Using Design Metrics for Predicting System Flexibility*

Robby, Scott A. DeLoach, and Valeriy A. Kolesnikov

Department of Computing and Information Sciences, Kansas State University,
234 Nichols Hall, Manhattan, Kansas, USA
{robby, sdeloach, valkov}@ksu.edu

Abstract. While multiagent systems have been extolled as dynamically configurable and capable of emergent behavior, these qualities can be a drawback. When the system changes so that it no longer achieves its goals, emergent behavior is undesirable. Giving agents the autonomy to adapt and then expecting them to adapt only in acceptable ways requires rigorous design analyses. In this paper, we propose metrics for determining system flexibility at design time. Our approach is based on organization-based multiagent systems, which allows multiagent systems to adapt within a preset structure. We tailored the Bogor model checker to efficiently analyze the adaptive behaviors of these systems and to determine their properties such as fault-tolerance and cost-efficiency. We develop state-space coverage metrics to allow designers to make informed trade-offs at design-time between computational cost and system flexibility.

1 Introduction

Distributed systems that can adapt to dynamically changing environments are becoming prevalent. The advent of the Internet and wireless communications has allowed users to expect the ability to integrate their local applications with data and computational capabilities from any location, at any time. Applications for distributed, adaptive systems include information systems, communication systems, sensor networks, and cooperative robotic teams. The prevailing approach to building these distributed, adaptive systems is that of multiagent systems in which locally autonomous agents coordinate with each other to provide access to distributed information and services. The power in the multiagent approach is that, because of autonomy, the agents can adapt to their environment and thus satisfy their assigned goals.

While multiagent systems have been widely touted as dynamically configurable and capable of emergent behavior, this has also been noted as a significant drawback. Most designers/users are not comfortable with the idea of pure emergent behavior where agents learn or discover and continually modify their behavior. As long as the behavior being learned or discovered is consistent with system goals, emergent behavior is not a problem. However, when the system functionality changes to where it no longer accomplishes its stated goals, emergent behavior becomes undesirable.

* This material is based upon work supported by the National Science Foundation under Grant No. 0347545 and by the Air Force Office of Scientific Research.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 184 – 198, 2006.

A key problem faced by the Agent-Oriented Software Engineering (AOSE) community is ensuring that multiagent systems will actually perform as desired without undesirable emergent behavior, which results from individual agent autonomy. Giving agents the autonomy to adapt and then expecting them to adapt only in acceptable ways requires rigorous analyses when designing and building these systems. In this paper, we propose some new design metrics and investigate one in depth for determining multiagent system flexibility at the design level. Our approach is based on previous work on organization-based multiagent systems [7] and model checking [6]. We describe how a software model checking framework such as Bogor [15] can be customized to efficiently analyze emergent behaviors of multiagent systems.

The novelties and the main contributions of our work are: (1) efficient state-space exploration of multiagent system behaviors at the design level, (2) mining the constructed state-spaces to determine their desirable/undesirable properties such as fault-tolerance and cost-efficiency, (3) proposing several useful design metrics based on state-space coverage measures to capture these properties, and (4) validating the predictions from the proposed metrics by using simulation methods. By using the proposed metrics, we believe system designers are better equipped to make informed trade-off between cost and effectiveness of multiagent systems, as well as preventing ineffective system designs.

The paper is organized as follows. Section 2 presents a motivating example used to illustrate our approach. Section 3 presents the multiagent organization design metamodel that we consider. Section 4 presents an efficient state-space exploration technique implemented using the Bogor framework. Section 5 presents some of our proposed metrics that we validate in Section 6 using simulation methods. Section 7 presents some related work. Finally, Section 8 concludes and presents some future work.

2 Motivating Example

Throughout this paper we use an example from cooperative robotics to demonstrate our model of organization-based multiagent systems and the application of our design metrics. A simplified cooperative robotics example is used (due to space constraint), however it is still interesting enough to illustrate the application of the organization metamodel and the effect of the loss of hardware capabilities to the system.

The example we use is the Cooperative Robotic Floor Cleaning Company (CRFCC). Essentially, we are designing a team of robots whose goal is to clean the floors of a building. At initialization, the team is given a map of the building including the type of flooring of each area. The floors may be tiled or carpeted and may be littered with large debris as well as small dirt particles that must be cleaned. Therefore, the CRFCC must be able to pick up any large objects and then vacuum or mop the floors, based on their type. The team should be able to clean the floors of the building even when faced with failures of individual robots or specific capabilities on those robots. This implies that the team must be able to (1) *assign* floor areas based on individual team member's capabilities (i.e. to mop, vacuum, sweep, etc.), (2) *recognize* when a robot is incapable of carrying out its responsibilities, and (3) *reorganize* the team to allow the team to achieve its goal in spite of individual failures.

3 Organizational Metamodel

To allow teams of agents (or robots) to adapt to their environment by determining their own organization at runtime, we developed a metamodel that describes the knowledge required to define and reason about an organization [7, 14]. Given this knowledge, we have shown that multiagent teams are able to organize (and reorganize) themselves in an attempt to adapt to dynamic environments.

