Using the Gatekeeper Concept to Design and Assign Multiagent Teams

Marie D. Manner
University of Minnesota
Minneapolis, MN, USA
manner@cs.umn.edu

M. Birna van Riemsdijk
TU Delft
Delft, The Netherlands
m.b.vanriemsdijk@tudelft.nl

Catholijn M. Jonker
TU Delft
Delft, The Netherlands
c.m.jonker@tudelft.nl

ABSTRACT

There are environments where multiple agents can take on different roles, but where any single agent may not have all the required capabilities for a role. We need a way to group agents appropriately such that the combination of their capabilities forms a team which satisfies the role requirements. We extend the Gatekeeper concept not only to interact with agents and decide if any of them can fulfill a role, but also to reason about role dependencies, gather subsequent information, and group agents into teams when needed to satisfy a role. We further extend the team-building abilities by adding a Monitor agent which monitors team member behavior and alerts the Gatekeeper of agent failures which may require member reassignments. Finally, we design abstract protocols for the interactions of Gatekeeper, Monitor, and other agents in any environment.

1. INTRODUCTION

Imagine a team of agents scattered in an environment – a set of Search and Rescue robots, for example – that each want to be helpful and do a particular job such as put out fires, rescue people, and clear blockades from roads. In the interest of scalable architectures which allow for failure of some agents, we try to veer away from set hierarchies of leadership and make small teams, instead. Because each agent may have different abilities or may change abilities throughout its lifetime (for example, if a small portion of a robot agent's hardware broke but most of the agent is fully functional), we also want to dynamically group teams based on ability. Now, how can we best make sure that the job gets done with this heterogeneous combination of agents? More specifically, if none (or few) of the agents is actually able to accomplish the task it wishes to do, how can we make sure the task gets done?

To make sure the task is finished with whatever partially-capable agents we have, we propose building an agent team based on agent abilities. Each agent must be self-aware, in that it can reason about its own abilities, such as the ability to go to a place or pick up an object [1]. Any team capable of accomplishing the task must collectively have the required capabilities for doing the task, and some team members may have additional requirements needed to coordinate with other agents in the team.

To properly form the team, we extend the Gatekeeper concept [1, 2] – an agent which identifies agents in an environment, evaluates the agent's suitability for some particular role, and assigns that role to the agent. The expanded Gatekeeper assigns multiple agents to the same role by assigning agents to each subrole in the main role. This necessitates all agents being able to communicate with the Gatekeeper to request a role and discuss abilities, and possibly coordinate with future teammates. The Gatekeeper must then be able to reason about agent capabilites, agent ability to satisfy the role, and collective agent abilities that might satisfy the role. The Gatekeeper will need a reasoning scheme, an interaction protocol, and the discretion to assign agents to a team as needed.

Because the Gatekeeper agent requires a very different knowledge set than workers in the environment, we also consider what happens if the Gatekeeper agent needs to leave the environment for some time – for example, to work with a different set of agents and structure other teams. In case agents become disabled while the Gatekeeper is gone, we also implement a local agent called the Monitor, who monitors all team agents and is equipped with the ability to notify the Gatekeeper of significant changes in a team member's ability or total agent failure. The Gatekeeper, thus armed with new knowledge, can determine if the current team is still viable and assign new teammates or recognize that the role's goal(s) will be left unfinished.

Our contributions are the abstract protocols and design specification used for any Gatekeeper, Monitor, and agent set within an environment. These abstracted, environment-independent rules arise from the related work described in Section 2 and the formal problem description in Section 3. We outline the design for the Gatekeeper and Monitor agents in Section 4, implement an example in the BlocksWorld for Teams test bed in Section 5, and explain results in Section 6. Concluding remarks and future work is given in Section 7.

2. RELATED WORK

We build on previous literature in organization modelling, teamwork, and shared mental models. We use an *organizational modelling language* to specify roles, subroles, the organizational structure, and environment norms [3]. The organizational structure allows us to abstract from the actual agents that will eventually function in the system, which means the organization must have rules that an entering agent (of varying design, purpose, or designer) must accept and roles that an agent can take. Reasoning agents that can interpret the organizational constraints and adopt roles are called *organization-aware agents*, and we require all of our agents to have a firm grasp of the organizational norms so that they can communicate and, later, coordinate. The roles in the environment may also be described as adopting a *goal*; the goal of an agent is to accomplish some task. Any task must be accomplished only by an agent with the appropriate *capability* [4], in which the agent has the ability to act rationally towards acheiving a goal and the oppor-

Appears in: *Proceedings of the Workshop on Multiagent Interaction Networks (MAIN 2013), held in conjunction with AAMAS 2013*, May 7, 2013, Saint Paul, Minnesota, USA.

tunity (at some point).

