Analyzing the tradeoffs between breakup and cloning in the context of organizational self-design

Sachin Kamboj [*]
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716
kamboj@cis.udel.edu

ABSTRACT

Organizational Self-Design (OSD) has been proposed as an approach to constructing suitable organizations at runtime in which the agents are responsible for constructing their own organizational structures. OSD has also been shown to be especially suited for environments that are dynamic and semi-dynamic.

Most existing OSD approaches work by changing the organizational structure in response to changes in the environment — usually by spawning a new agent when an agent is overloaded and composing agents when they are free. One approach to spawning involves "breaking" up a problem into smaller sub-problems and assigning one of the sub-problems to the newly spawned agent. An alternative approach works by "cloning" the source agent and assigning the clone agent a portion of the source's work load. We posit that both of these approaches are complementary, have their own advantages, and can be used together. In this paper we analyze the tradeoffs between cloning and breakup and generate a hybrid model that uses both cloning and breakup to generate more suitable organizations than those that could be generated when using a single approach.

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Multiagent systems

General Terms

Algorithms, Design, Performance, Experimentation

Keywords

Organizational-Self Design, Task and Resource Allocation

1. INTRODUCTION

The organization of a multiagent system is concerned with issues such as the number of agents needed to solve a problem, the allocation of tasks and resources amongst the agents and the coordination of inter-agent activities. These issues can be addressed by selecting an organizational structure, consisting of roles and relationships, and then instantiating agents for that structure.

Contingency theory tells us 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 being solved and the environmental conditions (problem arrival rate, deadlines, etc) under which the problem is being solved.

If the environment is dynamic or semi-dynamic, it precludes the use of a static, design-time generated organizational-structure. Organizational self-design (OSD), in which the agents are responsible for designing their own organizational structures at run-time, has been proposed as a mechanism for designing organizations for such environments. OSD is particularly suited to the problem of generating virtual organizations for grid/volunteer/cloud computing environments.

In our model, problem solving requests (tasks) arrive at the multiagent system at indeterminate times. The multiagent system is responsible for finding solutions to these requests by their deadlines. The multiagent system starts off with a *single* agent responsible for solving the problem in its entirety. As in [8], if the 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*.

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*.

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.

We use TÆMS as the underlying representation for our problem solving requests. TÆMS [9] (Task Analysis, Environment Modeling and Simulation) is a computational framework for representing and reasoning about complex task environments in which tasks

[*]Author is a student

Cite as: Analyzing the Tradeoffs Between Breakup and Cloning in the Context of Organizational Self-Design, Sachin Kamboj, *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, Decker, Sichman, Sierra and Castelfranchi (eds.), May, 10–15, 2009, Budapest, Hungary, pp. 829–836

(problems) are represented using extended hierarchical task structures. The root node of the task structure represents the high-level goal that the agent is trying to achieve. The sub-nodes 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 agents can perform. The executable methods, themselves, may have multiple outcomes, with different probabilities and different characteristics such as quality, cost and duration. TÆMS also allows various mechanisms for specifying subtask variations and alternatives, i.e. each node in TÆMS is labeled with a characteristic accumulation function that describes how many or which subgoals or sets of subgoals need to be achieved in order to achieve a particular higher-level goal. TÆMS has been used to model many different problem-solving environments including distributed sensor networks, information gathering, hospital scheduling, EMS, and military planning [1, 10].

The main contributions of this paper are as follows:

1. We perform an analysis of the tradeoffs between breakup and cloning in worth-oriented domains.
2. We describe five different strategies for selecting between breakup and cloning when spawning an agent. One of these strategies involves developing a hybrid model that tries to combine the benefits of both of these approaches.
3. We introduce an approach to OSD that uses task structure rewriting to represent and reason over organizational structures.

The structure of the rest of this paper is as follows: In the next section we describe some related work. Then, in Sections 3 and 4, we give a formal problem definition and describe our approach to OSD. We then describe our analytic model in Section 4.3 and present some experimental results in Section 5. Finally, we conclude in Section 6.

2. RELATED WORK

The tradeoffs between breakup and cloning have been studied extensively in the management sciences (see, for example, [11]), where this tradeoff is presented in terms of the difference between specialization and generalization. However, almost all of these studies have focused on human organizations and cannot directly be applied to virtual organizations in grid/cloud/volunteer computing environments. [5] and other researchers introduced these concepts to the distributed AI literature, though they did not offer any analysis of the tradeoff.

