

# Abstract Requirement Analysis in Multiagent System Design

Scott J. Harmon, Scott A. DeLoach, and Robby  
Multiagent and Cooperative Robotics Laboratory  
Kansas State University, Manhattan, Kansas, USA  
{harmon, sdeloach, robbey}@ksu.edu

**Abstract**—“Good, fast, or cheap, pick two.” What drives designers to make decisions on how to architect a system? The stake-holder has certain abstract qualities in mind: efficiency, quality, reliability, and so forth. How do we make sure our system is guided by these qualities? What happens when the system cannot always provide all the qualities? We describe a framework for analyzing a design allowing decisions about what qualities are more important to be made at design-time.

**Keywords**-policy; norms; AOSE; O-MaSE; guidance;

## I. INTRODUCTION

Organization-based multiagent systems engineering has been proposed as a way to design complex adaptable systems [4]. Agents interact and can be given tasks depending on their individual capabilities. The system designer is faced with the task of designing a system to not only meet functional requirements, but also non-functional requirements. System designers need tools to help them generate specification and evaluate design decisions early in the design process. Conflicts within non-functional requirements should be uncovered and, if necessary, the stakeholders should be consulted to help resolve those conflicts. The approach we are taking is to first generate a set of system traces using the models generated at design time. Second, we analyze these system traces, using additional information provided by the system designer. Third, we generate a set of policies that will guide the system toward the abstract qualities desired. And, fourth, we analyze the generated policies for conflicts.

The main contributions of this paper are: (1) a formal method for generating specifications at design time for multiagent systems from abstract qualities, (2) a method of automatically discovering conflicts in abstract qualities given a system design, and (3) a way to analyze these conflicts.

The remainder of this paper is organized as follows. In Section II, we give some background on the models used in our multiagent systems and describe some related work. In Section III, we present metrics constructed using the ISO 9126 Software Qualities document. Policy generation and conflict discovery are given in Sections IV and V respectively. Conflict analysis is presented in Section VI. Section VII presents and analyzes experimental results from the application of our policy generation on a multiagent system example. Section VIII concludes and presents some ideas for future work.

## II. BACKGROUND

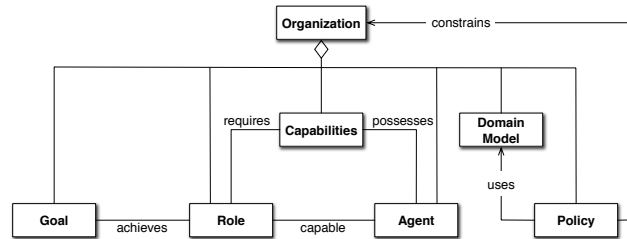


Figure 1: OMACS Metamodel.

The multiagent systems model that we are using throughout the paper is the Organization Model for Adaptive Computational Systems (OMACS) [4]. The basic OMACS metamodel is given in Figure 1. OMACS defines standard multiagent system entities and their relationships. Agents are *capable* of playing roles. Roles can *achieve* goals. Policies *constrain* the organization. The organization (which is comprised of agents) *assigns* agents to play specific roles in order to *achieve* specific goals. The organization may make these assignments through a centralized approach (a single agent makes the assignments), or through a distributed approach. The Goal Model for Dynamic Systems (GMoDS) [9] is used to model our goal structures. GMoDS allows for such things as AND/OR decomposition of goals, as well as, *precedence* between goals and *triggering* of new goal instances (i.e. while trying to achieve a particular goal, an event may cause another goal to become active). Instance goals (triggered goals) may have parameters that specialize them for the task at hand. For example, a *Search* goal may be parametrized on the coordinates of the area to search.

Organization-based Multiagent System Engineering (O-MaSE) [5] brings the entities and relationships in OMACS together into a set of models. These models are supported by AgentTool III <sup>1</sup>, an analysis and design tool for multiagent systems. The models are the *Goal Model*, *Role Model*, and *Agent Model*. The *Goal Model* defines the goals of the system and their relationships as specified by GMoDS. The *Role Model* defines the Roles as well as their relationships with Goals and Capabilities. The *Agent Model* defines the Agents and their relationships with Capabilities.