Organizations are typically defined as a set of agents who play roles within a structure that defines the relationships between those roles [3]. In our organization metamodel shown in Fig. 1 (simplified due to space constraint; we refer the readers to [7, 14] for a more complete description), we include these basic concepts of goals (G), roles (R), and agents (A), plus agent capabilities (C) and a set of assignments (Φ).

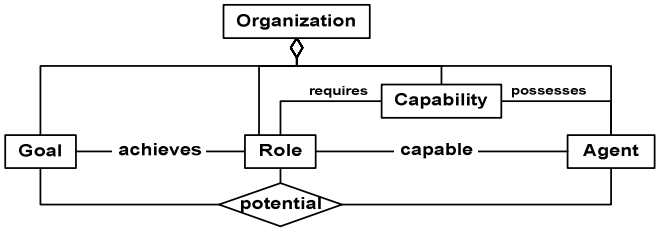

Fig. 1. Organization Metamodel (simplified)

3.1 Goals

Every organization is designed with a specific purpose or goal. In our metamodel, each organization has a set of goals, G, that it seeks to achieve in support of a top-level goal g_o, which we define as a desired end state. G is derived by decomposing g_o into a tree of sub-goals that describe how g_o can be achieved. Following the KAOS goal based requirements modeling approach [18], we allow goals to be decomposed into a set of non-cyclic sub-goals using either AND-refinement or OR-refinement. Eventually, g_o is refined into a set of leaf nodes, denoted by G_L, that are actually achieved by agents in order to achieve g_o. The active goal set, G_A (where $G_A \subseteq G_L$), is the set of goals that an organization is trying to achieve at the current time.

In order to provide an ordering for goal achievement, we define a precedence relation between goals. We say that goal g_1 *precedes* goal g_2 if g_1 must be achieved before g_2 can be achieved, which allows the team to work on a subset of the leaf goals, thus reducing the size of G_A. The initial active goal set, G_{A0}, consists of all leaf goals without predecessor goals. However, G_A changes as goals are achieved; achieved goals are removed from the active goal set and new goals are inserted. We denote a sequence of active goal sets G_A' as $G_A' = [G_{A1}, G_{A2}, \ldots, G_{An}]$.

The goal model for the CRFCC is shown in Fig. 2. Goals are denoted as specialized class components using the <<Goal>> notation. Conjunctive sub-goals are connected to their parents by a diamond shaped connector (◊) while disjunctive sub-goals are connected to their parent by a triangle shaped connector (Δ). Goals can have

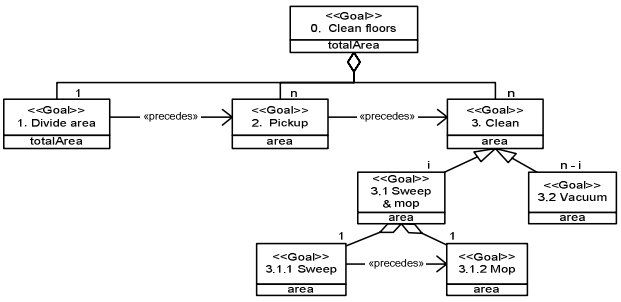

Fig. 2. Goal Model (simplified)

parameters. The *totalArea* parameter refers to the entire area to be cleaned. Since total area may include tile and carpeted areas, the team divides it into sub-areas (denoted by the *area* parameters) to be tackled independently. However, to ensure the entire task is completed as efficiently as possible, the team must consider the capabilities of its team members when partitioning the areas and assigning areas to robots. The multiplicity *n* represents the total number of sub-areas while *i* refers to the number of tiled areas. The <<precedes>> notation indicates precedence relation between goals.

The goal model consists of five leaf goals: Divide Area, Pickup, Sweep, Mop, and Vacuum. The precedence relations provide the natural ordering that is required to clean the floors. The *n* sub-areas must be created before work may begin; this results in *n* Pickup goal instances being created as well as *i* Sweep and Mop goals and *n-i* Vacuum goals. Due to the precedence relation, the individual areas must be picked up and any large debris removed before the areas can be swept, mopped, or vacuumed. Finally, depending on what type of flooring is present, the areas are either (1) swept and then mopped, or (2) vacuumed.

3.2 Capabilities

Capabilities are the key to determining exactly which agents can be assigned to what roles in the organization. Currently, we view a *capability* as an atomic entity used to define the abilities of agents. Capabilities can capture *soft* abilities such as the ability to access resources, communicate, migrate, or computational algorithms. They also capture *hard* capabilities such as those of hardware agents such as robots, which include sensors and effectors. In the CRFCC example, the robots must have specific capabilities to carry out the cleaning operation. Thus, we assume the capabilities shown in Table 1 are available for designing CRFCC robots.

