838))))))
             (:EXECUTE-FN NIL) (:MEASURE-CHARACTERISTICS-FN NIL)
             (:RESOURCES ((TAEMS::RESOURCE_D ((3 0.724) (2 0.276))))))
(NLE-ENABLES (:LABEL NLE1) (:SOURCE M6) (:SINK M1) (:DELAY 0))
(EXEC-METHOD (:LABEL M7)  (:SUPERTASKS (T3))
             (:OUTCOMES
              ((OUTCOME (:LABEL OUTCOME18) (:DENSITY 0.101)
                        (:QUALITY_DISTRIBUTION ((10 0.533) (8 0.467)))
                        (:DURATION_DISTRIBUTION
                         ((9 0.291) (6 0.088) (1 0.621)))
                        (:COST_DISTRIBUTION
                         ((5 0.599) (1 0.372) (6 0.028) (2 0.001))))
               (OUTCOME (:LABEL OUTCOME17) (:DENSITY 0.306)
                        (:QUALITY_DISTRIBUTION ((7 1.0)))
                        (:DURATION_DISTRIBUTION
                         ((10 0.176) (1 0.02) (9 0.804)))
                        (:COST_DISTRIBUTION ((4 0.992) (6 0.008))))
               (OUTCOME (:LABEL OUTCOME16) (:DENSITY 0.49)
                        (:QUALITY_DISTRIBUTION
                         ((5 0.146) (2 0.217) (3 0.087) (4 0.549)))
                        (:DURATION_DISTRIBUTION ((9 1.0)))
                        (:COST_DISTRIBUTION ((8 0.978) (6 0.022))))
               (OUTCOME (:LABEL OUTCOME15) (:DENSITY 0.103)
                        (:QUALITY_DISTRIBUTION
                         ((7 0.481) (3 0.488) (9 0.022) (2 0.008)))
                        (:DURATION_DISTRIBUTION ((6 1.0)))
                        (:COST_DISTRIBUTION ((5 0.932) (9 0.068))))))
             (:EXECUTE-FN NIL) (:MEASURE-CHARACTERISTICS-FN NIL)
             (:RESOURCES
              ((TAEMS::CPU ((5 0.259) (2 0.741)))
               (TAEMS::RESOURCE_C ((4 0.043) (3 0.957))))))
(EXEC-METHOD (:LABEL M8)  (:SUPERTASKS (T3))
             (:OUTCOMES
              ((OUTCOME (:LABEL OUTCOME19) (:DENSITY 1.0)
                        (:QUALITY_DISTRIBUTION
                         ((2 0.03) (6 0.633) (7 0.337)))
                        (:DURATION_DISTRIBUTION
                         ((10 0.499) (7 0.062) (4 0.438)))
                        (:COST_DISTRIBUTION ((4 0.61) (6 0.39))))))
             (:EXECUTE-FN NIL) (:MEASURE-CHARACTERISTICS-FN NIL)
             (:RESOURCES
              ((TAEMS::RESOURCE_A ((4 0.564) (3 0.436)))
               (TAEMS::RESOURCE_C ((4 0.759) (5 0.241))))))
(TASK-MIN (:LABEL T4)  (:TASK-TYPE TAEMS::SUBTASK)
          (:SUPERTASKS (T0)) (:SUBTASKS (M9 M10 M11 M12)) (:CAF-LABEL MIN))
(EXEC-METHOD (:LABEL M9)  (:SUPERTASKS (T4))
             (:OUTCOMES
              ((OUTCOME (:LABEL OUTCOME23) (:DENSITY 0.146)
                        (:QUALITY_DISTRIBUTION
                         ((10 0.348) (9 0.37) (1 0.282)))
                        (:DURATION_DISTRIBUTION ((1 1.0)))
                        (:COST_DISTRIBUTION ((2 1.0))))
               (OUTCOME (:LABEL OUTCOME22) (:DENSITY 0.143)
                        (:QUALITY_DISTRIBUTION ((3 0.07) (4 0.93)))
                        (:DURATION_DISTRIBUTION ((4 1.0)))
                        (:COST_DISTRIBUTION ((7 0.361) (4 0.639))))
               (OUTCOME (:LABEL OUTCOME21) (:DENSITY 0.03)
                        (:QUALITY_DISTRIBUTION ((3 0.69) (2 0.31)))
                        (:DURATION_DISTRIBUTION
                         ((6 0.546) (9 0.059) (10 0.394)))
                        (:COST_DISTRIBUTION
                         ((2 0.713) (6 0.17) (10 0.015) (3 0.102))))
               (OUTCOME (:LABEL OUTCOME20) (:DENSITY 0.68)
                        (:QUALITY_DISTRIBUTION
                         ((7 0.847) (5 0.02) (2 0.133)))
                        (:DURATION_DISTRIBUTION ((8 1.0)))
                        (:COST_DISTRIBUTION ((7 0.702) (10 0.297))))))
             (:EXECUTE-FN NIL) (:MEASURE-CHARACTERISTICS-FN NIL)
