

Trace-based Specification of Law and Guidance Policies for Multiagent Systems*

Scott J. Harmon, Scott A. DeLoach, Robby

Kansas State University, Manhattan KS 66506, USA
{harmon, sdeloach, robb}@ksu.edu

Abstract. Policies have traditionally been a way to specify properties of a system. In this paper, we show how policies can be applied to Organization-based Multiagent Systems Engineering (O-MaSE) [1], specifically in the Organization Model for Adaptive Computational Systems (OMACS) [2]. In OMACS, policies may constrain assignments of agents to roles, the structure of the goal model for the organization, or how an agent may play a particular role. In this paper, we focus on policies limiting system traces; this is done to leverage the work already done for specification and verification of properties in concurrent programs. We show how traditional policies can be characterized as *law policies*; that is, they must always be followed by a system. In the context of multiagent systems, law policies limit the flexibility of the system. Thus, in order to preserve the system flexibility while still being able to guide the system into preferring certain behaviors, we introduce the concept of *guidance policies*. These *guidance policies* need not always be followed; when the system cannot continue with the *guidance policies*, they may be suspended. We show how this can guide how the system achieves the top-level goal while not decreasing flexibility of the system. *Guidance policies* are formally defined and, since multiple *guidance policies* can introduce conflicts, a strategy for resolving conflicts is given.

1 Introduction

As computer systems have been charged with solving problems of greater complexity, the need for distributed, intelligent systems has increased. As a result, there has been a focus on creating systems based on interacting autonomous agents. This investigation has created an interest in multiagent systems and multiagent system engineering, which proscribes formalisms and methods to help software engineers design multiagent systems. One aspect of multiagent systems that is receiving considerable attention is the area of policies. These policies have been used to describe the properties of a multiagent system—whether that be behavior or some other design constraints. Policies are essential in designing societies of agents that are both predictable and reliable [3]. Policies have traditionally been interpreted as properties that must always hold. However, this does not capture the notion of policies in human organizations, as they are often used as normative guidance, not strict laws. Typically, when a policy cannot be followed in

* This work was supported by grants from the US National Science Foundation (0347545) and the US Air Force Office of Scientific Research (FA9550-06-1-0058).

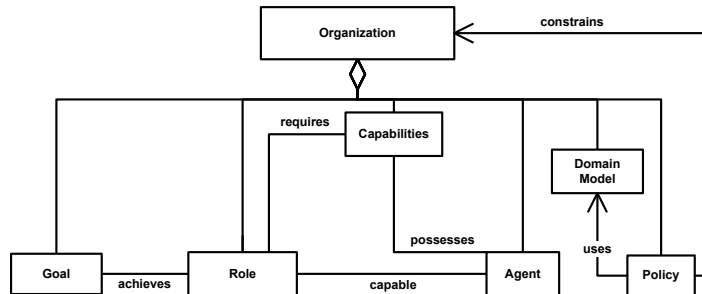


Fig. 1. Organization Model for Adaptive Computational Systems.

a multiagent system, the system cannot achieve its goals, and thus, it cannot continue to perform. In contrast, policies in human organizations are often suspended in order to achieve the overall goals of the organization. We believe that such an approach could be extremely beneficial to multiagent systems residing in a dynamic environment. Thus, we want to enable developers to guide the system without constraining it to the point where it cannot function effectively or loses its autonomy.

The main contributions of this paper are: (1) a *formal* trace-based foundation for *law* (must always be followed) and *guidance* (need not always be followed) policies, (2) a conflict resolution strategy for choosing between which guidance policies to violate, and (3) validation of our approach through a set of simulated multiagent systems.

The rest of the paper is organized as follows: In Section 2, we give some background on multiagent systems policies along with two multiagent system examples. In Section 3, we define the notion of *system traces* for a multiagent system, which are later used to describe policies. Section 4 defines *law policies* as well as *guidance policies*; we give examples and show how *guidance policies* are useful for multiagent systems and describe a method for ordering guidance policies according to importance. Section 5 presents and analyzes experimental results from applying policies to the two multiagent system examples. Section 6 concludes and presents some future work.

2 Background

Policies have been considered for multiagent systems for some time. Efforts have been made to characterize, represent, and reason [4] about policies in the context of multiagent systems. Policies have been referred to as laws in the past. Yoav Shoham and Moshe Tennenholtz wrote in [5] about *social laws* for multiagent systems. They showed how policies could help a system to work together, similar to how our rules of driving on a predetermined side of the road help the traffic to move smoothly. There has also been work on detecting global properties [6] of a distributed system, which could in turn be used to suggest policies for that system. Policies have also been proposed as a way to help assure that agents and that the entire multiagent system behave within certain boundaries. They have also been proposed as a way to specify security constraints in multiagent systems [7, 8]. There has been work to define policy languages by defining a description logic [9]. Policies have also been referred to as *norms*. Much

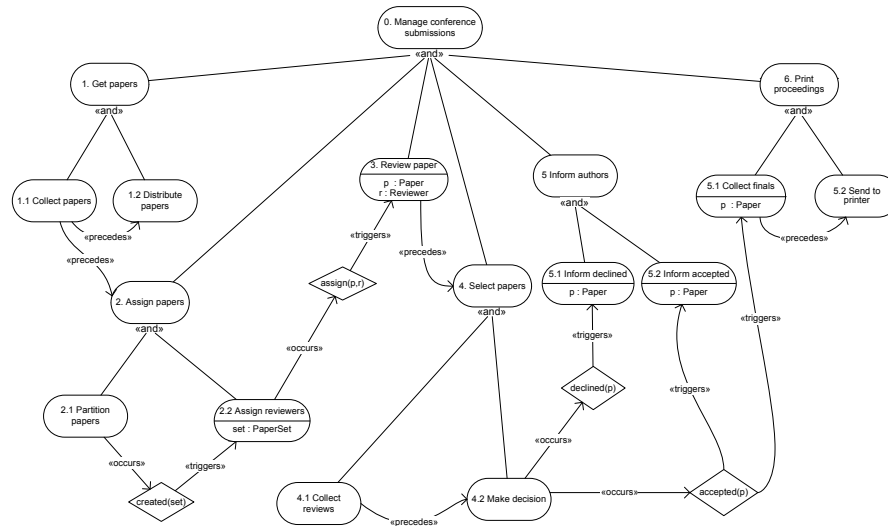


Fig. 2. Conference Management Goal Model.