<sup>1</sup><http://agenttool.projects.cis.ksu.edu/>

Policies (or norms) have been used in multiagent system engineering for some time. Various languages, frameworks, enforcement and checking mechanisms have been used [1], [11], [12]. Taking a model checking perspective (e.g. [13]), we say that policies *restrict* the behavior of multiagent systems. We view the multiagent system design as a set of states. The states may contain non-domain specific properties such as goal achievement history or domain specific properties such as the type of floor an agent currently occupies. Policies are rules designed to restrict this set of states. This restriction may happen at design time or at runtime. Guidance policies restrict the behavior of a multiagent system without sacrificing flexibility [6]. We use (guidance) policies as formal rules that are applied to our system. This is consistent with usage in formalizations such as KAoS [12].

Guidance policies are a trace-based formalization of policies ‘that need not always be followed’[6]. They must be followed when the system can still progress toward achieving its goal. If the system cannot continue, the guidance policies may be temporarily suspended. Guidance policies may be arranged in a *more-important-than* relation, creating a set of lattices. The policies are suspended from least-important to most-important. Thus it is possible to have conflicting guidance policies and yet still have a valid and viable (can still progress toward achieving its goal) system.

There has been work in incorporating abstract qualities into multiagent systems. Tropos defines the concept of a soft-goal [2] that describes abstract qualities for the system. These soft-goals, however, are mainly used to track decisions in the goal model design for human consideration. Some work has been done on model checking multiagent systems [10], [13]. Automated policy generation has been used for online learning [7]. These methods help the multiagent system better tune to the environment in which it is deployed. Chiang et al. [3] have automated learning of policies for mobile ad-hoc networks. Their learning was an offline approach using simulation to generate specific policies from general ‘objectives’ and possible configurations.

### III. QUALITY METRICS

ISO 9126 [8] defines a set of qualities for the evaluation of software. This document breaks down software qualities into six general areas: functionality, reliability, usability, efficiency, maintainability, and portability. These characteristics are broad and may apply in different ways to different types of systems. In our research, we identified a group of metrics that evaluate multiagent system traces in order to illustrate our concepts. Each metric is formally defined and may be measured precisely over the current system design.

#### A. System Traces

The set of traces form a tree. We use this structure to determine what choices to make at each decision point in

order to maximize (or minimize) some metric. The choice may be what role or what agent to use in order to achieve a goal, or even what goal to pursue (in the case of OR goals). It is important to note that the divergence in the traces may also happen by changes in goal parameters, which are normally beyond the control of the system. Thus, the metrics and policies generated may need to take into account some aspect of the goal parameters.

We used Bogor [10] to generate the system traces using the models defined by the OMACS meta-model. Bogor is an extensible, state of the art model checker. Our customized Bogor uses an extended GModS goal model, a role model, and an agent model to generate traces of a multiagent system. We have extended the GModS model with a maximum and minimum bounds on the *triggers* relationship in order to bound the exploration of the trace-space in the model checker. The traces consist of a sequence of *agent goal assignment* achievements. We generate only traces in which the top-level goal is achieved (system success).

An *agent goal assignment* is defined as a tuple,  $\langle G_i(x), R_j, A_k \rangle$ , containing a parametrized goal  $G_i(x)$ , a role  $R_j$ , and an agent  $A_k$ . The goal’s parameters  $x$  may be any domain specific specialization of the goal  $G_i$  given at the time the goal is dispatched (triggered) to the system.