             (:RESOURCES
              ((TAEMS::RESOURCE_A ((5 0.277) (1 0.723)))
               (TAEMS::RESOURCE_D ((1 0.732) (5 0.268))))))
(NLE-FACILITATES (:LABEL NLE6) (:SOURCE M9) (:SINK T3) (:DELAY 0)
                 (:QUALITY_POWER 0.678903)
                 (:COST_POWER 0.9109988)
                 (:DURATION_POWER 0.35027683)
                 (:MAX-SOURCE-QUALITY 10))
(EXEC-METHOD (:LABEL M10)  (:SUPERTASKS (T4))
             (:OUTCOMES
              ((OUTCOME (:LABEL OUTCOME24) (:DENSITY 1.0)
                        (:QUALITY_DISTRIBUTION
                         ((6 0.447) (4 0.277) (3 0.086) (8 0.189)))
                        (:DURATION_DISTRIBUTION ((2 0.554) (3 0.445)))
                        (:COST_DISTRIBUTION ((7 1.0))))))
             (:EXECUTE-FN NIL) (:MEASURE-CHARACTERISTICS-FN NIL)
             (:RESOURCES
              ((TAEMS::CPU ((4 0.975) (5 0.024)))
               (TAEMS::RESOURCE_B ((2 0.407) (1 0.593))))))
(NLE-DISABLES (:LABEL NLE5) (:SOURCE M10) (:SINK M7) (:DELAY 0))
(NLE-FACILITATES (:LABEL NLE3) (:SOURCE M10) (:SINK M0) (:DELAY 0)
                 (:QUALITY_POWER 0.42604792)
                 (:COST_POWER 0.29166973)
                 (:DURATION_POWER 0.38953853)
                 (:MAX-SOURCE-QUALITY 8))
(EXEC-METHOD (:LABEL M11)  (:SUPERTASKS (T4))
             (:OUTCOMES
              ((OUTCOME (:LABEL OUTCOME25) (:DENSITY 1.0)
                        (:QUALITY_DISTRIBUTION ((6 1.0)))
                        (:DURATION_DISTRIBUTION ((4 1.0)))
                        (:COST_DISTRIBUTION ((8 1.0))))))
             (:EXECUTE-FN NIL) (:MEASURE-CHARACTERISTICS-FN NIL)
             (:RESOURCES
              ((TAEMS::CPU ((5 0.064) (4 0.936)))
               (TAEMS::RESOURCE_B ((2 0.65) (4 0.35))))))
(NLE-DISABLES (:LABEL NLE2)
              (:SOURCE M11)
              (:SINK M6)
              (:DELAY 0))
(EXEC-METHOD (:LABEL M12) (:SUPERTASKS (T4))
             (:OUTCOMES
              ((OUTCOME (:LABEL OUTCOME28) (:DENSITY 0.318)
                        (:QUALITY_DISTRIBUTION
                         ((8 0.823) (1 0.132) (5 0.04) (2 0.004)))
                        (:DURATION_DISTRIBUTION
                         ((2 0.939) (8 0.016) (10 0.044)))
                        (:COST_DISTRIBUTION ((6 0.632) (10 0.367))))
               (OUTCOME (:LABEL OUTCOME27) (:DENSITY 0.277)
                        (:QUALITY_DISTRIBUTION
                         ((9 0.893) (3 0.015) (2 0.091)))
                        (:DURATION_DISTRIBUTION ((5 1.0)))
                        (:COST_DISTRIBUTION
                         ((1 0.351) (6 0.079) (8 0.57))))
               (OUTCOME (:LABEL OUTCOME26) (:DENSITY 0.405)
                        (:QUALITY_DISTRIBUTION
                         ((10 0.526) (1 0.436) (5 0.023) (4 0.014)))
                        (:DURATION_DISTRIBUTION
                         ((8 0.145) (6 0.087) (7 0.768)))
                        (:COST_DISTRIBUTION ((5 0.131) (3 0.869))))))
             (:EXECUTE-FN NIL) (:MEASURE-CHARACTERISTICS-FN NIL)
             (:RESOURCES
              ((TAEMS::RESOURCE_A ((1 0.696) (2 0.304)))
               (TAEMS::RESOURCE_D ((2 0.66) (5 0.339))))))
(NLE-FACILITATES (:LABEL NLE4) (:SOURCE M12) (:SINK T3) (:DELAY 0)
                 (:QUALITY_POWER 0.3149761)
                 (:COST_POWER 0.6441519)
                 (:DURATION_POWER 0.1155262)
                 (:MAX-SOURCE-QUALITY 10))
(NLE-FACILITATES (:LABEL NLE0) (:SOURCE M12) (:SINK M6) (:DELAY 0)
                 (:QUALITY_POWER 0.7681476)
                 (:COST_POWER 0.43372154)
                 (:DURATION_POWER 0.036467075)
                 (:MAX-SOURCE-QUALITY 10))
```

Bibliography

AMMAN, H.M., KENDRICK, D.A., & RUST, J. (eds). 1996. *Handbook of Computational Economics*. Elsevier Science Ltd.