work has been done on the formal specification of these norms [10]. We are taking this formal approach in our specification of guidance and law policies. Norms, however, are usually associated with *open systems*—while we are concerned with *closed, cooperative systems*. We want to use formal methods to prove whether a given system will abide by the policies as expected. Thus, we must give our guidance policies for multiagent societies a solid formal foundation. In order to achieve this end, we borrow concepts that are widely used in program analysis, in particular, model checking. Taking a model checking approach to policies has been done [11] and is a natural extension of program analysis.

The multiagent systems model we are using for this paper is called the Organization Model for Adaptive Computational Systems (OMACS) [2]. Figure 1 is a graphical depiction of the OMACS model. OMACS defines standard multiagent system components such as goals, roles, capabilities, and agents. Roles *achieve* goals, agents *posses* capabilities, and agents are *capable* of playing roles depending on the capabilities they *posses*. The organization, which represents the entire set of agents, decides which agents to *assign* to what roles to *achieve* particular goals. When the organization makes an *assignment* of an agent to a particular role to achieve a specific goal, the organization is constrained by agents capabilities as well as any applicable policies. To model goals, we use the Goal Model for Dynamic Systems (GMDs) as defined in [12]. Events may occur while an agent is playing a role. These events may *trigger* (activate) goals. Only active goals may be assigned to an agent.

2.1 Conference Management Example

A well known example in multiagent systems is the Conference Management [13, 14] example. The Conference Management example models the workings of a scientific conference, for example, authors submit papers, reviewers review the submitted papers,

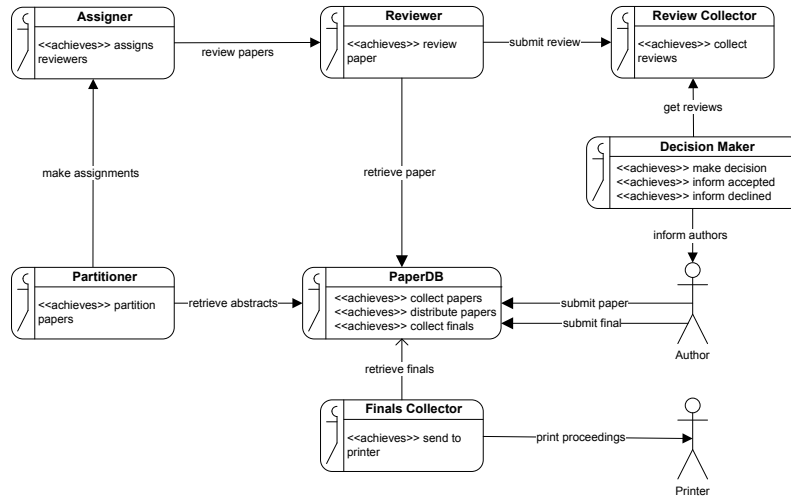


Fig. 3. Conference Management Role Model.

and certain papers are selected for the conference and printed in the proceedings. Figure 2 shows the complete goal model for the conference management example, which we are using to illustrate our policies. In this example, a multiagent system represents the goals and tasks of a generic conference paper management system. Goals of the system are identified and are decomposed into subgoals.

The top-level goal, *0. Manage conference submissions*, is decomposed into several “and” subgoals, which means that in order to achieve the top goal, the system must achieve all of its subgoals. These subgoals are then associated through precedence and trigger relations. The *precedes* arrow between goals indicates that the source of the arrow must be *achieved* before the destination can become active. The *triggers* arrow indicates that the domain-specific event in the source may trigger the goal in the destination. The *occurs* arrow from a goal to a domain-specific event indicates that while pursuing that goal, said event may occur. A goal that triggers another goal may trigger multiple instances of that goal.

Leaf goals are goals that have no children. The leaf goals in this example consist of *Collect papers*, *Distribute papers*, *Partition papers*, *Assign reviewers*, *Collect reviews*, *Make decision*, *Inform accepted*, *Inform declined*, *Collect finals*, and *Send to printer*. For each of these leaf goals to be achieved, agents must play specific roles. The roles required to achieve the leaf goals are depicted in Figure 3. The role model gives seven roles as well as two outside actors. Each role contains a list of leaf goals that the role can achieve. For example, the *Assigner* role can achieve the *Assign reviewers* leaf goal. In GModS, roles only achieve leaf goals. The arrows between the roles indicates interaction between particular roles. For example, once the agent playing the *Partitioner* role has some partitions, it will need to hand off these partitions to the agent playing the *Assigner* role. OMACS allows an agent to play multiple roles simultaneously, as long as it has the capabilities required by the roles and it is allowed by the policies.