#### B. Efficiency

We are using a trace-length based definition of efficiency. The shorter the trace the more efficient is the top-level goal achievement. Our strategy here is to make assignments such that we minimize the total expected trace length. This minimization will be dynamic in that it will take into consideration the current trace progress. Thus we consider goal achievements the system has made when determining shortest trace length to pursue. We then convert this logic statically into a set of policies that when enforced will exhibit the same behavior as a minimization algorithm, given current system goal achievements. Given the initial system state, we prefer to make assignments to achieve the overall shortest path. Thus, we will be generating directing policies proscribing assignments (may still present multiple options). Expected trace length is defined in Formula 1.

$$\text{ExpLength}(\xi) = \sum_{i=1}^n \frac{1}{1 - pf_i} \quad (1)$$

$\xi$  represents a single trace, while  $pf_i$  is the probability of failure for the assignment achievement  $i$  within that trace. An *assignment achievement* is the accomplishment of an assignment by an agent. Thus, an *assignment achievement failure* is a failure of the agent to complete its assignment. The probability of failure for the agent goal assignment achievement ( $pf_i$ ) is the maximum probability of failure for all the capabilities required for the role in the assignment:

$$pf_i = \max_{c_x \in \text{capreq}(As_i)} Pf(c_x, As_i) \quad (2)$$

It is evident here that some traces will have an infinite expected length (probability of failure is 100%).

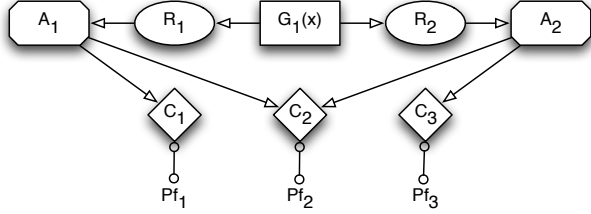


Figure 2: Goal Choice & Capability Failure Probabilities

We are concentrating on the assignment of goals to agents in the system. A goal assignment choice is depicted in Figure 2.  $G_1(x)$  represents the parametrized goal that needs to be achieved.  $R_1$  and  $R_2$  are two different roles that are able to achieve the goal  $G_1$ .  $A_1$  and  $A_2$  are two agents that possess capabilities required to play  $R_1$  and  $R_2$  respectively. The failure probabilities are connected to each capability.

### C. Quality of Product

Quality of achieved goals falls under the ISO software quality of Functionality. Here we define quality as a measure of the quality of goal achievement. Quality of goal achievement may depend on the role, agent, and goal (including parameters). In our analysis, we limit ourselves to these influences although goal achievement quality may depend on environment, history of the system, and current assignments.

Certain roles may obtain a better result when achieving certain goals, likewise certain agents may play certain roles better than other agents. These properties can be known at design time. Usually this intuition is known by the implementers and they may manually design an agent goal assignment algorithm to favor these assignments.

In our analysis and experiments, we mapped assignments to scores. The higher the score, the higher the quality of product. The scores can be specified over the entire assignment, or just portions. For example, role  $R_1$  may achieve goal  $G_1$  better than role  $R_2$  when the parameters of  $G_1$  is of class  $x$ . Agents who produce higher quality products could also be part of the score determination.

The trace score is the average quality of product. We realize some products may be more important than others, however, we use an average as given in Formula 3. Let  $\xi_i$  be the  $i$ th agent goal assignment in trace  $\xi$  of length  $n$ .

$$\text{Qual}(\xi) = \sum_{i=1}^n \frac{\text{score}(\xi_i)}{n} \quad (3)$$

### D. Reliability

Sometimes failure should be avoided at all costs, thus, even if there is a probabilistically shorter trace, it could be the case that we choose the longer trace because we want to minimize the chance of any failure. Formally, we want to minimize goal assignment failures. We do this by

minimizing the probability of capability failure. Our strategy here is to pick the minimal failure trace given the current completed goals in the system.

We can use the capability failure matrix defined for Efficiency. The score we are minimizing is defined as:

$$\text{Fail}(\xi) = \sum_{i=1}^n pf_i \quad (4)$$

Where  $pf_i$  is defined as in the Efficiency metric (the probability of failure of assignment  $i$  within trace  $\xi$ ).