ANDERSON, DAVID P. 2004. BOINC: A System for Public-Resource Computing and Storage. *Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, 4–10.

ANDERSON, DAVID P., & REED, KEVIN. 2009. Celebrating Diversity in Volunteer Computing. *Proceedings of the Hawaii International Conference on System Sciences (HICSS)*.

ARTIKIS, ALEXANDER, KAPONIS, DIMOSTHENIS, & PITT, JEREMY. 2009. Dynamic Specifications for Norm-Governed Computational Systems. *In:* DIGNUM, VIRGINIA (ed), *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models*. Information Science Reference. IGI Global.

ATLAS, JAMES. 2009 (August). *Efficient Coordination Techniques for Non-deterministic Multi-agent Systems using Distributed Constraint Optimization*. Ph.D. thesis, University of Delaware, Newark, DE, USA. Adviser-Keith S. Decker.

ATLAS, JAMES, SWANY, MARTIN, & DECKER, KEITH S. 2005. Flexible Grid Workflows Using TAEMS. *Workshop on Exploring Planning and Scheduling for Web Services, Grid and Autonomic Computing*, July.

Barber, K. S., & Martin, C. E. 2001. Dynamic reorganization of decision-making groups. *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, 513–520.

Bernon, Carole, Chevrier, Vincent, Hilaire, Vincent, & Marrow, Paul. 2006. Applications of Self-Organising Multi-Agent Systems: An Initial Framework for Comparison. *Informatica*, **30**(1), 73–82.

Boella, Guido, Pigozzi, Gabriella, & van der Torre, Leendert. 2009. Normative Framework for Normative System Change. *Proceedings of the Eighth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, 169–176.

Bonabeau, Eric, Dorigo, Marco, & Theraulaz, Guy. 1999. *Swarm Intelligence: From Natural to Artificial Systems.* Santa Fe Institute Studies in the Sciences of Complexity Proceedings. Oxford University Press.

Briot, J.-P., Guessoum, Z., Aknine, S., Almeida, A. L., Malenfant, J., Marin, O., Sens, P., Faci, N., Gatti, M., & Lucena, C. 2006. Experience and prospects for various control strategies for self-replicating multi-agent systems. *SEAMS '06: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, 37–43.

Budati, Krishnaveni, Sonnek, Jason, Chandra, Abhishek, & Weissman, Jon. 2007. Ridge: combining reliability and performance in open grid platforms. *Pages 55–64 of: HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing.* New York, NY, USA: ACM.

Burton, Richard M., & Obel, Borge. 1984. *Designing Efficient Organizations.* Advanced Series in Management, vol. 7. Elsevier Science Ltd.

Carley, Kathleen M. 2002a. Computational organizational science and organizational engineering. *Simulation Modelling Practice and Theory*, **10**(July), 253–269.

Carley, Kathleen M. 2002b. Smart Agents and Organizations of the Future. *The Handbook of New Media*, 206—220.

Carley, Kathleen M., & Gasser, Les. 1999. Computational Organization Theory. *Pages 299–330 of:* Wiess, Gerhard (ed), *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. Cambridge, MA: MIT Press.

Carley, Kathleen M., & Wallace, W.A. (eds). *Computational and Mathematical Organization Theory*. Springer US.

Cassandra, Anthony R. 1998. *Exact and Approximate Algorithms for Partially Observable Markov Decision Processes*. Ph.D. thesis, Department of Computer Science, Brown University, Providence, RI.

Chang, Myong-Hun, & Harrington, Joseph E. 2006. Agent-Based Models of Organizations. *Handbook of Computational Economics*, **2**(26), 1273–1337.

Chen, Wei, & Decker, Keith S. 2005. The Analysis of Coordination in An Information System Application - Emergency Medical Services. *Lecture Notes in Computer Science (LNCS)*, May, 36–51.

Chevaleyre, Yann, Dunne, Paul E., Endriss, Ulle, Lang, Jérôme, Lemaître, Michel, Maudet, Nicolas, Padget, Julian, Phelps, Steve, Rodríguez-aguilar, Juan A., & Sousa, Paulo. 2006. Issues in Multiagent Resource Allocation. *Informatica*, **30**, 2006.