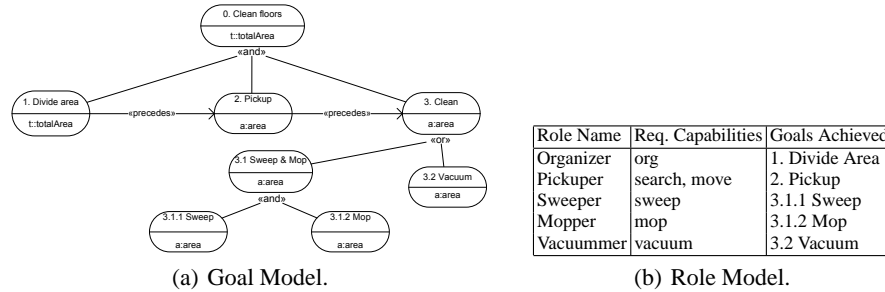


Fig. 4. CRFCC Models.

2.2 Robotic Floor Cleaning Example

Another example to illustrate the usefulness of the concept of guidance policies is the Cooperative Robotic Floor Cleaning Company Example (CRFCC), which was first presented by Robby et al. in [15]. In this example, a team of robotic agents clean the floors of a building. The team has a map of the building as well as indications of whether a floor is tile or carpet. Each team member will have a certain set of capabilities (e.g. vacuum, mop, etc). These capabilities may become defective over time. In their analysis, Robby et al. showed how breaking up the capabilities affected a team’s flexibility to overcome loss of capabilities. We have extended this example by giving the information that the vacuum cleaner’s bag needs to be changed after vacuuming three rooms. Thus, we want to minimize the number of bag changes. For this we introduce a guidance policy and show how it affects the performance of the organization.

The goal model for the CRFCC system is fairly simple. As seen in Figure 4(a), the overall goal of the system (Goal 0) is to clean the floors. This goal is decomposed into three conjunctive subgoals: 1. *Divide Area*, 2. *Pickup*, and 3. *Clean*. The 3. *Clean* goal is decomposed into two disjunctive goals: 3.1 *Sweep & Mop* and 3.2 *Vacuum*. Depending on the floor type, only one subgoal must be achieved to accomplish the 3. *Clean* goal. If an area needs to be swept and mopped (i.e. it is tile), then goal 3.1 *Sweep & Mop* is decomposed into two conjunctive goals: 3.1.1 *Sweep* and 3.1.2 *Mop*. After an agent achieves the 1. *Divide area* goal, a certain number of 2. *Pickup* goals will become active (depending on how many pieces the area is divided into). After the 2. *Pickup* goals are completed, a certain number of 3. *Clean* goals become active, again depending on how many pieces the area was broken into. This then will activate goals for the tile areas (3.1.1 *Sweep* and 3.1.2 *Mop*) as well as goals for the carpeted areas (3.2 *Vacuum*).

Figure 4(b) gives the role model for the CRFCC. In this role model, each leaf goal of the system is achieved by a specific role. The role model may be designed many different ways depending on the system’s goal, agent, and capability models. Thus, depending on the agents and capabilities available, the system designer may choose different role models. For this paper, we will look at just one of these possible role models. In the role model in Figure 4(b), the only role requiring more than one capability is the *Pickuper* role. This role will require both the *search* and *move* capability. Thus, in order to play this role, an agent must possess both capabilities.

Event	Definition	Property	Definition
$C(g_i)$	goal g_i has been completed.	$a.reviews$	the number of reviews agent a has performed.
$T(g_i)$	goal g_i has been triggered.	$a.vacuumedRooms$	the number of rooms agent a has vacuumed.
$A(a_i, r_j, g_k)$	agent a_i has been assigned role r_j to achieve goal g_k .		

(a) System Events. (b) Properties

Fig. 5. Events and Properties of Interest.

3 Multiagent Traces

There are several observable events in an OMACS system. A *system event* is simply an action taken by the system. In this paper, we are concerned with specific actions that the organization takes. For instance, an assignment of an agent to a role is a system event. The completion of a goal is also a system event. In an OMACS system, we can have the system events of interest shown in Figure 5(a).

At any stage in a multiagent system, there may be certain properties of interest. Some may be domain-specific (only relevant to the current system), while others may be general properties such as the number of roles an agent is currently playing. State properties that are relevant to the examples we are presenting in the next section are shown in Figure 5(b).

3.1 System Traces

In order to describe multiagent system execution, we use the notion of a system trace. An (abstract) *system trace* is a projection of system execution with only desired state and event information preserved (role assignments, goal completions, domain-specific state property changes, etc). In this paper, we are only concerned with the events and properties given above and only traces that result in a successful completion of the system goal. Let E be an event of interest and P be a property of interest. A *change of interest* in a property is a change for which a system designer has made some policy. For example, if a certain integer should never exceed 5, a change of interest would be when that integer became greater than 5 and when that integer became less than 5. Thus a change of interest in a property is simply an abstraction of all the changes in the property. ΔP indicates a change of interest in property P . A system trace may contain both events and changes of interest in properties. Changes of interest in properties may be viewed as events, however, for simplicity we include both and use both interchangeably. Thus, a system trace is defined as:

$$E_1 \rightarrow E_2 \rightarrow \dots \quad (1)$$

As shown in equation 1, a trace is simply a sequence of events. An example subtrace of a multiagent system, where g_1 is a goal, a_1 is an agent, and r_1 is a role, might be:

$$\dots T(g_1) \rightarrow A(a_1, r_1, g_1) \rightarrow C(g_1) \dots \quad (2)$$

Formula 2 means that goal g_1 is triggered, then agent a_1 is assigned role r_1 to achieve goal g_1 , finally, goal g_1 is completed.

We use the terms *legal trace* and *illegal trace*. An *illegal trace* is an execution we do not want our system to exhibit, while a *legal trace* is an execution that our system may exhibit. Intuitively, policies cause some traces to become *illegal*, while others remain *legal*.