This paper primarily depends on and attempts to unify two distinct lines of research:

1. The research on Cloning performed by Shehory, Sycara et. al. [13, 2]. This research was responsible for introducing the concept of cloning as a mechanism for load balancing — 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.
2. The work on OSD by [6]which introduced decomposition (spawning) and composition as a way of performing adaptive work allocation and load balancing. [8] extended their work to worth oriented domains.

These two approaches are different in that there is no specialization of the agents in the former – the cloned agents are perfect replicas of the original agents and fulfill the same roles and responsibilities as the original agents. The latter approach, on the other hand *always* specializes and does not consider the situations in which simple cloning might be better. In this paper we attempt to study this tradeoff.

There are numerous other approaches to both task allocation ([4, 14] and others), in general, and OSD ([15, 7, 12, 16, 3]), in particular. The market-based task allocation mechanisms [4, 14] implicitly result in agent specialization in most cases, even though the tradeoffs are not explicitly reasoned over. Both [15, 16] are concerned with hierarchical organizations which, by definition, have at least some degree of specialization between the layers of the hierarchy. However, the siblings may either fulfill the same roles or fulfill different roles depending on the organization of the hierarchy. These decisions are implicit in the design of the hierarchies and are not explicitly reasoned over in these papers.

Finally, some OSD papers are more concerned with changing (a) the interaction patterns between the agents [7]; and (b) changing the autonomy and decision making authority of the agents [12]. These areas of research are only orthogonally related to ours.

3. TASK AND RESOURCE MODEL

Our task and resource model is based on the one presented in [8, 1] and is summarized here for formal grounding. The primary input to the multi-agent system is an ordered set of problem solving requests or task instances, $< P_1, P_2, P_3, ..., P_n >$, where each problem solving request, $P_i =< \mathbf{t_i}, \mathbf{a_i}, \mathbf{d_i} >$. Here, $\mathbf{t_i}$ 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.

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

- $T = \{t_1, t_2, ..., t_n\}$ is the set of tasks, the non-leaf nodes of a TÆMS task structure. Tasks denote goals that the agents must achieve and are represented using the pair $t_i = (q_i, s_i)$, where $q_i \in Q$ is the quality/characteristic accumulation function (QAF) (see below); and $s_i \subset (T \cup M)$ is a set of subtasks and or methods of T.
- $\tau \in T$, is the root of the task structure, that is, the highest level goal that the organization is trying to achieve.
- $M = \{m_1, m_2, ..., m_n\}$, is the set executable methods, i.e. the primitive methods that can be directly executed by an agent. Each method, m_k, is represented using the outcome distribution, $\{(o_1, p_1), (o_2, p_2), ..., (o_m, p_m)\}$; where o_l is an outcome and p_l is the probability that executing m_k will result in the outcome o_l. Furthermore, $o_l = (q_l, c_l, d_l)$, 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 $\sum_{i=1}^{m} p_l = 1$.
- $Q = \{$ MIN, MAX, SUM, EXACTLY_ONE, ...$\}$ is the set of quality/characteristic accumulation functions (CAFs). The quality accumulation function determines how the quality of a task is computed from the quality of its subtasks. See [1] for formal definitions.
- E is the set of (non-local) effects, i.e. $E = \{e_1, e_2, ..., e_n\}$, where each effect, $e_i =< \alpha, \beta, \delta >$. Here, $\alpha \in (T \cup M)$, is

the source of the non-local effect, $\beta \in M$, is the sink method of the non-local effect and δ, is a temporal function that computes the characteristics of the sink given the characteristics of the source. Again, see [1] for formal definitions.

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

4. ORGANIZATIONAL SELF DESIGN

Most approaches to OSD, including ours, consist of two primary components: (1) An evaluation component that is responsible for monitoring the performance of the organization according to some utility function; and (2) an adaptation component that is triggered by the evaluation component when the utility falls below a threshold. The adaptation component is responsible for modifying some aspect of the organizational structure so as to increase the performance of the organization. These two components form a feedback system and result in the dynamics of the organization.

In our approach to OSD, both these components are present in *each* agent and any agent can change the organization at run time (hence, the term *self-design*). In the next section, we will describe the evaluation component and in the following section we will discuss the adaptation component.