Corkill, Daniel, & Lesser, Victor. 1983. The use of meta-level control for coordination in a distributed problem solving network. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, August, 748–756.

Costa, Fernando, Silva, Luis, Kelley, Ian, & Taylor, Ian. 2008. Peer-To-Peer Techniques for Data Distribution in Desktop Grid Computing Platforms. *Making Grids Work*, 377–391.

Coutinho, Luciano R., Sichman, Jaime S., & Boissier, Olivier. 2007. Modeling Dimensions for Multiagent System Organizations. *IJCAI 2007 Workshop on Agent Organizations: Models and Simulations (AOMS)*, January.

Daniel, Wayne W. 2000. *Applied Nonparametric Statistics*. 2nd edition edn. Duxbury Press.

Decker, Keith S. 1995. Environment Centered Analysis and Design of Coordination Mechanisms. *Ph.D. Thesis, Department of Computer Science, University of Massachusetts, Amherst*, May.

Decker, Keith S. 1996. TAEMS: A Framework for Environment Centered Analysis & Design of Coordination Mechanisms. *Pages 429–448 of: Foundations of Distributed Artificial Intelligence, Chapter 16*. G. O'Hare and N. Jennings (eds.), Wiley Inter-Science.

Decker, Keith S., & Li, Jinjiang. 2000. Coordinating Mutually Exclusive resources using GPGP. *Autonomous Agents and Multi-Agent Systems*, **3**(2), 133–157.

Decker, Keith S., Durfee, Edmund H., & Lesser, Victor R. 1989. Evaluating Research in Cooperative Distributed Problem Solving. *Pages 487–519 of:* Gasser, L., & Huhns, M. (eds), *Distributed Artificial Intelligence Volume II*. Pitman Publishing: London and Morgan Kaufmann: San Mateo, CA.

Decker, Keith S., Sycara, Katia P., & Williamson, Mike. 1997. Cloning for Intelligent Adaptive Information Agents. *Pages 63–75 of: Revised Papers from the Second Australian Workshop on Distributed Artificial Intelligence.* London, UK: Springer-Verlag.

Dellarocas, C., & Klein, M. 2000. An Experimental Evaluation of Domain-Independent Fault Handling Services in Open Multi-Agent Systems. *Proceedings of the International Conference on Multi-Agent Systems (ICMAS-2000)*, July.

Dellarocas, Chrysanthos, & Klein, Mark. 1999. Civil Agent Societies: Tools for Inventing Open Agent-Mediated Electronic Marketplaces. *Pages 24–39 of: IJCAI 1999 Workshop on Agent Mediated Electronic Commerce.*

DeLoach, Scott, Oyenan, Walamitien, & Matson, Eric. 2008. A capabilities-based model for adaptive organizations. *Autonomous Agents and Multi-Agent Systems*, **16**(1), 13–56.

DeLoach, Scott A. 2009. OMACS: A Framework for Adaptive, Complex Systems. *Pages 76–104 of:* Dignum, Virginia (ed), *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models.* Information Science Reference. IGI Global.

Dignum, Virginia, Dignum, Frank, & Sonenberg, Liz. 2004 (September). Towards Dynamic Reorganization of Agent Societies. *Pages 22–27 of: Proceedings of CEAS: Workshop on Coordination in Emergent Agent Societies at ECAI.*

Dignum, Virginia, Vazquez-Salceda, Javier, & Dignum, Frank. 2005. *OMNI: Introducing Social Structure, Norms and Ontologies into Agent Organizations.*

dos Reis Coutinho, Luciano, Sichman, Jaime Simao, & Boissier, Olivier. 2005. Modeling organization in MAS: a comparison of models. *1st. Workshop on Software Engineering for Agent-Oriented Systems (SEAS'05)*, October.

Estrada, Trilce, Taufer, Michela, & Reed, Kevin. 2009. Prediction of Upper Time Bounds from Volunteer Computing Traces. *IEEE*.

Fatima, S. Shaheen, & Wooldridge, Michael. 2001. Adaptive task and resources allocation in multi-agent systems. *Pages 537–544 of: AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*. New York, NY, USA: ACM Press.

Ferber, J., Michel, F., & Baez, J. 2005. *AGRE: Integrating Environments with Organizations*. Springer Berlin / Heidelberg.

Ferber, Jacques, Gutknecht, Olivier, & Michel, Fabien. 2004. *From Agents to Organizations: An Organizational View of Multi-agent Systems*. Springer Berlin / Heidelberg.

Forno, Arianna Dal, & Merlone, Ugo. 2002. A multi-agent simulation platform for modeling perfectly rational and bounded-rational agents in organizations. *Journal of Artificial Societies and Social Simulation*, **5**(2).