We are able to use the notion of system traces because the framework we are using to build multiagent systems constructs mathematically specified models of various aspects of the system (goal model, role model, etc.). This can be leveraged to formally specify policies as restriction of system traces. Once we have a formal definition of system traces, we can leverage existing research on property specification and concurrent program analysis.

4 Policies

Policies may restrict or proscribe behaviors of a system. Policies concerning agent assignments to roles have the effect of constraining the set of possible assignments. This can greatly reduce the search space when looking for the optimal assignment set [16].

Other policies can be used for verifying that a goal model meets certain criteria. This allows the system designer to more easily state properties of the goal model that may be verified against candidate goal models at design time. For example, one might want to ensure that our goal model in Figure 2 will always trigger a *Review Paper* goal for each paper submitted.

Yet, other policies may restrict the way that roles can be played. For example, *when an agent is moving down the sidewalk it always keeps to the right*. These behavior policies also restrict how an agent interacts with its environment, which in turn means that they can restrict protocols and agent interactions. One such policy might be that an agent playing the *Reviewer* role must always give each review a unique number. These sort of policies rely heavily on domain-specific information. Thus it is important to have an ontology for relevant state and event information prior to designing policies [17].

4.1 Language for policy analysis

To describe our policies, we use temporal formula with quantification similar to [18]. This may be converted into Linear Temporal Logic (LTL) [19] or Büchi automata [20] for infinite system traces, or to something like Quantified Regular Expressions [21] for finite system traces. The formulas consist of predicates over goals, roles, events, and assignments (recall that an assignment is the joining of an agent and role for the purpose of achieving a goal). The temporal operators we currently use are as follows: $\Box(x)$, meaning x holds always; $\Diamond(x)$, meaning x holds eventually; and $x \mathcal{U} y$, meaning x holds until y holds.¹ We use a mixture of state properties as well as events [22] to obtain compact and readable policies. An example of one such policy formula is:

$$\forall a_1 : Agents, \mathcal{L} : \Box(\text{sizeOf}(a_1.\text{reviews}) \leq 5) \quad (3)$$

Formula 3 states that it should always be the case that each agent never review more than five papers. The $\mathcal{L} :$ indicates that this is a *law policy*. The property *.reviews* can be

¹ We only reason about bounded liveness properties because we only consider successful traces.

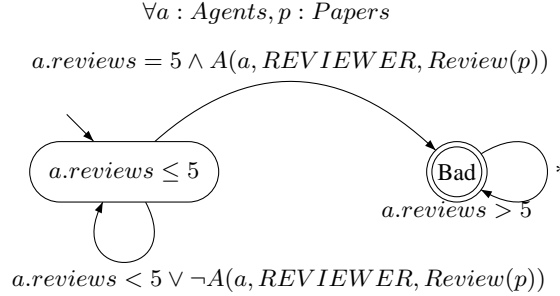


Fig. 6. No agent may review more than five papers.

considered as part of the system’s state information. This is domain-specific and allows a more compact representation of the property. This policy may be easily represented by a finite automata as shown in Figure 6.

The use of the $A()$ predicate in Figure 6 indicates an assignment of the *Reviewer* role to achieve the *Review paper* goal, which is parametrized on the paper p . This automata depicts the policy in Formula 3, but in a manner for a model checker or some other policy enforcement mechanism to detect when violation occurs. The accepting state indicates that a violation has occurred. Normally, this automata would be run alongside the system, either at design time with a model checker [23], or at run-time with some policy enforcement mechanism [24].

4.2 Law Policies

The traditional notion of a policy is a rule that must always be followed. We refer to these policies as *law policies*. An example of a law policy with respect to our conference management example would be *no agent may review more than five papers*. This means that our system can never assign an agent to the *Reviewer* role more than five times. A law policy can be defined as:

$$\mathcal{L} : Conditions \rightarrow Property \quad (4)$$

Conditions are predicates over state properties and events, which, when held true, imply that the *Property* holds true. The *Conditions* portion of the policy may be omitted if the *Property* portion should hold in all conditions, as in Formula 3.

Intuitively, for the example above, no trace in the system may contain a subtrace in which an agent is assigned to the *Reviewer* role more than five times. This will limit the number of legal traces in the system. In general, *law policies reduce the number of legal traces for a multiagent system*. The policy to limit the number of reviews an agent can perform is helpful in that it will ensure that our system does not overburden any agent with too many papers to review. This policy as a pure law policy, however, could lead to trouble in that the system may no longer be able to achieve its goal. Imagine that more papers than expected are submitted. If there are not sufficient agents to spread the load, the system will fail since it is cannot assign more than five papers to any agent. This is a common problem with using only law policies. They limit the *flexibility* of the system, which we define as *how well the system can adapt to changes* [15].

Node	Definition
P_1	No agent should review more than 5 papers.
P_2	PC Chair should not review papers.
P_3	Each paper should receive at least 3 reviews.
P_4	An agent should not review a paper from someone whom they wrote a paper with.

Table 1. Conference Management Policies.

4.3 Guidance Policies

While the policy in (3) is a seemingly useful policy, it reduces flexibility. To overcome this problem, we have defined another, weaker type of policy called *guidance policies*. Take for example the policy used above, but as a *guidance policy*:

$$\forall a_1 : Agents, \mathcal{G} : \square(\text{sizeOf}(a_1.\text{reviews}) \leq 5) \quad (5)$$

This is the same as the policy as in (3) except for the \mathcal{G} :, which indicates that it is a *guidance policy*. In essence, the formalization for guidance and law policies are the same, the difference is the intention of the system designer. *Law policies* should be used when the designer wants to make sure that some property is always true (e.g. for safety or security), while *guidance policies* should be used when the designer simply wants to guide the system.