4.1 Detecting the need for organizational change

The primary trigger for change in our system is a change in the *task load* of an agent, usually as a result of some change in the environment. For example, the task arrival rate may change, the deadlines on the arriving tasks might change or the available resources might change.

If an agent cannot perform the tasks in its task queue by their deadlines, the agent is said to be *overloaded.* The agent responds to this situation by *spawning* off a new agent to handle part of its task load. If, on the other hand, the agent is idle for an extended period of time, it can find another idle agent and try to *compose* with it.

4.2 Organizational Structure

In our approach, organizations are represented using an organizational structure that 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 any 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.

Also note that 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. 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 — the combined agent is responsible for enacting all the roles of the agents being merged.

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 structures, 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.

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 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: $\Sigma \subseteq T_\mathcal{O}$[1], 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 $t_i \in \{ROOT, CLONE, COORDINATION\}$ 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[2]. 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: $\mathcal{S_C} \subset M_\mathcal{O}$ 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 the agents owning those clones.
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 4.2 for the rationale behind these node.

To allow for a change in an agent's organizational knowledge, we define three rewriting operators on a local task structure, which are

[1]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.

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

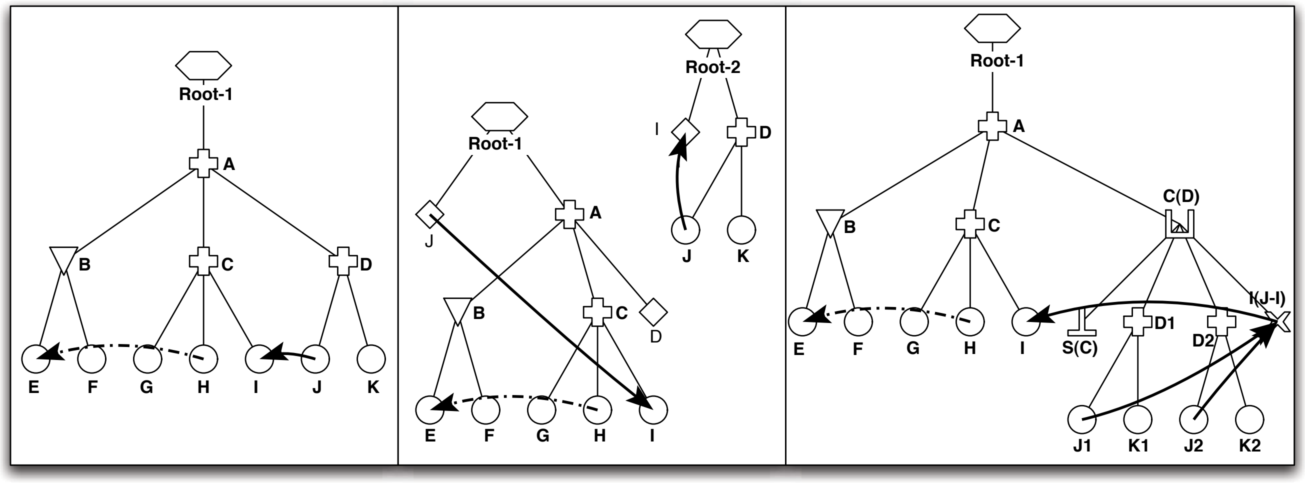

Figure 1: Figure demonstrating the use of task rewriting to breakup and clone a node. The leftmost figure shows a TÆMS task structure. In the middle figure, *Root 1* has been broken at Node D. Note the diamonds J and D in *Root 1*, which represent non-local nodes in *Root 1* corresponding to local nodes in *Root 2*. Similarly, diamond I represents a non-local node in *Root 2*. In the rightmost figure, Node C has been cloned once. Note the clone container node, $C(D)$, the selector node $S(C)$ and the inherited-method, $I(J-I)$.

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 $\sigma_1 =< ROOT, \{\mathbf{t}\} >$.

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 over 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.