The *org* capability is a reasoning ability that allows a robot to divide the current search area up into *n* areas based on the type of flooring (as well as other possible factors such as size, wall placement, etc.). The *search* capability allows robots to move about an area and identify items that need to be picked up before cleaning can begin. This capability is actually a combination of low-level capabilities such as movement and sensing as well as reasoning abilities to identify target items based on shape, size, color, etc. The *move* capability refers to the ability of a robot to pickup an item and to move it out of the way for cleaning. This capability could be representa-

Table 1. CRFCC Capabilities

Name	Description
org	Ability to logically divide the area between team members
search	Ability to search an area for large debris
move	Ability to move large debris
sweep	Ability to sweep a tiled area
mop	Ability to mop a tiled area
Vacuum	Ability to vacuum a carpeted area

Table 2. RM0 Roles

Name	Required Capabilities	Leaf Goals Achieved
Organizer	org	1. Divide Area
Pickuper	search, move	2. Pickup
TileCleaner	sweep, mop	3.1.1. Sweep & 3.1.2. Mop
Vacuummer	vacuum	3.2. Vacuum

tive of robotic arms or gripper devices. The last three capabilities, *sweep*, *mop* and *vacuum* are straightforward capabilities that also require integration of low-level capabilities. These capabilities provide the ability to clean tile and carpeted floors.

3.3 Roles

Each organization has a set of roles R that it can use to achieve its goals. A role defines the capabilities required for an entity to achieve a goal (or set of goals) in the organization. The *achieves* function ($R \times G_L \rightarrow [0,1]$) tells how good a role is for realizing a specific goal (0 = no ability to achieve the goal, 1 = excellent ability to achieve the goal); if agent A is *better* at attaining goal G than agent B, we would expect that *achieves(A,G) > achieves (B,G)*. However, to be assigned to play a role, agents must have a sufficient set of capabilities to play that role. Thus, agents *possess* capabilities while roles *require* a certain set of capabilities. The set of capabilities required by a role is captured using the *requires* relation ($R \times C$).

For the CRFCC example, we developed two sets of roles, or *role models*, that the individual robots can play in order to accomplish the overall CRFCC goal. In the first role model (RM0), we attempted to combine basic capabilities to carry out specific goals. For RM0 we came up with four roles as shown in Table 2. In this case, we would need a robot with the org capability to be assigned to the Organizer role in order to achieve the initial goal, Divide Area. Once the area was divided into sub-areas, the robots with the search and move capabilities would be assigned to play the Pickuper role to achieve all the Pickup goals generated for each sub-area. Once this goal was achieved, robots with sweep and mop capabilities would be assigned to the *TileCleaner* role to achieve goals Sweep and Mop for each tiled sub-area while robots with the vacuum capability would be assigned to play the *Vacuummer* role to achieve the Vacuum goal for each carpeted area.

In a second version of the role model, Role Model 1 (RM1) as shown in Table 3, we took a slightly different approach to defining the roles for the CRFCC. Instead of defining roles to carry out basic functions in the application, we defined a role for each leaf goal. Essentially, we divided the *TileCleaner* role into *Sweeper* and *Mopper*.

Table 3. RM1 Roles

Name	Required Capabilities	Leaf Goals Achieved
Organizer	org	1. Divide Area
Pickuper	search, move	2. Pickup
Sweeper	sweep	3.1.1. Sweep
Mopper	mop	3.1.2. Mop
Vacuummer	vacuum	3.2. Vacuum

3.4 Agents

The organization metamodel also includes a set of heterogeneous agents, *A*. For our purposes, *agents* are computational system instances that inhabit a complex dynamic environment, sense, and act autonomously in this environment, and by doing so realize a set of goals. Agents are assigned specific roles in order to achieve organizational goals. The current set of possible assignments of agents to a role is captured by the *potential* function ($G_L \times R \times A \rightarrow$ [0,1]). The range of the *potential* function indicates how well an agent can play a role and how well that role can achieve the goal, based on the *achieves* and the *capable* scores.

However, the *potential* function does not indicate the actual assignment of agent *a* to role *r* to achieve goal *g*, it simply defines possible assignments. To capture the actual assignments, we define an *assignment set* Φ, which consists of goal-role-agent tuples, <*g,r,a*>. If <*g,r,a*> ∈ Φ, then agent *a* has been assigned by the organization to play role *r* in order to achieve goal *g*. As discussed above, however, only agents with the right set of capabilities may be assigned to a role. To capture a given agent's capabilities, we define a *possesses* function ($A \times C \rightarrow$ [0,1]), whose dynamic value ranges from no (0) capability to an excellent (1) capability. Using a role's required capabilities and the capabilities possessed by an agent, we compute the ability of an agent to play a given role, which we capture in the *capable* function (A x $R \rightarrow$ [0,1]).