We desire not only an abstract organizational structure, but an idea of the team structure and each individual agent's reasoning model. The team work literature is a well-known force for accomplishing tasks better or faster, and organizing the agents into teams adds the cooperative element that lets an agent request roles it may not be entirely suited for. Research has inspired self-organized and distributed teams, human-inspired robot teams, and human teams modelled with robots [5, 6, 7, 8, 9, 10]. Previous work also shows that a smoothly-functioning team can structure team information and action processes in the so-called "team mental model," which includes communication, understanding, shared goals, technological abilities, intention and need prediction [11, 12, 13]. A team mental model is the particular schema used by each agent in a team, whether the agent is human or robot; as an agent experiences the world, its personal mental model becomes populated with data or beliefs. The better the team mental model, the more easily agents can understand each others' actions or interests, enabling the agents to interact and cooperate more smoothly.

An environment may contain heterogeneous agents for two important reasons: the agents were designed differently to begin with, or the agents became different through various on-the-fly adjustments or malfunctions. Using organization-aware agents allows us to ignore the reasons behind such heterogeneity; however, this means the team-organizing Gatekeeper must be able to request and receive information on applying agent's capabilities. Requiring agents able to self-reason as in [1] allows agents to understand their own capabilities and reply honestly when queried.

3. PROBLEM DESCRIPTION

To summarize the problem, we have an environment with one or more heterogenous, autonomous but reasoning robots with or without physical bodies. Each robot, or agent, must perform a task in an environment, which it may not be able to do; we seek a way to formally assign these agents to a team to perform the task. Each agent can reason about itself enough to determine what it can or cannot do. Such capabilities may be based in the physical (move forward, pick up item) or mental (reason about what task must be done next). An agent's capabilities may be stored internally in the format used in [1]; e.g. the tuple *capability(ableTo(parameters))*. For example, consider an agent tasked with delivering car parts to stationary robots inside an automotive manufacturing plant. This agent has a body for moving around, finding and delivering parts to the proper robot. This agent would at least have abilities:

- cap(ableToDo(goTo,[place]))
- cap(ableToReason(nextBox, []))

Additionally, actions or events in the environment may contain dependencies. For example, if the agent must move a particular box of parts from the factory floor to a table near a specific welding robot, it must first determine the correct box, find it, pick it up, then move with the box in hand to the table; this particular task may be called box-moving. The agent which performs it takes on the role of *boxMover*, and the activity may be split into separate tasks, which could be thought of as subroles:

- role(boxMover)
- subrole(boxMover, determineTargetBox)
- subrole(boxMover, searcher)
- subrole(boxMover, deliverer)

To determine the target box, the agent requires some prior knowledge – perhaps which box of parts in a sequence of parts must be delivered. Knowing which box is required allows the agent to look for it; finding the box's location allows the agent to pick it up and deliver it. Thus, there is a flow of information from each task to the next. If different agents claim each subroles, they must coordinate and share information (namely, which box must be found and where it is). A required capability to deliver the proper box may be encoded as *reqCap(deliverer, ableToDo(goToBox, [boxID]))* and the information flow from determining the target box (using an index into a box sequence) to the next box may be encoded as:

- roleLink(determineTargetBox, [(sequenceIndex,[index]), (sequence, [])], nextBox)
- roleLink(searcher, nextBox, locBox)

The interesting problem arises if an agent cannot do everything the *boxMover* role requires; our solution is to create an organized team of coordinated agents, and we implement the Gatekeeper to assign and organize this team. A second and simpler agent, the Monitor, may be useful to keep an eye on the newly formed team. In the next section we describe the interaction protocols for the agent-to-agent communications needed to learn about agent capabilities and design and assign a team.

A single agent performing all the tasks required in the environment may not need to know about the information flow given above. However, for a Gatekeeper to determine if a combination of agents can satisfy the role, it must know the capabilities required for each role and whether an agent assigned to one subrole is able to send information needed by a different agent taking another subrole, and that the second agent is able to receive it – thus, the Gatekeeper must know the *reqCap* and *roleLink* knowledge given above. This information flow also cues agents to target other agents for certain data; the Gatekeeper may alert one agent to send information output to another agent(s) based on the information result of the first agent and required input knowledge of the second agent.