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: $x \Leftarrow$ GETNONLOCALNODE($\upsilon$)
11: REPLACENODE($\tau, \upsilon, x$)
12: return CREATEROOTNODE($\upsilon$)
```

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 1 and the algorithm for the breakup operator is shown in Algorithm 1.

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

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

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

The algorithm for the merge operator is shown in Algorithm 2. In order to fulfill these 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 3–7 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 10–13 of the algorithm. Finally, any two non local nodes that have the same identity should be merged into a single non-local node (lines 8–9) or formally $\exists \diamond_1, \exists \diamond_2 \mid (\diamond_1 = < \eta_1 > \wedge \diamond_2 = < \eta_2 > \wedge \eta_1 = \eta_2) \Rightarrow \diamond_1 = \diamond_2$

Cloning: To allow for situations in which (a) breakup might be infeasible and/or (b) the agent would prefer to do simple load balancing instead of breaking up the task structure, we introduce a *cloning operator* that is responsible for making two copies, $< c_1, c_2 >$ of a substructure, $\upsilon \in \mathcal{D}$[3] so that the root task, τ can be broken up at node, υ, 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 the rightmost image in Figure 1 and the algorithm is described in Algorithm 3. To clone a node, υ in a root task, τ, we first create a new container node, $\sigma_c = < CLONE, \{\upsilon\} >$, called a *clone container* and replace υ in τ with σ_c. The clone container will be used to "hold" all the created clones.

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 need to create a clone selector $s_c \in S_c$ method and add *enables* NLEs from s_c to both c_1 and c_2. Then method s_c has to be executed before any of the clones can be run and it can selectively enable one or more of the clones.

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, $\{e \in E \mid [$SOURCE(e) $\in$DESCENDANTS(c_1) $\wedge$ SINK(e) $\in$ (DESCENDANTS(σ) $-$ DESCENDANTS(c_1))] $\vee$ [SINK(e) $\in$DESCENDANTS(c_1) $\wedge$ SOURCE(e) $\in$ (DESCENDANTS(σ) $-$ 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.

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

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: for all { $x \mid x \in \overline{\upsilon}$ } do
5:   $y \Leftarrow$ COPYNODE($x$)
6:   ADDNODE($\phi, y$)
7: for all { $N \mid N \in$ NLEs($\upsilon$) } do
8:   if SOURCE($N$) $\in \overline{\tau}$ then
9:     $x \Leftarrow$ CREATEINHERITINGNODE()
10:    ADDNODE($\phi, x$)
11:    $L \Leftarrow$ COPYNLE($N$)
12:    $M \Leftarrow$ COPYNLE($N$)
13:    REPLACENODE($N$, SINK($N$), $x$)
14:    REPLACENODE($L$, SOURCE($L$), $x$)
15:    REPLACENODE($M$, SOURCE($M$), $x$)
16:    $y \Leftarrow$ FINDNODE($\phi$, SINK($M$))
17:    REPLACENODE($M$, SINK($M$), $y$)
18:  else if SINK($N$) $\in \overline{\tau}$ then
19:    {Similar to the source}
20: return $\phi$
```

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. However, the details are outside the scope of this paper.

These operators result in the rewriting of a local task structure. In the case of agent spawning, the spawning agent, A, selects a node, $\upsilon \in \mathcal{D}$ for breakup, runs the 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.

4.3 Selecting a Spawning Strategy

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 as described in Section 4.2. We used the *Balancing Execution Time* (BET) heuristic to select the node to be allocated to the new agent, as defined in [8].

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 $feasible(Breakup(\sigma)) \Leftrightarrow |\{x \mid x \in \textsc{Descendents}(\sigma) \wedge x \in M\}| > 1$). 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[4]. 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 ψ. 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: $U_{Breakup}(\vartheta) = U_{Breakup}(\vartheta)(1 - (\xi * var))$, 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. 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.
   - 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.
   - 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.
   - 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.

[4]Not reported here because of space constraints.

5. EVALUATION

We ran a series of experiments to compare our five agent spawning strategies. At the start of each experimental run, a random TÆMS task structure with a maximum depth of 4 and maximum branching factor of 4 was generated. The five strategies were then run in parallel, on the same task structure while maintaining the same exact environmental conditions (i.e. task arrival times, task deadlines, random numbers, etc...) for all of them. Each experiment was repeated 15 times with a new randomly generated task structure for each repetition. The 15 repetitions of an experiment form an *experimental set*.

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.) 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:

- 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 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.[5].

- 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*.