It is important here to see the distinction between Reliability and Efficiency. Efficiency is concerned with minimizing the expected trace length, while Reliability is concerned with minimizing the total number of failures. Thus, if we are pursuing Efficiency, we may choose a path in which there may be some failures, even through there is a path with no failures, because the expected trace length is shorter in the path with failures. Reliability will always choose a path with less expected failures, even if the path is longer than another.

## IV. POLICY GENERATION

To construct our policies, we first generate a set of traces using our OMACS models as input to our customized Bogor model checker. We then run a metric over each trace, giving it a score. The aim here is to create policies to guide the system to select the highest (or lowest) scoring trace, given any subtrace prefix. Thus, we create a set of assignments that will guide the system toward the maximum scoring traces. There may be many traces with the same maximum score, in this case we have a set of options. This selection is illustrated in Figure 3. We generate policies that, when followed, guide the system to traces that look like the highest scoring traces. Thus, for the figure, we proscribe that from state  $S_1$ , you must make goal agent assignments that are in the highest scoring traces ( $S_2, S_2'$ , etc). For every subtrace, we generate a set of agent goal assignment options. This may lead to many policies, for this reason, after we generate the initial set of policies, we prune and generalize them. Policies will be of the form:

$$[\text{guard}] \rightarrow \alpha_1 \vee \alpha_2 \vee \dots \quad (5)$$

where  $\alpha_i$  is a generalized agent goal assignment and  $[\text{guard}]$  is a conditional on the state of the system. The guard is also specified in terms of the achieved agent goal assignments.

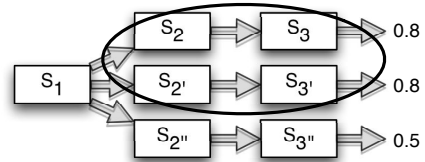


Figure 3: System Traces to Prefer

We use two methods to reduce the size of the policy sets generated. One method is to prune useless policies. Since

we produce a policy for every prefix trace, we may end up proscribing actions when the system did not initially even have a choice in the matter (the system was already forced to take that choice). We find these policies by checking the traces matched by the policies’ left-hand side (the guard). If a policy only matches one trace, we can prune that policy as it had no effect on the system.

The second method combines multiple policies through generalization. If two or more policies offer the same choices for assignments (meaning their right hand sides are the same), the common pieces of the left hand side are computed. If the new left hand construct (the common pieces of the two policies) matches a non-empty intersection of traces with the current policies and the right-hand side of the new policy is not a subset of the right-hand side of each of the matching policies, the potential policy is discarded. Otherwise we remove the two policies that we have combined and add the new combined policy. We repeat this procedure until we do not combine any more policies.

## V. CONFLICT DISCOVERY

Now that we are automatically generating policies for different abstract qualities, we may generate conflicting policies. After discovering these conflicts, we may use a partial ordering of policies to overcome them or decide to rework our initial system design.

Recall the policy structure in Formula 5, since we have this policy structure and we are generating policies for different qualities, we may have different types of conflicts between policies generated for different qualities. These conflicts may be discovered by examining the guard of each policy and checking if there is a non-empty intersection of traces where both guards are active. If the intersection is non-empty and if the assignment choices are not equal, there is a possibility of conflict. Now, it may be the case that the right hand side of both AND’d together are satisfiable with the OMACS constraints, for instance, that only one agent may play a particular instance goal at once (Formula 6).

$$Sat((\alpha_x \vee \alpha_y \vee \alpha_z) \wedge (\alpha_a \vee \alpha_b \vee \alpha_c) \wedge \beta) \quad (6)$$

Even if the formula is satisfiable, there may still be a need for partial ordering between the policies. Since there may be agent failure, we want to know what policies to relax first.