4 Using Bogor to Explore Behaviors of Multiagent Organization

Bogor [4, 15] is a model checking framework designed for extensibility to enable more effective incorporation of domain knowledge into verification models and model checking algorithms. In contrast to most existing model checkers, Bogor's modeling language (BIR) provides constructs commonly found in modern programming languages including dynamic object and thread creation, garbage collection, virtual method calls and exception handling. This rich modeling language has enabled us to model check relatively large concurrent Java programs. In addition, BIR can be extended with new primitive types, expressions, and commands associated with a particular domain (e.g., multi-agent systems, avionics, security protocols, etc.) and a particular level of abstraction (e.g., design metamodels, design models, source code, byte code, etc.) to enable efficient modeling and state-space representation. Furthermore, Bogor's well-organized module facility allows new algorithms (e.g., for state-space exploration, state storage, etc) and new optimizations (e.g., heuristic search strategies, domain-specific scheduling, etc.) to be easily swapped in to replace Bogor's default model checking algorithms. To support effective BIR software model

```
system OrganizationMetamodel {
  extension Set for SetModule {
    typedef type<'a>;
    expdef Set.type<'a> create<'a>('a ...);
    actiondef add<'a>(Set.type<'a>, 'a);
  } …
extension AOM for AOMModule {
    typedef Agent; typedef Goal; typedef Role;
    expdef boolean isTopGoalAchieved(Set.type<Goal> goals);
    expdef Goal chooseGoal(Set.type<Goal>);
    expdef Role chooseRole(Goal goal);
    expdef Agent chooseAgent(Role role);
  }
  active thread Search() {
    Goal g; Role r; Agent a;
    Set.type<Goal> achievedGoals;
    Set.type<Triple.type<Goal, Role, Agent>> assignments;
    achievedGoals := Set.create<Goal>();
    assignments := Set.create< Triple.type<Goal, Role, Agent>>();
    while (!AOM.isTopGoalAchieved(achievedGoals)) do
      g := AOM.chooseGoal(achievedGoals);
      r := AOM.chooseRole(g);
      a := AOM.chooseAgent(r);
      Set.add(achievedGoals, g);
      Set.add(assignments, Triple.create(g, r, a));
    end
  }
}
```

Fig. 3. Organization Metamodel and Search Algorithm in BIR (excerpts)

checking, we have extended well-known optimization/reduction strategies [8, 16] such as collapse compression [11], data [12] and thread [5] symmetry, partial-order reduction [6] strategies that leverage static/dynamic escape and locking analyses.

We leverage BIR's extensibility to represent the organization metamodel presented in the previous section, as shown in Fig. 3. Each entity in the metamodel (e.g., agents) is modeled as a (native) first-class type in BIR (e.g., Agent). Similarly, we define auxiliary structures such as tuple and set and their corresponding abstract operations to enable more concise model. Moreover, by modeling organization entities and data structures as first-class type in BIR, we can instruct Bogor to use customized state representations better suited to the analysis' level of abstraction. For example, we leverage symmetric property of set to efficiently store set instances in the state-space representation (e.g., {Agent1, Agent2} = {Agent2, Agent1}). Accordingly, first-class abstract operations are implemented as an extension of the model checker instead of being a part of the model itself, thus, they are interpreted in the model checker's space instead of the model's space. This is analogous to adding new native types and instructions in a processor. That is, we can use the new types and instructions to better represent and more efficiently execute programs instead of representing them using a limited set of types and instructions. ([15] describes how to implement Bogor extensions.) The extension module AOM requires an organization instance as a Bogor configuration that contains information such as the goal structure, functions, and relations described in the previous section for that particular instance. Given the configuration, Bogor exhaustively explores the state-space of the BIR model in Fig. 3 for the

specified organization instance. That is, the BIR model is reusable for any model instance of the organization metamodel specified in Fig. 1.

We now describe the extensions used in Fig. 3: (1) `isTopGoalAchieved`: given a set of achieved goals, the extension determines whether the goal set can satisfy the requirement of achieving the top goal of an organization by looking at its goal structure, (2) `chooseGoal`: given a set of achieved goals, the extension non-deterministically chooses the next goal to be achieved. The extension leverages the *precedes* relation such that it does not choose goals whose preceding goals are not in the achieved goal set, which reduces the number of paths during the state-space explorations. In other words, `chooseGoal` non-deterministically chooses a goal from the active goal set G_A. The user can also specify to optimize paths on disjunctive goals as an option, i.e., by preventing to choose a goal whose disjunctive sibling goals are already achieved, (3) `chooseRole`: given a goal, the extension non-deterministically chooses the role that can achieve the goal based on the organization *achieves* function (i.e., when *achieves* gives a non-zero value), and (4) `chooseAgent`: given a role, the extension non-deterministically chooses the agent that can assume that role based on the organization *capable* function.

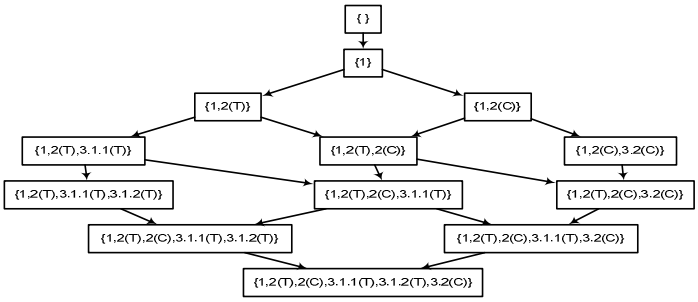

Fig. 4. Goal Achievement State-space (**G**) for CRFCC Example in Fig. 2

The `Search` thread explores all possible assignment sets that satisfy an organization's top goal. For optimization, we only store states in the beginning of each iteration of the `Search`'s loop. Fig. 4 presents the model's goal achievement state-space of the organization in Fig. 2 (without disjunctive goal optimization). The graph is generated based only on the goal structure (without considering roles and agents); Bogor can generate several state-spaces, for example, on goal ($\mathcal{G}$), goal-role ($\mathcal{GR}$), and goal-role-agent ($\mathcal{GRA}$). Each node in the figure represents a set of goals that has been achieved, and each edge represents an achievement of a goal. Note that each node in $\mathcal{G}$ implicitly represents the active goal set G_A, i.e., the set of goal achievements represented by the outgoing edges; thus, each path captures the sequence of active goal set G_A'. For goals that may be achieved at the same time, we follow the usual concurrency interleaving model that represents two transitions t_1 and t_2 that are executed at the same time as two paths $t_1 \rightarrow t_2$ and $t_2 \rightarrow t_1$. In the case where an organization cannot achieve the top goal, Bogor can give an empty (or a partial) state-space.