This policy limits our agents to reviewing no more than five papers, *when possible*. Now, the system can still be successful when it gets more submissions than expected since it can assign more than five papers to an agent. When there are sufficient agents, however, the policy still limits each agent to five or fewer reviews.

In the definition of *guidance policies*, we have not specified how the system should choose which guidance policy to violate in a given situation. We propose a partial ordering of guidance policies to allow the system designer to set precedence relationships between guidance policies. We arrange the guidance policies as a lattice, such that a policy that is a parent of another policy in the lattice, is *more-important-than* its children. By analyzing a system trace, one can determine a set of policies that were violated during that trace. This set of violations may be computed by examining the policies and checking for matches against the trace. When there are two traces that violate policies with a common ancestor, and one (and only one) of the traces violate the common ancestor policy, we mark the trace violating that common ancestor policy as illegal. Intuitively, this trace is illegal because the system could have violated a less important policy. Thus, if the highest policy node violated in each of the two traces is an ancestor of every node violated in both traces, and that node is not violated in both traces, then we know the trace violating that node is illegal and should not have happened.

Take, for example, the four policies in the Table 1. Let these policies be arranged in the lattice shown in Figure 7(a). The lattice in Figure 7(a) means that policy P_1 is more important than P_2 and P_3 , and P_2 is more important than P_4 . Thus, if there is any trace that violates any guidance policies other than P_1 (and does not violate a law policy), it should be chosen over one which violates P_1 .

When a system cannot achieve its goals without violating policies, it may violate guidance policies. There may be traces that are still illegal, though, depending on the

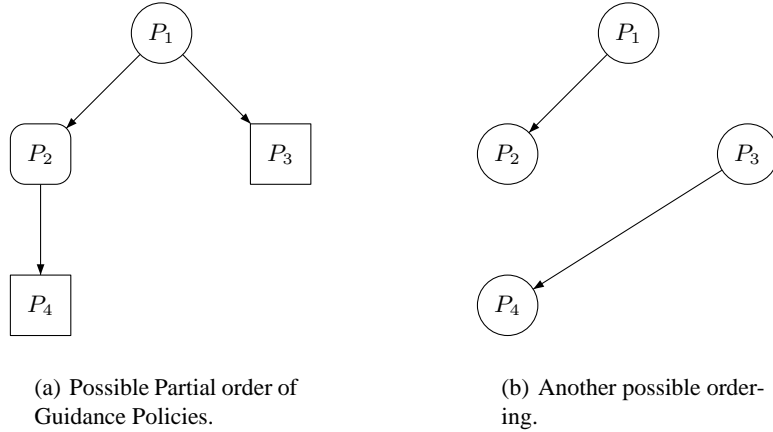


Fig. 7. Partial orders of Guidance Policies.

ordering between policies. For every pair of traces, if the least upper bound of the policies violated in both traces, let us call this policy violation \mathcal{P} , is in one (and only one) of the traces, the trace with \mathcal{P} is illegal. For example, consider the ordering in Figure 7(a), let trace t_1 violate P_1 and P_2 , while trace t_2 violates P_2 and P_3 . Round nodes represent policies violated in t_1 , box nodes represent policies violated in t_2 , and boxes with rounded corners represent policies violated in both t_1 and t_2 . Since P_1 is the least upper bound of P_1 , P_2 , and P_3 and since P_1 is not in t_2 , t_1 is illegal.

As shown in Figure 7(b), the policies may be ordered in such a way that the policy violations of two traces do not have a least upper bound. If there is no least upper bound, \mathcal{P} , such that \mathcal{P} is in one of the traces, the two traces cannot be compared and thus both traces are legal. The reason they cannot be compared is that we have no information about which policies are more important. Thus, either option is legal. It is important to see here that all the guidance policies do not need to be ordered into a single lattice. The system designer could create several unrelated lattices. These lattices then can be iteratively refined by observing the system behaviors or by looking at metrics generated for a certain policy set and ordering (e.g., [15]). This allows the system designer to influence the behavior of the system by making logical choices as to what paths are considered better. Using the lattice in Figure 7(a), we may even have the situation where P_1 is not violated by either trace. In this case, the violation sets cannot be compared, and thus, both traces are legal. In situations such as these, the system designer may want to impose more ordering on the policies.

Intuitively, guidance policies constrain the system such that at any given state, transitions that will not violate a guidance policy are always chosen over transitions that violate a guidance policy. If guidance policy violation cannot be avoided, a partial ordering of guidance policies is used to choose which policies to violate.

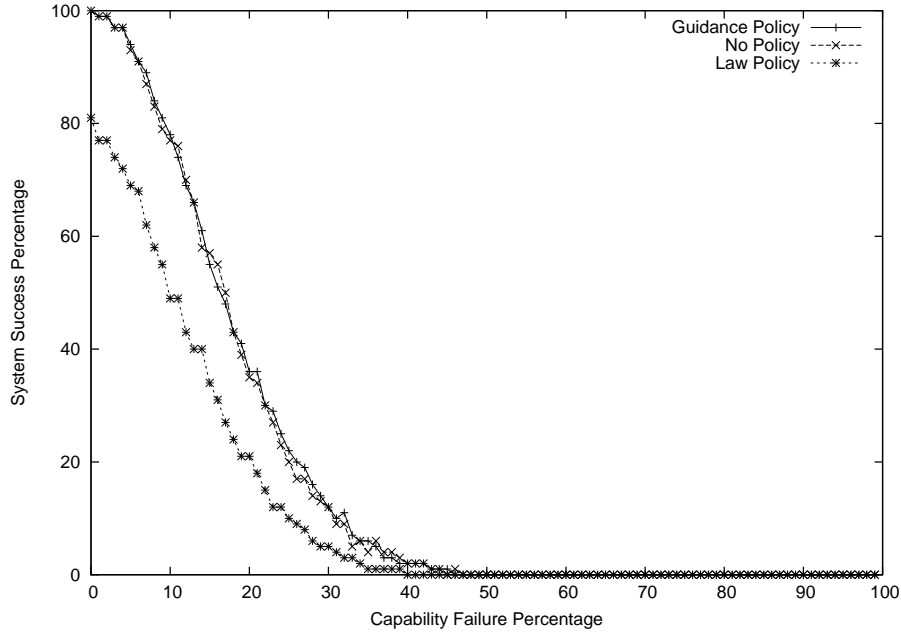


Fig. 8. The success rate of the system given capability failure.