We can partition the conflicts into two sets: *definitely conflicts* and *possibly conflicts*. The *definitely conflicts* elements will always conflict given the system design. The *possibly conflicts* will only conflict if the configuration of the system changes, i.e., in the case where there is capability or agent loss. If we have statistics on capability failure, we can even compute a probability of conflict. This probability could help the designer determine if the possibility of conflict is likely enough to spend more resources on overcoming it. Some conflicts may be inherent in the design, due to constraints on agents and capabilities or because of a sub-optimal

configuration. Being able to see these conflicts as early in the design process (and especially before implementation) will greatly help as it is much cheaper to change the design earlier rather than later.

If policies generated from different abstract qualities *definitely conflict*, then this is an indicator to the system designer that with the current constraints (agents, roles, and goals), it is not possible to satisfy all of the stake-holder’s abstract requirements. The designer and/or stake holder must modify the models by adding agents, changing goals, or by relaxing or redefining the abstract requirement.

We can choose to resolve the conflicts by specifying which quality we prefer in each conflicting case. The designer may prefer efficiency over quality in certain cases and quality over efficiency in other cases. This choice will be a conscious decision by the designer (perhaps after consulting the stake-holders), and thus, a more engineered approach than the ad-hoc and unclear decisions that might be inadvertently made by implementers.

## VI. CONFLICT ANALYSIS

Once we generate policies using the design models and the abstract qualities desired, we can analyze any potential conflicts between the abstract qualities by analyzing the conflicts between the generated policies. We could simply take each policy from a set of policies generated for a particular quality and determine if it conflicts with any policy generated for a different quality. This makes our potential conflict space large and unmanageable. If we are only analyzing conflicts between two qualities, the potential conflict space is  $P_1 \times P_2$ . Where  $P_1$  and  $P_2$  are the sets of policies generated to support quality  $Q_1$  and  $Q_2$  respectively.

To make the conflict analysis more tractable for the designer, we make an analysis of the parts of the policies that are the point of conflict. These points of conflict tend to be repeated. Thus, by looking at only the points of conflict we are able to summarize the conflict information in a format to allow a system designer to more easily make decisions concerning the conflicts.

## VII. EVALUATION

We took the Conference Management System (CMS) example described in [6] and extended it with additional choices for achieving goals by modifying the goal, role, and agent models. The CMS example models the workings of a scientific conference, authors submit papers, reviewers review the submitted papers, and certain papers are selected for the conference and printed in the proceedings. Goals of the system are identified and decomposed into subgoals.

The top-level goal, *Manage 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. Leaf goals are goals that have no subgoals. The leaf goals in this example consist of *Collect papers*,

Cap./Assign.	$\langle *, *, PR \rangle$	$\langle Thr, *, SR \rangle$	$\langle App, *, SR \rangle$	$\langle *, Part, NPC \rangle$	$\langle *, Part, RPC \rangle$	$\langle *, Part, BPC \rangle$
<i>ReviewerInterface</i>	0	0.5	0			
<i>PCMemInterface</i>				0.1	0	0.5

Figure 4: Capability Failure Probabilities

*Distribute papers, Partition papers, Review papers, Collect reviews, Make decision, Inform accepted, Inform declined, Collect finals, Master CD, and Print Proceedings.* For each leaf goals to be achieved, agents must play specific roles.

We augmented our models with a capability failure matrix and a partial ordering. In our assignments ‘\*’ represents the wild card element. Agent type abbreviations are given as:

Abbr.	Agent Type
SR	Specialized Reviewer
PR	Plain Reviewer
NPC	Normal Program Chair
RPC	Rookie Program Chair
BPC	Busy Program Chair

Figure 4 gives an abbreviated capability failure matrix for our system. *ReviewerInterface* and *PCMemInterface* are capabilities defined in the CMS role model. The assignment is represented by a tuple containing parameters, goal, and agent. ‘Thr’ and ‘App’ represent theory and application paper parameters respectively, while ‘Part’ represents the *Partition* goal. Thus, the *ReviewerInterface* fails with a 50% probability when being used by the *Specialized Reviewer* on goals parametrized by theory papers. Our quality of product ordering is depicted in Figure 5. Nodes represent *agent goal assignments*, while a directed edge from node  $N_1$  to  $N_2$  indicates that the assignment in  $N_1$  produces a higher quality of product than the assignment in  $N_2$ .