The CRFCC organization's goal diagram in Fig. 2 has a parameter n, which is the number of area that the agents have to clean up. Based on several experiments that we

have on varying *n*, we concluded that for the kinds of analyses that we perform, we do not actually need to have the actual concrete numbers of area; it is enough to focus on the characteristic of each area, i.e., whether it is tiled or carpeted. The reason is that we cannot actually distinguish between the areas at the design level; thus, for example, one carpeted area is the same as another carpeted area, i.e., one goal achievement sequence of one carpeted area would be similar to the other carpeted area's. Therefore, we only divide the area into two logical categories: carpeted (C) or tiled (T). This approach is akin to symmetry reduction [12] techniques usually used in model checking (e.g., the symmetric set mentioned previously), i.e., using one representative to reason about a set of entities that share the same properties. Section 8 describes extrapolation methods to recover the actual achievement sequences.

The key property of our analysis is that the state-space represents all possible ways to achieve the top goal even in the presence of agent failures/retries and malfunctions/recoveries. That is, an agent may retry several times before actually achieving a goal, or an agent malfunctions completely at some point in time, hence, the rest of the goals must be achieved by some other agents. In the end, if we take an actual system trace that achieves the organization top goal (with failures/retries and malfunctions), and if we project a sequence of the actual goal achievements for that trace, that sequence is in the state-space constructed by our analysis. For instance, let us consider the edge {}→{1}. This edge actually represents any system trace prefix that eventually achieves 1. For example, an agent A can be assigned to achieve 1 and then it somehow malfunctions without completing it, the system then reorganizes and assigns a different agent B to the goal. After several attempts, that B finally achieves 1. In a goal-agent state-space, this trace is represented by a path with prefix {}→{<1, B>} (and without A contributing to goal achievements in the path's suffix).

5 Design Metrics

Based on the analysis results presented in the previous sections, we have developed a set of metrics that can be used at design time to measure system performance. Specifically, in this paper we focus on a set of metrics based on path coverage in an attempt to measure the flexibility of the system. We define *system flexibility* as the ability of the system to reorganize to overcome individual agent failures. Ideally, such a metric would be unambiguous, simple to compute, and produce a small set of values that allows the designer to directly compare a set of possible system designs.

There are several pieces of coverage information that can be mined from the different state-spaces generated by Bogor. To measure system flexibility, we compare the state spaces of $\mathcal{G}$ and $\mathcal{GRA}$ for particular organization designs. Based on this approach, we have proposed the following metrics:

- **Covering Percentage:** For each path in **G**, we determine whether there exists a path in the **GRA**. For covering, we compute the percentage of paths in **G** that are covered in **GRA** (higher is better).
- **Coarse Redundancy:** For each path in **G**, we determine the number of paths in **GRA** (or **GR**) that cover it and give a coarse redundancy rate (paths in **GRA** divided by paths in **G**; higher is better).

There are other statistics that can be mined from the state-spaces (Section 7 describes more metrics). For example, given two GRAs (or GRs) of two different organization instances with the same goal diagram (hence, the same $\mathcal{G}$), we are able to compare coverage differences of the two with respect to $\mathcal{G}$. These coverage metrics allow designers to explore different role models and agent models for a given goal structure.

6 Metric Validation

We created four predefined robot teams to validate the metric set proposed in the previous section; we designed four different robot teams implementing RM0 and RM1 as defined in Section 3 for the goal model of Fig. 2. The robot teams were designed to provide a wide range of capabilities while keeping the same number of robots on each team at five. Each agent was given the capabilities to carry out exactly one of the application's leaf goals. The specific capabilities given to each robot are shown in Table 4. We want to predict and to compare the flexibility of each system.

Table 4. Agent System Designs

Name	AS0	AS1	AS2	AS3
A1	org, search, move, vacuum, sweep, mop	org, search, move, vacuum	org, search, move	org
A2	org, search, move, vacuum, sweep, mop	search, move, vacuum, sweep	search, move, vacuum	search, move
A3	org, search, move, vacuum, sweep, mop	vacuum, sweep, mop	vacuum, sweep	vacuum
A4	org, search, move, vacuum, sweep, mop	org, sweep, mop	sweep, mop	sweep
A5	org, search, move, vacuum, sweep, mop	org, search, move, mop	org, mop	mop

Table 5. Bogor Coverage Results

Organization (# paths in **G** = 10)	Coarse Redundancy (G-GRA) Rate	
	RM0	RM1
AS0	15625	15625
AS1	324	729
AS2	16	64
AS3	.3	1

We applied our analysis to the agent system designs; Table 5 presents Bogor analysis results for AS0-3 with RM0-1. For the experiments, we used an Opteron 248 workstation, Linux OS, and Java 5.0 (64-bit) with maximum heap of 256 MB; all the state-space analyses for $\mathcal{G}$ and $\mathcal{GRA}$ finished under 15 seconds (combined). All systems achieve 100% covering of $\mathcal{G}$ as there are agents that can achieve each goals (if a goal model has disjunctive sub-goals, it is possible to create organizations that can achieve the overall goal without agents that can achieve all disjunctive sub-goals). Based on the numbers, Bogor predicts that RM1 is more flexible than RM0, AS0 is the most flexible system, AS3 is the least flexible, and AS1 is more flexible than AS2.