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, $a_{sp\text{-}diff}$, which is used to control the task arrival rate (defined as the rate at which a new task instance is generated). The arrival rate was set to $a_{sp\text{-}diff} *$ *sp-diff*. We set $a_{sp\text{-}diff}$ to the following values: 0.01, 0.1, 0.5 and 1.0.

[5]The exact algorithm used to compute these times is beyond the scope of this paper

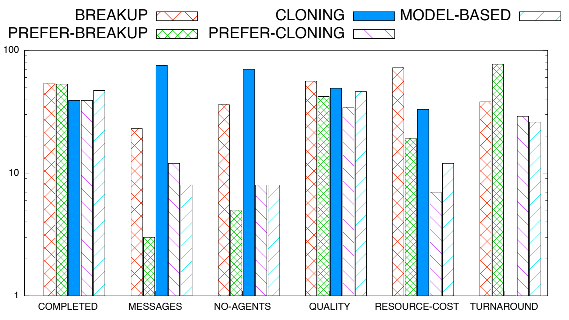

Figure 2: Graph showing the number of times each strategy performed the best or was in a group that performed statistically equivalent to the best. The y-axis uses a logarithmic scale.

2. The deadline *sp-diff* multiple, $d_{sp\text{-}diff}$, used to set the deadline window, defined as the difference between the task deadline and the arrival time. The deadline window for a task was set to $PET + d_{sp\text{-}diff} * sp\text{-}diff$. We used $d_{sp\text{-}diff}$ 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[6].

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.

2. The percentage of tasks completed, defined as the number of tasks completed divided by the number of tasks generated.

3. The total number of messages sent by the agents in the organization.

4. The average quality accrued by the organization.

5. The total resource cost of the organization.

6. The average turnaround time of the tasks in the organization. The turnaround time is defined as the time at which a task was completed or failed minus the time at which the task was generated.

Except for the percentage of tasks completed and the quality accrued, lower numbers of the above criteria indicate better performance. We ran a total of 96 experimental sets or 1440 experiments. To test for statistically significant differences between the performance of strategies, we ran the *Wilcoxon Matched-Pair Signed-Rank* tests. *Matched-Pair* signifies that we are comparing the performance of each system on precisely the same randomized task set within each separate experimental set.

The results of these significance tests are shown in Figure 2. The vertical bars show the number of times (out of 96, for the 96 experimental sets) that each strategy either performed the best or was statistically equivalent to a strategy that performed the best.

[6]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

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 3. 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. 2 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. 3, 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. 3, 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[7]. As the $d_{sp\text{-}diff}$ 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. 2, our *Hybrid Model* performs statistically equivalent to the

[7]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.