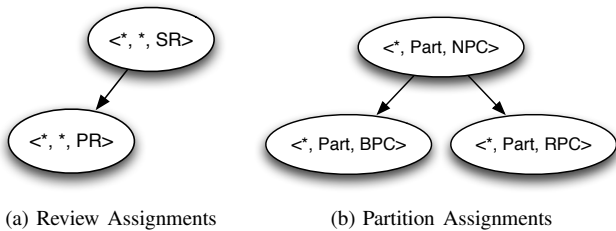
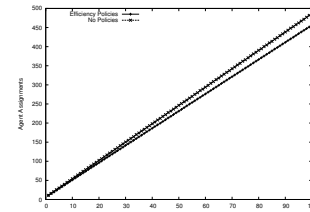


Figure 5: Quality of Product Ordering

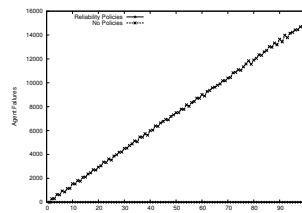
We ran our customized Bogor on the goal, role, and agent models to generate traces for our system. This produced 2592 unique system traces. Using this information along with our failure matrix and quality orderings, we then produced policies to guide the system toward the abstract qualities desired by the system designer.

We loaded our policies into a simulation engine, which ran the Conference Management System to test the effects of having the generated policies. The simulator picks a random role and agent who is capable of achieving a goal without violating any policies (or, if necessary for progress,

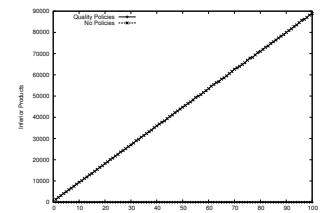
violating guidance policies) and achieves an active goal. This is repeated until either the top-level goal is achieved or we cannot make an more progress (system failure). We varied the number of papers submitted to the system from 1 to 100 and took an average of 1000 runs for each number of papers.



(a) Efficiency



(b) Reliability



(c) Quality

Figure 6: CMS Results (Lower is Better)

Figure 6a shows the results of the policies generated using the efficiency metric on the number of assignments used to achieve the top-level goal. The top line is without policies while the lower line is with the generated policies enabled. The policies are clearly having an effect on the system assignments (and thus efficiency).

To evaluate the impact of our reliability policies, we measure the number of agent assignment failures. That is, we counted the number of times an agent failed to achieve an assigned goal. For the runs using the generated policies, it is not surprising that we did not have a single agent failure. The runs without employing the policies, however had many agent failures as depicted in Figure 6b (note line on the x-axis).

To evaluate the quality policies, we measured how many times the system produced a lower quality result when it could have produced a higher one. Figure 6c (note line on the x-axis) shows that without the generated policies, our system produced much lower quality products than with the generated policies. We have also run similar experiments with other multiagent system design models such as those described in [7] and obtained similar results.

## VIII. CONCLUSIONS AND FUTURE WORK

We have provided a framework for stating, measuring, and evaluating abstract quality requirements against potential multiagent system designs. We do this by generating policies that can be used either as a formal specification, or dynamically within a given multiagent system to guide the system toward the maximization (or minimization) of the abstract quality constraints. These policies are generated offline, using model checking and automated trace analysis.

Our method allows the system designer to see potential conflicts between abstract qualities that can occur in their system. This allows them to resolve these conflicts early in the system design process. The conflicts can be resolved using an ordering of the qualities which can be parametrized on domain specific information in the goals. They could also cause a system designer to decide to change their design to better achieve the quality requirements. This is easier, since we are in the system design and not in the implementation.