Foster, I., Kesselman, C., & Tuecke, S. 2001. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, **15**(3).

Fox, Mark S. 1979 (December). *Organization structuring: Designing large complex software*. Tech. rept. 79–155. Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Fox, Mark S. 1981. An Organizational View of Distributed Systems. *IEEE Transactions on Systems, Man, and Cybernetics*, **11**(1), 70–80.

Gardner, Martin. 1985. *Wheels, Life, and Other Mathematical Amusements.* W.H. Freeman and Company.

Garvey, A., & Lesser, V. 1996. Design-to-time Scheduling and Anytime Algorithms. *SIGART Bulletin*, **7**(3).

Gasser, Les, & Ishida, Toru. 1991. A Dynamic Organizational Architecture for Adaptive Problem Solving. *Pages 185–190 of: Proceedings of the Ninth National Conference on Artificial Intelligence.* Menlo Park, California: AAAI Press.

Ghallab, Malik, Nau, Dana, & Traverso, Paolo. 2004. *Automated Planning: Theory and Practice.* Morgan Kaufmann Publishers.

Goldman, Claudia V., & Rosenschein, Jeffrey S. 1997. *Evolving organizations of agents.*

Guan, Zhijie, Hernandez, Francisco, Bangalore, Purushotham, Gray, Jeff, Skjellum, Anthony, Velusamy, Vijay, & Liu, Yin. 2005. Grid-Flow: A Grid-enabled scientific workflow system with a Petri-net-based interface. *Concurrency and Computation: Practice and Experience*, **18**(10), 1115–1140.

Guttman, Robert H., Moukas, Alexandros G., & Maes, Pattie. 1998. Agent-mediated electronic commerce: a survey. *Knowledge Engineering Review*, **13**(2), 147–159.

Horling, Bryan. 2006 (February). *Quantitative Organizational Modeling and Design for Multi-Agent Systems.* Ph.D. thesis, University of Massachusetts at Amherst.

Horling, Bryan, & Lesser, Victor. 2005a. Analyzing, Modeling and Predicting Organizational Effects in a Distributed Sensor Network. *Journal of the Brazilian Computer Society, Special Issue on Agents Organizations*, July, 9–30.

Horling, Bryan, & Lesser, Victor. 2005b. A Survey of Multi-Agent Organizational Paradigms. *Knowledge Engineering Review.*

Horling, Bryan, & Lesser, Victor. 2005c. Using ODML to Model Organizations for Multi-Agent Systems. *Pages 72–80 of: Proceedings of the 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2005).* Compiegne, France: IEEE Computer Society.

Horling, Bryan, Lesser, Victor, Vincent, Regis, Wagner, Tom, Raja, Anita, Zhang, Shelley, Decker, Keith, & Garvey, Alan. 1999 (January). *The TAEMS White Paper.*

Horling, Bryan, Benyo, Brett, & Lesser, Victor. 2001. Using self-diagnosis to adapt organizational structures. *Pages 529–536 of: AGENTS '01: Proceedings of the fifth international conference on Autonomous agents.* New York, NY, USA: ACM Press.

Huynh, Trung Dong. 2006 (June). *Trust and Reputation in Open Multi-Agent Systems.* Ph.D. thesis, University of Southampton, Electronics and Computer Science.

Ishida, T., Gasser, L., & Yokoo, M. 1992. Organization Self-Design of Distributed Production Systems. *IEEE Transactions on Knowledge and Data Engineering*, **4**(2), 123–134.

Jennings, Nicholas R., Faratin, P., Lomuscio, A. R., Parsons, S., Wooldridge, Michael, & Sierra, Carles. 2001. Automated Negotiation:

Prospects, Methods and Challenges. *Group Decision and Negotiation*, **10**(2), 199–215.

Juric, Matjaz B. 2006. *Business Process Execution Language for Web Services: BPEL and BPEL4WS*. 2nd edition edn. Packt Publishing.

Kamboj, Sachin. 2009. Analyzing the tradeoffs between breakup and cloning in the context of organizational self-design. *Proceedings of the Seventh International Joint Conference on Autonomous Agents and Multiagent Systems*, May.

Kamboj, Sachin, & Decker, Keith S. 2007a. Organizational Self-Design in Semi-dynamic Environments. *IJCAI 2007 Workshop on Agent Organizations: Models and Simulations (AOMS)*, January.

Kamboj, Sachin, & Decker, Keith S. 2007b. Organizational Self-Design in Semi-dynamic Environments. *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*, May, 1220–1227.

Kamboj, Sachin, & Decker, Keith S. 2009a. Exploring Robustness in the context of Organizational Self-Design. *In:* Hubner, Jomi, Matson, Eric, Boissier, Olivier, & Dignum, Virginia (eds), *Coordination, Organizations, Institutions, and Norms in Agent Systems IV*. Lecture Notes in Computer Science. Springer-Verlag.