5 Evaluation

5.1 CRFCC

Using our CRFCC example and a modified simulator from [15], we collected results running simulations with the guidance policy: *no agent should vacuum more than three rooms*. We contrast this with the law policy: *no agent may vacuum more than three rooms*. The guidance policy is presented formally in Equation 6.

$$\forall a_1 : Agents, \mathcal{G} : \Box(a_1.vacuumedRooms \leq 3) \quad (6)$$

For this experiment, we used five agents each having the following capabilities: a_1 , org, search, and move; a_2 , search, move, and vacuum; a_3 , vacuum and sweep; a_4 , sweep and mop; and a_5 , org and mop. These capabilities restrict the roles our simulator can assign to particular agents. For example, the Organizer role may only be played by agent a_1 or agent a_5 , since those are the only agents with the *org* capability. In the simulation we randomly choose capabilities to fail based on a probability given by the *capability failure rate*.

For each experiment, the result of 1000 runs at each capability failure rate was averaged. At each simulation step, a goal being played by an agent is randomly achieved. Using the capability failure rate, at each step, a random capability from a random agent may be selected to fail. Once a capability fails it cannot be repaired.

Figure 8 shows that while the system success rate decreases when we enforce the law policy, it does not, however, decrease when we enforce the guidance policy. Figure 9 shows the total number of times the system assigned vacuuming to an agent who

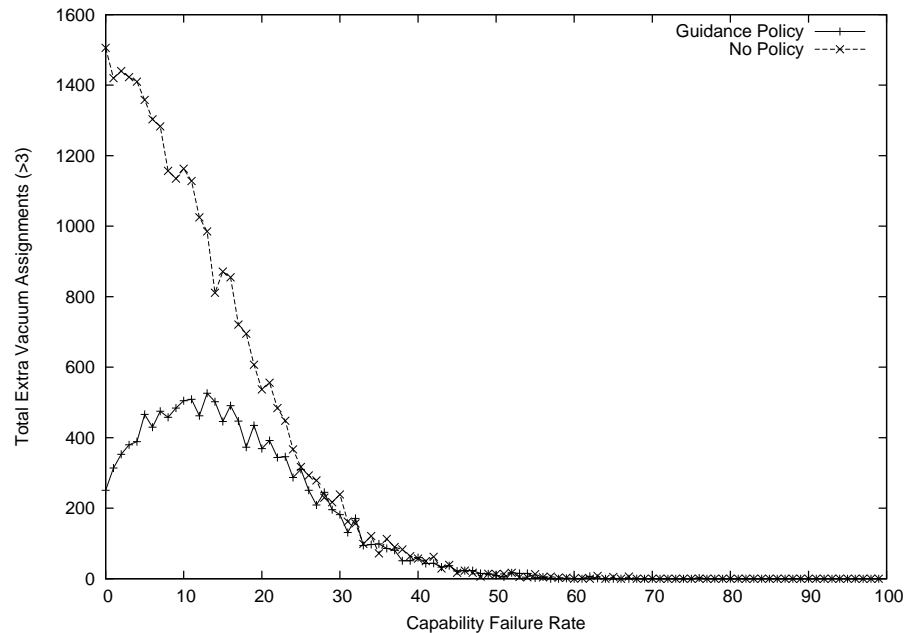


Fig. 9. The extra vacuum assignments given capability failure.

already vacuumed at least 3 rooms for 1000 runs of the simulation at each failure rate. With no policy, it can be seen that the system will in fact assign an agent to vacuum more than 3 rooms quite often. With the guidance policy, however, the extra vacuum assignments (> 3) stay minimal. The violations of the guidance policy increase as the system must adapt to an increasing failure of capabilities until it reaches a peak. At the peak, increased violations do not aid in goal achievement and eventually the system cannot succeed even without the policy. Thus, the system designer may now wish to purchase equipment with a lower rate of failure, or add more redundancy to the system to compensate. The system designer may also evaluate the graph and determine whether the cost of the maximum number of violations exceeds the maximum cost he is willing to incur, and if not, make appropriate adjustments.

5.2 Conference Management System

We also simulated the conference management system described in Section 2.1. We held the number of agents constant, while increasing the number of papers submitted to the conference. The system was constructed with a total of 13 agents, 1 *PC Member* agent, 1 *Database* agent, 1 *PC Chair* agent, and 10 *Reviewer* agents. The simulation randomly makes goals available to achieve, while still following the constraints imposed by GMoDS. Roles that achieve the goal are chosen at random as well as agents that can play the given role. The policies are given priority using the *more-important-than* relation as depicted in Figure 7(a).