These policies can be seen to guide the system toward the abstract qualities as defined in the metrics. Our experiments showed significant performance, quality, and reliability increases over a system using the same models, but without the automated policy generation.

We are further evaluating the effect of applying multiple qualities within the same multiagent system. We expect that with conflict resolution, the designer will be able to prioritize the different qualities in different situations. One approach is to use an unsatisfiable-core to determine the smallest points of conflict in order to suggest the easiest way to resolve the conflict. Another approach involves the use of a minimizing satisfaction algorithm to suggest the best way to resolve the conflict while minimizing costs (economical, temporal, etc).

Automated abstraction of the goal parameters using classification is an area that needs to be explored. If we had a more precise notion of the goal parameters, we would better be able to generate policies targeting attributes of the goal parameters. Another area that can be explored is splitting policies to target conflicts more precisely. This of course needs to be balanced with generality (we would not want one policy for each possible scenario). More metrics over the multiagent system traces should be developed. For example, Security also falls under the ISO software quality of Functionality. Focusing on information flow, we could generate a set of policies to minimize information dissemination (maximizing privacy).

### Acknowledgments

This research was performed as part of grants provided by the Air Force Office of Scientific Research (FA9550-06-1-0058) and the National Science Foundation (IIS-0347545).

## REFERENCES

- [1] A. Artikis, M. Sergot, and J. Pitt. Specifying norm-governed computational societies. *ACM Transactions on Computational Logic*, 2007.
- [2] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, pages 203–236, 2004.
- [3] C. Chiang, G. Levin, Y. Gottlieb, R. Chadha, S. Li, A. Poylisher, S. Newman, R. Lo, and T. Technologies. On Automated Policy Generation for Mobile Ad Hoc Networks. *Policies for Distributed Systems and Networks, 2007. POLICY'07. Eighth IEEE International Workshop on*, pages 256–260, 2007.
- [4] S. A. DeLoach, W. Oyenan, and E. T. Matson. A capabilities based theory of artificial organizations. *Journal of Autonomous Agents and Multiagent Systems*, 2007.
- [5] J. C. Garcia-Ojeda, S. A. DeLoach, Robby, W. H. Oyenan, and J. Valenzuela. O-MaSE: A customizable approach to developing multiagent development processes. In *Proceedings of the 8th International Workshop on Agent Oriented Software Engineering*, Honolulu, HI, May 2007.
- [6] S. J. Harmon, S. A. DeLoach, and Robby. Trace-based specification of law and guidance policies for multiagent systems. *Engineering Societies in the Agents World VIII*, 4995:333–349, 2007.
- [7] S. J. Harmon, S. A. DeLoach, Robby, and D. Caragea. Leveraging organizational guidance policies with learning to self-tune multiagent systems. In *The Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, October 2008.
- [8] ISO. *ISO/IEC: 9126 Information technology-Software Product Evaluation-Quality characteristics and guidelines for their use*. International Organization for Standardisation (ISO), 1991.
- [9] M. Miller. A goal model for dynamic systems. Master's thesis, Kansas State University, April 2007.
- [10] Robby, S. A. DeLoach, and V. A. Kolesnikov. Using design metrics for predicting system flexibility. In *Fundamental Approaches to Software Engineering (FASE 2006)*, volume 3922 of *Lecture Notes in Computer Science*, pages 184–198. Springer-Berlin/Heidelberg, March 2006.
- [11] Y. Shoham and M. Tennenholtz. On social laws for artificial agent societies: Off-line design. *Artificial Intelligence*, 73(1-2):231–252, 1995.
- [12] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott. 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*, pages 93–96. IEEE, 2003.
- [13] F. Viganò and M. Colombetti. Model checking norms and sanctions in institutions. *Coordination, Organizations, Institutions, and Norms in Agent Systems III*, pages 316–329, 2008.