Kamboj, Sachin, & Decker, Keith S. 2009b. Organizational Self-Design in Worth-Oriented Domains. *Chap. 22, pages 541–568 of:* Dignum, Virginia (ed), *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models*. Information Science Reference. IGI Global.

Kinnebrew, John S., Biswas, Gautam, Shankaran, Nishanth, Schmidt, Douglas C., & Suri, Dipa. 2008. Integrating Task Allocation, Planning, Scheduling, and Adaptive Resource Management to Support Autonomy in a Global Sensor Web. *NASA Science and Technology Conference.*

Klusch, M., & Gerber, A. 2002. Dynamic coalition formation among rational agents. *IEEE Intelligent Systems*, **17**(3), 42–47.

Koifman, Gabi, Shehory, Onn, & Gal, Avigdor. 2004. Negotiation-Based Price Discrimination for Information Goods. *Pages 679–686 of: AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems.* Washington, DC, USA: IEEE Computer Society.

Kota, Ramachandra, Gibbins, Nicholas, & Jennings, Nick. 2008. Decentralised structural adaptation in agent organisations. *Proc. AAMAS Workshop on Organised Adaptation in Multi-Agent Systems*, May, 1–16.

Kota, Ramachandra, Gibbins, Nicholas, & Jennings, Nicholas R. 2009. Self-Organising Agent Organisations. *The Eighth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, May.

Kraus, Sarit. 2001. *Strategic Negotiation in Multiagent Environments (Intelligent Robotics and Autonomous Agents).* The MIT Press.

Krishnan, Sriram, Wagstrom, Patrick, & von Laszewski, Gregor. 2002. GSFL: A Workflow Framework For Grid Services. *ANL/MCS-P980-0802.*

Krupansky, Jack. 2005 (November). *Closed Multi-Agent System.* published online at: http://www.agtivity.com/def/cmas.htm.

Lamieri, Marco, & Mangalagiu, Diana. 2009. Interactions Between Formal and Informal Organizational Networks, Formation of routines and Performance in Hierarchical Structures. *In:* Dignum, Virginia (ed), *Handbook of Research* *on Multi-Agent Systems: Semantics and Dynamics of Organizational Models*. Information Science Reference. IGI Global.

Lant, Theresa K. 1994. Computer simulations of organizations as experiential learning systems: implications for organization theory. *Computational organization theory*, 195–215.

Lawrence, P. R., & Lorsch, J. W. 1967. *Organisation and Environment: Managing Differentiation and Integration*. Division of Research, Graduate School of Business Administration, Harvard University.

Lee, L. C., Nwana, H. S., Ndumu, D. T., & Wilde, P. De. 1998. The Stability, Scalability and Performance of Multi-agent Systems. *BT Technology Journal*, **16**(3), 94–103.

Lesser, V., Decker, K., Wagner, T., Carver, N., Garvey, A., Horling, B., Neiman, D., Podorozhny, R., NagendraPrasad, M., Raja, A., Vincent, R., Xuan, P., & Zhang, X.Q. 2002. Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework. *Proceedings 1st International Conference on Autonomous Agents and Multi-Agent Systems (Plenary Lecture/Extended Abstract)*, 1–2.

Lesser, Victor. 1999. Cooperative Multiagent Systems: A Personal View of the State of the Art. *IEEE Transactions on Knowledge and Data Engineering*, **11**(1).

Lesser, Victor, Horling, Bryan, Klassner, Frank, Raja, Anita, Wagner, Thomas, & Zhang, Shelley XQ. 2000. BIG: An agent for resource-bounded information gathering and decision making. *Artificial Intelligence*, **118**(1-2), 197–244.

Lesser, Victor R., Decker, Keith, Wagner, Thomas, Carver, Norman, Garvey, Alan, Horling, Bryan, Neiman, Daniel E., Podorozhny, Rodion M., Prasad, M. V. Nagendra, Raja, Anita, Vincent, Régis, Xuan, Ping, & Zhang, Xiaoqin. 2004. Evolution of the GPGP/TÆMS Domain-Independent Coordination Framework. *Autonomous Agents and Multi-Agent Systems*, **9**(1-2), 87–143.

Maheswaran, Rajiv, Szekely, Pedro, Becker, Marcel, Fitzpatrick, Stephen, Gati, Gergely, Jin, Jing, Neches, Robert, Noori, Nader, Rogers, Craig, Sanchez, Romeo, Smyth, Kevin, & Buskirk, Chris Van. 2008. Predictability and Criticality Metrics for Coordination in Complex Environments. *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, May, 647–654.