Figure 10 shows a plot of how many times a guidance policy is violated versus the number of papers submitted for review. For each set of paper submissions (from 1 to

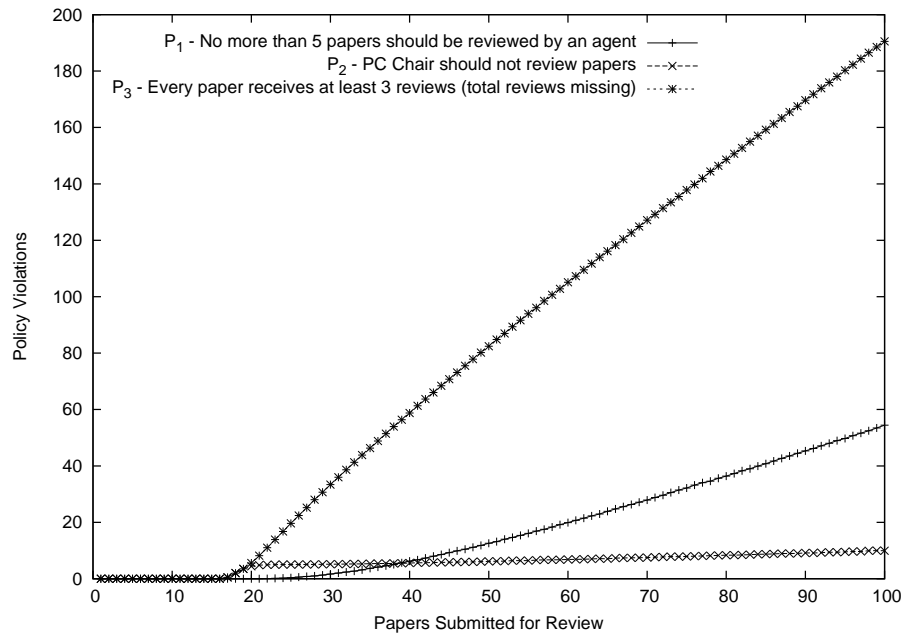


Fig. 10. Violations of the guidance policies as the number of papers to review increases.

100) we ran the simulation 1000 times and then took the average of the 1000 runs to determine the average number of violations. In all the runs the system succeeded in achieving the top level goal.

As seen by the graph in Figure 10, no policies are violated until around 17 papers (this number is explained below). The two least important policies (P_2 and P_3) are violated right away. The violation of P_2 , however, levels off since it is interacting with P_1 . The violations of P_3 is seen to grow at a much greater rate since it is the least important policy.

We then changed all the guidance policies to law policies and re-ran the simulation. For 17 or more submissions, the system always failed to achieve the top level goal. This makes sense because we have only 10 Reviewer agents and we have the policies: the PC Chair should not review papers and no agent should review more than 5 papers. This means the system can only produce $5 \times 10 = 50$ reviews. But, since we have the policy that each paper should have at least 3 reviews, 17 submissions would need $17 \times 3 = 51$ reviews. For 16 or fewer papers submitted, the law policies perform identical to the guidance policies.

5.3 Common Results

As the experimental results in Figure 8 show, guidance policies *do not decrease the flexibility of a system to adapt to a changing environment*, while law policies *do decrease the flexibility of a system to adapt to a changing environment*. Guidance policies, however, do help guide the system and improve performance as shown in Figure 9 and

Figure 10. The partial ordering using the *more-important-than* relation helps a system designer put priorities on what policies they consider to be more important and helps the system decide which policies to violate in a manner consistent with the designer's intentions.

6 Conclusions and Future Work

Policies have proven to be useful in the development of multiagent systems. However, if implemented inflexibly, situations such as described in [25] in which a policy caused a spacecraft to crash into an asteroid will occur. Guidance policies allow a system designer to guide the system while giving it a chance to adapt to new situations.

With the introduction of guidance policies, policies are an even better mechanism for describing desired properties and behaviors of a system. It is our belief that guidance policies more closely capture how policies work in human organizations. Guidance policies allow for more flexibility than law policies in that they may be violated under certain circumstances. In this paper, we demonstrated a technique to resolve conflicts when faced with the choice of which guidance policies to violate. Guidance policies, since they may be violated, can have a partial ordering. That is, one policy may be considered more important than another. In this manner, we allow the system to make better choices on which policies to violate. Traditional policies may be viewed as *law policies*, since they must never be violated. Law policies are still useful when the system designer never wants a policy to be violated—regardless of system success. Such policies might concern security or human safety.

Policies may be applied in an OMACS system by constraining assignments of agents to roles, the structure of the goal model for the organization, or how the agent may play a particular role. Through the use of OMACS, the metrics described in [15], and the policy formalisms presented here, we are able to provide an environment in which a system designer may formally evaluate a candidate design, as well as evaluate the impact of changes to that design without deploying or even completely developing the system.

Policies can dramatically improve run-time of reorganization algorithms in OMACS as shown in [16]. Guidance policies can be a way to achieve this run-time improvement without sacrificing system flexibility. The greater the flexibility, the better the chance that the system will be able to achieve its goals.

Policies are an important part of a multiagent system. Future work is planned to ease the expression and analysis of policies. Some work has already been done in this area [26, 27], but it has not been integrated with a multiagent system engineering framework. Another area of work is to provide a verification framework from design all the way to implementation. The goal would be to determine the minimum guarantees needed from the agents to guarantee the overall system behavior specified by the policies. These minimum guarantees could then be checked against the agent implementations to determine whether the implemented system follows the policies given.

Guidance policies add an important tool to multiagent policy specification. However, with this tool comes complexity. Care must be taken to insure that the partial ordering given causes the system to exhibit the behavior intended. Tools which can

visually depict the impact of orderings would be helpful to the engineer considering various orderings. We are currently working on inferring new policies from a given set of policies. For example, if a system designer wanted to get their system to a state for which they defined policy, we would automatically generate guidance policies. This could be useful when the policies are defined as finishing moves in chess. That is they proscribe optimal behavior, given a state. Thus, we would like to get to the state where we know that optimal behavior. Another exciting area of research is to determine a method of dynamically learning guidance policies, which would allow an organization to evolve within its changing environment.