To empirically evaluate the flexibility of designs AS0 – AS3 on the role models RM0 and RM1, we developed a simulation that stepped through the CRFCC applica-

tion. To measure the flexibility, we simulated capability failure. At each step in the simulation, a randomly selected assigned goal was achieved. Then, one robots capability was randomly selected and then tested to see whether or not it had failed. Based on a predefined *capability failure rate* (0 – 100%), we determined whether or not the selected capability had failed. For simplicity of presentation we used a single failure rate; however, the model could easily be extended to handle different failure rates. In addition, in contrast to the coarse redundancy metric that takes into account the possibility of agents to recover from a failure, we assumed once failed, a capability remained failed for the life of the system. Then, reorganization was performed to assign available robots to available goals and to de-assign robots if their capability had failed, and they were no longer able to play their assigned role.

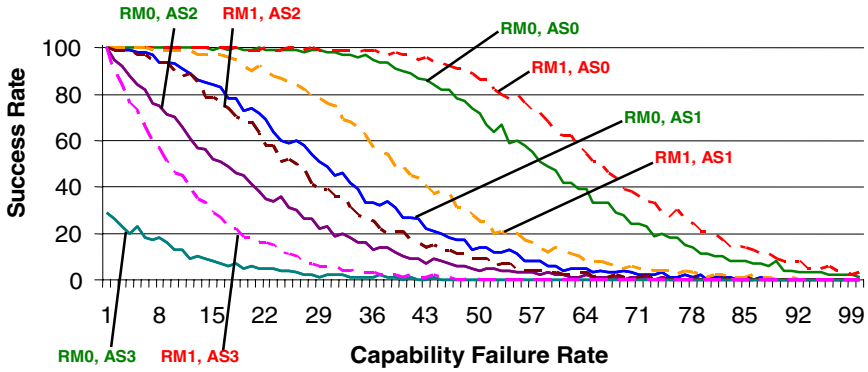

Fig. 5. Comparison of Role Model 0 vs. Role Model 1 for Agent Sets AS0 - AS3

Using a floor with 10 separate areas, we simulated each system (AS0 – AS3) on each role model (RM0 and RM1). For each role model, system combination was simulated for failure rates ranging from 0 to 100% for 1000 system executions. To compare the effectiveness of the role models using the four agent system designs, we looked at the results using each of the agent systems above. The results in Fig. 5 show that Role Model 1 provides more flexibility than Role Model 0. Furthermore, the simulation results confirm that AS0 is the most flexible while AS3 is the least one, and AS1 is more flexible than AS2. Note that the curve for (RM0, AS3) does not start at 100% since AS3 does not have an agent capable of playing the TileCleaner role.

The Bogor predictions and the simulation results make sense because: (1) in contrast to RM1, not all agents can assume the *TileCleaner* role in RM0, (e.g., A4 and A5 in AS3), (2) AS0 is the most flexible because each agent in AS0 can achieve any goal, (3) AS3 is the least flexible because each of its agents can assume at most one role, and (4) AS1 is more flexible than AS2 because AS1 agents have more capabilities.

6.1 Tradeoff Analysis

To demonstrate the usefulness of our metric in making design decisions, consider the following situation. Assume we have already developed a system based on RM1 and AS2, but now want to upgrade our system with a fixed budget. Our engineers deter-

mined that we could either (1) buy a single additional robot with three capabilities, or (2) buy five additional capabilities and integrate them onto our current robots. Essentially, option 2 equates to upgrading from AS2 to AS1 while option 1 would produce, for example, AS5 (option 1a) or AS6 (option 1b) as shown in Table 6.

Bogor's analysis results indicate that option 2 is better with a coarse redundancy rate of 729. The coarse redundancy rates for both option 1a and 1b are 216 while the original system (AS2) had a coarse redundancy rate of 64. Thus, using the coarse redundancy metric, we would choose option 2.