March, James G., & Simon, Herbert A. 1993. *Organizations.* 2nd edition edn. Blackwell Publishers.

Marin, O., Sens, P., Briot, J., & Guessoum, Z. 2001. Towards Adaptive Fault Tolerance for Distributed Multi-Agent Systems. *Proceedings of European Research Seminar on Advances in Distributed Systems (ERSADS 2001)*, May.

Martin, Cheryl, & Barber, K. Suzanne. 2006. Adaptive decision-making frameworks for dynamic multi-agent organizational change. *Autonomous Agents and Multi-Agent Systems*, **13**(3), 391–428.

McNamara, P., & Prakken, Henry (eds). 1999. *Norms, Logics and Information Systems.* IOS Press.

Moore, Dana, & Wright, William. 2003. Emergent behaviours considered harmful. *Pages 1070–1071 of: AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*. New York, NY, USA: ACM Press.

Nair, Ranjit, Tambe, Milind, & Marsella, Stacy. 2003. Role Allocation and Reallocation in Multiagent Teams: Towards A Practical Analysis. *Pages 552–559 of: Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS-03)*.

Parunak, H. Van Dyke. 1997. Go to the ant: Engineering principles from natural multi-agent systems. *Annals of Operations Research*, **75**(0), 69–101.

Pynadath, David V., & Tambe, Milind. 2002. The Communicative Multiagent Team Decision Problem: Analyzing Teamwork Theories and Models. *Journal of Artificial Intelligence Research*, **16**, 389–423.

Rich, Elaine, & Knight, Kevin. 1991. *Artificial Intelligence*. 2nd edn. McGraw Hill.

Robbins, S. 1989. *Organization Theory — Structure Design and Applications*. 3rd edn. Prentice-Hall, Englewood Cliffs.

Rosenschein, Jeffrey S, & Zlotkin, Gilad. 1994. *Rules of Encounter: Designing conventions for automated negotiation among computers*. MIT Press.

Russell, Stuart J., & Norvig, Peter. 2002. *Artificial Intelligence: A Modern Approach*. 2nd edn. Prentice Hall.

Scott, W. Richard. 1998. *Organizations: Rational, Natural, and Open Systems*. Prentice Hall.

Serugendo, Giovanna Di Marzo, Gleizes, Marie-Pierre, & Karageorgos, Anthony. 2006a. Self-Organisation and Emergence in MAS: An Overview. *Informatica*, **30**(1), 45–54.

Serugendo, Giovanna Di Marzo, Gleizes, Marie-Pierre, & Karageorgos, Anthony. 2006b. Self-organization in multi-agent systems. *The Knowledge Engineering Review*, **20**(02), 165–189.

Shehory, Onn, & Kraus, Sarit. 1998. Methods for task allocation via agent coalition formation. *Artificial Intelligence*, **101**(1-2), 165–200.

Shehory, Onn, Sycara, Katia, Chalasani, Prasad, & Jha, Somesh. 1998. Agent cloning: an approach to agent mobility and resource allocation. *IEEE Communications Magazine*, **36**(7), 58–67.

Shirts, Michael, & Pande, Vijay S. 2000. COMPUTING: Screen Savers of the World Unite! *Science*, **290**(5498), 1903–1904.

Shoham, Yoav, & Tennenholtz, Moshe. 1995. On social laws for artificial agent societies: off-line design. *Artif. Intell.*, **73**(1-2), 231–252.

Simon, Herbert. 1957. A Behavioral Model of Rational Choice. *In: Models of Man, Social and Rational: Mathematical Essays on Rational Human Behavior in a Social Setting*. New York: John Wiley and Sons.

Simon, Herbert A. 1991. Bounded Rationality and Organizational Learning. *Organizational Science*, **2**(1), 125–134.

Sims, Mark, Mostafa, Hala, Horling, Bryan, Zhang, Haizheng, Lesser, Victor, & Corkill, Dan. 2006. Lateral and Hierarchical Partial Centralization for Distributed Coordination and Scheduling of Complex Hierarchical Task

Networks. *AAAI 2006 Spring Symposium on Distributed Plan and Schedule Management.*

Singh, Munindar P., & Huhns, Michael N. 2005. *Service-Oriented Computing: Semantics, Processes, Agents.* John Wiley and Sons.

Smith, R. G., & Davis, R. 1978. Distributed Problem Solving: The Contract Net Approach. *In: Proceedings of the 2nd National Conference of the Canadian Society for Computational Studies of Intelligence.*

Smith, Reid G. 1980. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers*, **C-29**(12), 1104–1113.