References

1. DeLoach, S.A.: Engineering organization-based multiagent systems. In: Software Engineering for Multi-Agent Systems IV. Volume 3914 of Lecture Notes in Computer Science., Springer-Berlin/Heidelberg (May 2006) 109–125
2. DeLoach, S.A., Oyenon, W.H.: An organizational model and dynamic goal model for autonomous, adaptive systems. Multiagent & Cooperative Robotics Laboratory Technical Report MACR-TR-2006-01, Kansas State University (March 2006)
3. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. *Computer* **36**(1) (2003) 41–50
4. Bradshaw, J., Uszok, A., Jeffers, R., Suri, N., Hayes, P., Burstein, M., Acquisti, A., Benyo, B., Breedy, M., Carvalho, M., Diller, D., Johnson, M., Kulkarni, S., Lott, J., Sierhuis, M., Hoof, R.V.: Representation and reasoning for DAML-based policy and domain services in KAoS and Nomads. In: AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems, New York, NY, USA, ACM Press (2003) 835–842
5. Shoham, Y., Tennenholtz, M.: On social laws for artificial agent societies: Off-line design. *Artificial Intelligence* **73**(1-2) (1995) 231–252
6. Stoller, S.D., Unnikrishnan, L., Liu, Y.A.: Efficient detection of global properties in distributed systems using partial-order methods. In: *Computer Aided Verification*. (2000) 264–279
7. Kagal, L., Finin, T., Joshi, A.: A policy based approach to security for the semantic web. In: *The SemanticWeb - ISWC 2003*. Volume 2870 of Lecture Notes in Computer Science., Springer-Berlin/Heidelberg (2003) 402–418
8. Paruchuri, P., Tambe, M., Ordóñez, F., Kraus, S.: Security in multiagent systems by policy randomization. In: AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems, New York, NY, USA, ACM Press (2006) 273–280
9. Uszok, A., Bradshaw, J., Jeffers, R., Suri, N., Hayes, P., Breedy, M., Bunch, L., Johnson, M., Kulkarni, S., Lott, J.: Kaos policy and domain services: toward a description-logic approach to policy representation, deconfliction, and enforcement. In: *POLICY 2003: IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, IEEE (2003) 93–96
10. Artikis, A., Sergot, M., Pitt, J.: Specifying norm-governed computational societies. *ACM Transactions on Computational Logic* (2007)
11. Viganò, F., Colombetti, M.: Symbolic Model Checking of Institutions. *Proceedings of the 9th International Conference on Electronic Commerce* (2007)
12. Miller, M.: A goal model for dynamic systems. Master's thesis, Kansas State University (April 2007)

13. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Organisational rules as an abstraction for the analysis and design of multi-agent systems. *International Journal of Software Engineering and Knowledge Engineering* **11**(3) (2001) 303–328
14. DeLoach, S.A.: Modeling organizational rules in the multi-agent systems engineering methodology. In: *Advances in Artificial Intelligence: 15th Conference of the Canadian Society for Computational Studies of Intelligence (AI 2002)*. Volume 2338 of *Lecture Notes in Computer Science.*, Springer-Berlin/Heidelberg (May 2002) 1–15
15. Robby, DeLoach, S.A., Kolesnikov, V.A.: Using design metrics for predicting system flexibility. In: *Fundamental Approaches to Software Engineering (FASE 2006)*. Volume 3922 of *Lecture Notes in Computer Science.*, Springer-Berlin/Heidelberg (March 2006) 184–198
16. Zhong, C., DeLoach, S.A.: An investigation of reorganization algorithms. In: *Proceedings of the International Conference on Artificial Intelligence (IC-AI'2006)*, CSREA Press (June 2006) 514–517
17. DiLeo, J., Jacobs, T., DeLoach, S.: Integrating ontologies into multiagent systems engineering. In: *Fourth International Conference on Agent-Oriented Information Systems (AIOS-2002)*, CEUR-WS.org (July 2002)
18. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Robby: Expressing checkable properties of dynamic systems: The bandera specification language. *International Journal on Software Tools for Technology Transfer (STTT)* **4**(1) (2002) 34–56
19. Manna, Z., Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag (1991)
20. Büchi, J.R.: On a decision method in restricted second-order arithmetics. In: *Proceedings of International Congress of Logic Methodology and Philosophy of Science, Palo Alto, CA, USA, Stanford University Press (1960)* 1–12
21. Olender, K., Osterweil, L.: Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering* **16**(3) (1990) 268–280
22. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: State/event-based software model checking. In Boiten, E.A., Derrick, J., Smith, G., eds.: *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM '04)*. Volume 2999 of *Lecture Notes in Computer Science.*, Springer-Verlag (April 2004) 128–147
23. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: *Model Checking*. The MIT Press (1999)
24. Ligatti, J., Bauer, L., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. In: *International Journal of Information Security*. Volume 4., Springer-Verlag (2004) 2–16
25. Peña, J., Hinchey, M.G., Sterritt, R.: Towards modeling, specifying and deploying policies in autonomous and autonomic systems using an AOSE methodology. *EASE* **0** (2006) 37–46
26. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: *Proceedings of the 1999 International Conference on Software Engineering*, IEEE (May 1999)
27. Smith, R.L., Avrunin, G.S., Clarke, L.A., Osterweil, L.J.: Propel: an approach supporting property elucidation. In: *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, New York, NY, USA, ACM Press (2002) 11–21