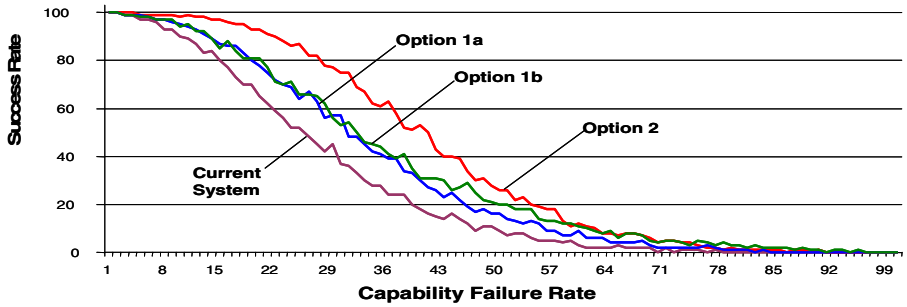

Fig. 6. Comparison of Possible System Updates

Table 6. Additional Agent System Designs Based on Agent Capabilities

Name	AS5	AS6
A1	org, search, move	org, search, move
A2	search, move, vacuum	search, move, vacuum
A3	vacuum, sweep	vacuum, sweep
A4	sweep, mop	sweep, mop
A5	org, mop	org, mop
A6	search, move, sweep	search, move, vacuum

To validate the metric results, we extended our simulation to include the definition of AS5 and AS6. The results of the four different options are shown in Fig. 6 where it is obvious that option 2 provides the best results followed by option 1a and 1b, which are very close. However, all three options are significantly better than the original system, which are consistent with the metric results that we obtained from Bogor.

7 Related Work

Software metrics as a subject area has been around for over 30 years. A number of metrics have been developed to predict or measure various parameters of software systems for different stages of software development lifecycle. For example, metrics to predict software performance were studied in [19, 20], software scalability in [20, 21], software adaptability in [17]. However, metrics and measures for intelligent software systems are as yet vaguely defined and sometimes controversial [2] and are not used extensively in software engineering [10]. There is also little work done in designing and applying metrics at the design level to predict adaptive systems per-

formance. We are proposing new design metrics and examine in details one such metric for distributed, adaptive systems in this paper.

Fault Tree Analysis (FTA) has been studied and used extensively [198]; it provides a top-down approach to systematically describe the combinations of possible occurrences in a system that results in undesirable outcomes such as failures or a malfunctions. Our approach complements FTA, since our technique *automatically* predicts the system flexibility for a given system configuration i.e., it generates traces of system behaviors. If failure patterns are exposed in the traces (e.g., an agent is not used after a certain time), then FTA can analyze the possible set of failure points.

8 Conclusions and Future Work

While the work presented in this paper is a starting point (i.e., there are many additional metrics that must be considered for providing a thorough design evaluation), there are several conclusions that can be made. Likewise, there are several assertions we can make about future work in the areas of performance prediction, additional metrics, scalability, and the integration of metric computations into a model design tool.

Performance Prediction: From the results presented in the previous section, it seems clear that the coarse redundancy rate does predict the flexibility of the robot systems. Unfortunately, system design is seldom as simple as maximizing one metric or parameter. Increased flexibility increases the number of possible assignments that can be made and thus increases the computation burden of generating near optimal assignments at run time. Obviously, a tradeoff exists. In future work, we hope to define additional predictive metrics that a designer can use to help tune the system at design time by performing tradeoff analysis. Our research will not eliminate this predicament, but give the designer predictive numbers to use in making those tradeoffs without developing expensive prototypes/simulations.

Additional Metrics and Query Environment: Based on the state-space analysis in Section 4, we believe the following metrics are helpful; however, we are still working on simulation methods to validate them:

- *Relative Cost Efficiency* (RCE): Using the *potential* function described in Section 3, we can determine path potentials in a goal-role-agent achievement state-space (**GRA**). This would be useful in defining a relative measure of the most/least efficient assignments and giving designers a feel for the organization's best/worst performance. (The actual best/worst performance of the system is not necessarily interesting as either all the agents may fail or the organization's goal may be achieved by changes in the environment.) Thus, the RCE metric would give reasonable feedback about organization instances. If the *potential* function always returns a constant value, thus, it reduces the metric to the shortest/longest achievement paths.
- *Relatively Optimistic Time Efficiency* (ROTE): This metric gives us the most optimistic best/worst time (logical ticks) to achieve the top goal. Consider the path A:{}→ B:{1}→ C:{1, 2(C)}→ D:{1, 2(T), 2(C)}→ E:{1, 2(T), 2(C), 3.1.1(T)}→ F:{1, 2(T), 2(C), 3.1.1(T), 3.2(C)}→ G:{1, 2(T), 2(C), 3.1.1(T), 3.1.2(T), 3.2(C)}. Note that optimistically, 2(C) and 2(T) can be achieved at the same time because

there are no precedence relation among them; similarly with 3.1.1(T) and 3.2 (C). Thus, if we group goal achievements that can happen at the same time, we have the sequence: {A}→{B}→{C, D}→{E, F}→{G}, i.e., 5 logical time ticks. This sub-path grouping approach is akin to partial order reduction techniques in model checking [6] where independent transitions can occur at the same time, thus, one ordering representative of the sub-path is enough. In our case, goal dependences are determined based on the precedence relation. Note that this grouping can also be done in the goal-role-agent achievement paths. If an agent cannot achieve goals simultaneously, the algorithm does not group goal achievements by the same agent in one group. Thus, system designers can evaluate different goal structures, role models, and agent systems for time efficiency. We plan to investigate using these dependence relations for partial order reduction in the near future.

We believe that there are more metrics that can be mined from the state-spaces of system designs. In addition, we also believe that work on query languages (e.g., [13]) can be used to ease system designers when evaluating multiagent system designs.