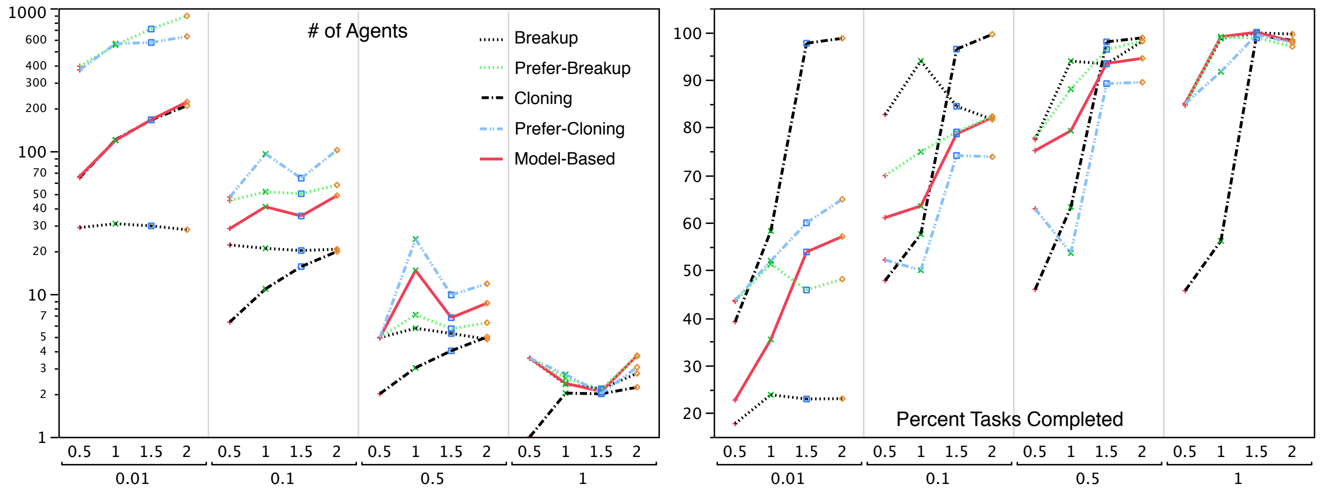

Figure 3: Graph showing the *Number of agents* (lower is better) and *Percent Tasks Completed* (higher is better). The x-axis shows the $d_{sp\text{-}diff}$ values (the top numbers) within the outer $a_{sp\text{-}diff}$, values (the bottom numbers).

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 ($a_{sp\text{-}diff} \geq 0.5$)and midway between *Breakup* and *Cloning* for situations where $a_{sp\text{-}diff} \leq 0.1$ and $d_{sp\text{-}diff} \leq 0.5$. In particular, for $a_{sp\text{-}diff} = 0.01$ and $d_{sp\text{-}diff} \leq 1$, 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.

6. CONCLUSION

In this paper, we have analyzed the tradeoffs between breakup and cloning and have compared five strategies for picking between the two approaches while spawning an agent. We have also developed a hybrid model that tries to incorporate the best of both the two approaches.

In our future work, we would like to come up with an automated way of learning the constants and parameters used in our hybrid model.

7. REFERENCES

[1] K. S. Decker. Environment centered analysis and design of coordination mechanisms. *Ph.D. Thesis, Department of Computer Science, University of Massachusetts, Amherst*, May 1995.

[2] K. S. Decker, K. P. Sycara, and M. Williamson. Cloning for intelligent adaptive information agents. In *Revised Papers from the Second Australian Workshop on Distributed Artificial Intelligence*, number 3-540-63412-6, pages 63–75, London, UK, 1997. Springer-Verlag.

[3] S. DeLoach, W. Oyenan, and E. Matson. A capabilities-based model for adaptive organizations. *Autonomous Agents and Multi-Agent Systems*, 16(1):13–56, 2008.

[4] S. S. Fatima and M. Wooldridge. Adaptive task resources allocation in multi-agent systems. In *AGENTS '01*, pages 537–544, New York, NY, USA, 2001. ACM Press.

[5] M. S. Fox. An organizational view of distributed systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(1):70–80, January 1981.

[6] L. Gasser and T. Ishida. A dynamic organizational architecture for adaptive problem solving. In *AAAI '91*, pages 185–190, Menlo Park, California, 1991. AAAI Press.

[7] B. Horling, B. Benyo, and V. Lesser. Using self-diagnosis to adapt organizational structures. In *AGENTS '01*, pages 529–536, New York, NY, USA, 2001. ACM Press.

[8] S. Kamboj and K. S. Decker. Organizational self-design in semi-dynamic environments. In *AAMAS '07*, pages 1220–1227, May 2007.

[9] V. R. Lesser, et. al. Evolution of the GPGP/TÆMS Domain-Independent Coordination Framework. *Autonomous Agents and Multi-Agent Systems*, 9(1-2):87–143, 2004.

[10] R. Maheswaran, et. al. Predictability and criticality metrics for coordination in complex environments. In *AAMAS '08*, pages 647–654, May 2008.

[11] J. G. March and H. A. Simon. *Organizations*. Blackwell Publishers, 2nd edition edition, 1993.

[12] C. Martin and K. S. Barber. Adaptive decision-making frameworks for dynamic multi-agent organizational change. *Autonomous Agents and Multi-Agent Systems*, 13(3):391–428, 2006.

[13] O. Shehory, K. Sycara, et. al. Agent cloning: an approach to agent mobility and resource allocation. *IEEE Communications Magazine*, 36(7):58–67, 1998.

[14] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. In *Distributed Artificial Intelligence*, pages 357–366, San Francisco, CA, USA, 1988. Morgan Kaufmann.

[15] Y. So and E. H. Durfee. Designing tree-structured organizations for computational agents. *Computational and Mathematical Organization Theory*, 2(3):219–245, September 1996.

[16] H. Zhang and V. Lesser. A dynamically formed hierarchical agent organization for a distributed content sharing system. In *IAT '04*, pages 169–175, Beijing, September 2004.