Smith, Reid. G. 1988. The contract net protocol: High-level communication and control in a distributed problem solver. *Pages 357–366 of: Distributed Artificial Intelligence.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

So, Y., & Durfee, E. 1993 (July). An organizational self-design model for organizational change. *Pages 8–15 of: AAAI-93 Workshop on AI and Theories of Groups and Organizations: Conceptual and Empirical Research.*

So, YoungPa, & Durfee, Edmund H. 1996. Designing Tree-Structured Organizations for Computational Agents. *Computational and Mathematical Organization Theory*, **2**(3), 219–245.

Sonnek, J., Nathan, M., Chandra, A., & Weissman, J. 2006. Reputation-Based Scheduling on Unreliable Distributed Infrastructures. *Proceedings of the 26th International Conference on Distributed Computing Systems*, July.

Sutton, Richard S., & Barto, Andrew G. 1998. *Reinforcement Learning: An Introduction.* 1 edn. Cambridge, MA: MIT Press,.

Tambe, Milind. 1997. Towards Flexible Teamwork. *Journal of Artificial Intelligence Research*, **7**, 83–124.

Tambe, Milind, Adibi, Jafar, Alonaizon, Y., Erdem, Ali, Kaminka, Gal A., Marsella, Stacy, & Muslea, Ion. 1999. Building Agent Teams Using an Explicit Teamwork Model and Learning. *Artificial Intelligence*, **110**(2), 215–239.

Taufer, Michela, Teller, Patricia J., Anderson, David P., & III, Charles L. Brooks. 2005a (December). Metrics for Effective Resource Management in Global Computing Environments. *Pages 204–211 of: First International Conference on e-Science and Grid Technologies (e-Science 2005).*

Taufer, Michela, Crowley, Michael, Price, Daniel J., Chien, Andrew A., & III, Charles L. Brooks. 2005b. Study of a highly accurate and fast protein-ligand docking method based on molecular dynamics. *Concurrency and Computation: Practice and Experience*, **17**(14), 1627–1641.

van Elst, Ludger, Dignum, Virginia, & Abecker, Andreas. 2003. *Towards Agent-Mediated Knowledge Management.* Lecture Notes in Computer Science, vol. 2926. Springer Berlin / Heidelberg.

Vazquez-Salceda, Javier, Dignum, Virginia, & Dignum, Frank. 2005. Organizing Multiagent Systems. *Autonomous Agents and Multi-Agent Systems*, **11**(3), 307 – 360.

Vera, Julio, Curto, Raul, Cascante, Marta, & Torres, Nestor V. 2007. Detection of potential enzyme targets by metabolic modelling and optimization: application to a simple enzymopathy. *Bioinformatics*, **23**(17), 2281–2289.

Vriend, Nicolaas J. 2006. ACE Models of Endogenous Interactions. *Handbook of Computational Economics*, **2**(21), 1047–1079.

Wagner, Thomas, & Lesser, Victor. 2000. Design-to-Criteria Scheduling: Real-Time Agent Control. *Proceedings of AAAI 2000 Spring Symposium on Real-Time Autonomous Systems*, March, 89–96.

Wagner, Tom. 2004. *Coordination Decision Support Assistants (COORDINATORs).* Tech. rept. 04-29. DARPA Technical Report - BAA.

Wellman, Michael P. 1997. Market-Aware Agents for a Multiagent World. *Page 1 of: Proceedings of the 8th European Workshop on Modelling Autonomous Agents in a Multi-Agent World.* London, UK: Springer-Verlag.

Weyns, Danny, Parunak, H. Van Dyke, Michel, Fabien, Holvoet, Tom, & Ferber, Jacques. 2004 (July). Environments for Multiagent Systems — State-of-the-Art and Research Challenges. *In: The First International Workshop on Environments for Multiagent Systems.*

Wooldridge, Michael, & Jennings, Nicholas R. 1998. Pitfalls of agent-oriented development. *Pages 385–391 of: AGENTS '98: Proceedings of the second international conference on Autonomous agents.* New York, NY, USA: ACM Press.

Zambonelli, Franco, Jennings, Nicholas R., & Wooldridge, Michael. 2003. Developing Multiagent Systems: The Gaia Methodology. *ACM Transactions on Software Engineering and Methodology*, **12**(3), 317–370.

Zhang, Yingqian, Manisterski, Efrat, Kraus, Sarit, Subrahmanian, V.S., & Peleg, David. 2009. Computing the fault tolerance of multi-agent deployment. *Artificial Intelligence*, **173**(3-4), 437 – 465.

Zimmerman, Terry Lyle, Smith, Stephen, Gallagher, Anthony T, Barbulescu, Laura, & Rubinstein, Zack. 2007 (May). Distributed Management of Flexible Times Schedules. *In: Sixth International conference on Autonomous Agents and Multiagent Systems (AAMAS).*