Extrapolation Methods for Scalability: Note that we do not actually need to use all five agents of the same type (i.e., agents with like capabilities) when exploring the state-space, for example, for AS0; it is sufficient to use one agent for each type (i.e., symmetry reduction [12]), and then extrapolate the actual number of paths based on the paths using representative agents. For example, if we use only one agent for AS0, the number of paths in $\mathcal{GRA}$ is 10. For each path, there are six goals achievements, thus, if we extrapolate each path when using five actual agents, we will have $10\times5^6=156250$ actual paths (which is the one we have from Bogor when directly using 5 agents). Thus, we believe that we can apply symmetry reduction on agent instances based on their type (i.e., they are indistinguishable at the design level) along with the partial order reduction technique hinted above, and use extrapolation methods to recover the actual paths for further analysis.

Integration of Metric Computations in a Model Design Environment: While we manually generated the Bogor configurations for this paper, it would be straightforward to automate such analysis by integrating Bogor into a multiagent design tool. We are currently developing agentTool III (aT^3), an advanced version of the agentTool system for developing organization-based multiagent systems [1]. aT^3 is being developed as an Eclipse plug-in and Bogor already works within the Eclipse plug-in environment. In the integrated system, designers will graphically create system goal, role, and agent models in aT^3 and will simply "click" on a button to popup an interface to select various analysis options; aT^3 will then automatically generate the appropriate configuration and invoke Bogor to explore its state-space and to predict its flexibility.

References

1. agentTool Website: http://macr.cis.ksu.edu/projects/agentTool/agentool.htm
2. Albus, J. S. Metrics and Performance Measures for Intelligent Unmanned Ground Vehicles. Proceedings of the 2002 Performance Metrics for Intelligent Systems Workshop, 2002.

3. Blau, P.M. & Scott, W.R., Formal Organizations, Chandler, San Fran., CA, 1962, 194-221.
4. Bogor Website: http://bogor.projects.cis.ksu.edu
5. Bosnacki, D., Dams, D., Holenderski, L. Symmetric Spin. Proceedings of the Seventh International SPIN Workshop. LNCS 1885, pp. 1-19, 2000.
6. Clarke, E., Grumberg, O., Peled, D. Model Checking. MIT Press, 2000.
7. DeLoach, S.A., & Matson, E. An Organizational Model for Designing Adaptive Multi-agent Systems. The AAAI-04 Workshop on Agent Organizations: Theory and Practice. 2004.
8. Dwyer, M.B., Hatcliff, J., Robby, Prasad, V.R. Exploiting Object Escape and Locking Information in Partial Order Reduction for Concurrent Object-Oriented Programs. International Journal of Formal Methods in System Design (FMSD), 25(2/3), pp. 199-240, 2004.
9. Ericson, C. Fault Tree Analysis – A History. Proceedings of the 17th International System Safety Conference – 1999.
10. Fenton, N.E., Neil, M. Software metrics: roadmap. ICSE-Future of SE, pp 357-370, 2000.
11. Holzmann, G. J. State Compression in SPIN: Recursive Indexing and Compression Training Runs. Proceedings of the Third International SPIN Workshop, 1997.
12. Ip, C. N., Dill, D. L. Better Verification Through Symmetry. International Journal of Formal Methods in System Design (FMSD), 9 (1/2), pp. 47-75, 1996.
13. Liu, Y. A., Stoller, S. D. Querying Complex Graphs. Proceedings of the Eighth Intl Symposium on Practical Aspects of Declarative Languages (PADL). Springer-Verlag, 2006. (to appear)
14. Matson, E., DeLoach, S. Capability in Organization Based Multi-agent Systems, Proceedings of the Intelligent and Computer Systems (IS '03) Conference, 2003.
15. Robby, Dwyer, M.B., Hatcliff, J. Bogor: An Extensible and Highly-Modular Model Checking Framework. Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003), pp. 267-276, 2003.
16. Robby, Dwyer, M.B., Hatcliff, J., Iosif, R. Space-Reduction Strategies for Model Checking Dynamic Software. Proceedings of the 2nd Workshop on Software Model Checking (SoftMC 2003). Electronic Notes in Theoretical Computer Science, 89 (3), Elsevier, 2003.
17. Subramanian, N., Chung, L., Metrics for Software Adaptability, Applied Technology Division, Anritsu Company, Richardson, TX, USA, 2000.
18. van Lamsweerde, A., Darimont, R., Letier, E. Managing conflicts in goal-driven requirements engineering. IEEE Transactions on Software Engineering. 24(11), pp 908-926, 1998.
19. Verkamo, A.I., Gustafsson, J., Nenonen, L., Paakki, J. Design patterns in performance prediction. Proceedings of the Second International Workshop on Software and Performance, ACM Press, pp 143-144, September 2000.
20. Weyuker, E. J., Avritzer, A. A metric for predicting the performance of an application under a growing workload. IBM Systems Journal, Vol 41, No 1, pp. 45-54, 2002.
21. Weyuker, E. J., Avritzer, A. A Metric to Predict Software Scalability. Proceedings of the Eight IEEE Symp. on Software Metrics (METRICS 2002), Ottawa, Canada, pp. 152-159, June 2002.