4. PROTOCOL DESIGN

The agents in this organization require at least three basic conversations, for which we design three interaction protocols. First, the Gatekeeper interacts with agents to determine if teams are possible or required; second, the Gatekeeper sends those details to the Monitor and receives updates during task execution; third, the Monitor agent monitors the assigned agent teams for partial or total failure. Because hostile environments may totally disable agents, the Monitor periodically polls team members; non-responsiveness is considered total agent failure. Finally, the assigned Agents do the task while possibly interacting with each other. The Gatekeeper knows before assigning teams whether additional communication between team members is required, and that communication will vary by environment; therefore, we do not design a separate protocol for those smaller member-to-member interactions. Instead, note that the Gatekeeper can alert each agent which other agent requires as input the information output it generates from an assigned subrole; this means an agent can communicate information directly to the agent that requires it.

After receiving updates from the Monitor, if the Gatekeeper determines the task can still be accomplished with the current set of Agents, nothing need occur. If the team requires a member addition and a suitable agent already exists in the environment, the Gatekeeper notifies the Monitor, who relays that new role assignment to the new teammate. If all else fails, the agent team will continue to execute with the assumption that it is better to accomplish part of the task than stop completely (alternatives exist, varying by environment – see comments in Section 7). The Gatekeeper, however, is aware at all times that the task can (no longer) be completed.

In this way, the Gatekeeper (a robust, role-knowledgeable agent) can move about the environment and accomplish other task assignments, leaving only the fairly simple Monitor agent to keep track of its agent team in the environment. This enables the Gatekeeper agent to interact with a small subset of agents (any Monitors within the environment) during execution, and allows us to equip only the Monitor agent with any robust hardware or software necessary to interact with the Gatekeeper, possible at long distances (or around other difficulties) within the environment. This keeps the number of Gatekeeper agents low, keeps local agent abilities simpler (no long-distance communication required), and adds only very small costs associated with the Monitor agents. The Gatekeeper may also press a local agent into service as the Monitor, in which case we do not require any more agents than are already present.

We format the communication as in [1], taking or slightly modifying several formalizations from that work. First, agent capabilities are defined as follows: actions (*ableToDo*, specific actions an agent can take, such as pick up an item), percepts (*ableToPerceive*, information sent from the environment and received by some sensing from the agent and possibly stored in a knowledge base), and communication (*ableToComm*, a message to or from an agent). As in the same work, a goal is the state an agent wants to reach; a goal can be adopted and reached (and thereafter discarded). Internally, an agent's capabilities are stored as *cap(<Cap>)*, and having an ability also means an agent knows it has that ability (by virtue of reasoning about the agent's own code). We also use the following message notation from [14]: "!" means an imperative (request), "?" means a question, and ":" means information. In the interaction protocols, some messages are optional or are alternatives; therefore, Figures 1, 2, and 3 use arrows with solid lines to indicate communications that always occur but dotted lines to indicate communication may not occur. If two alternative responses may occur, both are listed with dotted lines.

The interaction protocol between Gatekeeper and Applicant agents in Figure 1 requires the most dialogue between agents, as well as several points of internal Gatekeeper reasoning. After agents request a role, the Gatekeeper must determine if this role needs to be filled, asks if the Applicant agent has required capabilities, and makes assignments. Assignments are based on the capability information as well as an internal calculation that determines if, with the current set of agents, the task can be satisfied at all, potentially by filling subroles of the main role. If an Applicant is assigned a role, the Gatekeeper notifies it of the assignment.

The initial contact between agent and Gatekeeper is the request to become a *role-enacting agent*, or *rea*, for some role *Role*: `!rea (agt_x, Role)`. Here, the *x* indicates only that any number of agents may enter into discussion with the Gatekeeper at a time; future work will focus on the maximum number of agents can coordinate with a Gatekeeper at one time. If the Gatekeeper has already filled the role, it may immediately deny the agent's request with a rejection notice. Alternatively, if the Gatekeeper knows about another role `role(NewRole)` that the agent may want to take, it may suggest this role, after which the agent can stop responding or send a new `!rea` message. If the Gatekeeper considers allowing the agent to take the role, then for each initially known required capability `reqCap(Role, Cap)`, the Gatekeeper asks if the agent has the capability with `?cap(agt_x, Cap)`, which the agent must confirm or deny. If the agent has all required capabilities, the Gatekeeper may assign the role immediately and confirm role acceptance. If the agent does *not* have all the required capabilities, the Gatekeeper must wait and communicate with other agents.

If the Gatekeeper determines a set of agents can perform the role (by assigning agents to one or more subroles) with possible extra coordination requirements (as determined in Section 4.1), the Gatekeeper must confirm those extra capabilities by repeating the capability-affirming conversation. Extra coordination may be any ability such as explicit communication, `ableToComm`, or an action, `ableToDo`. A coordination capability is no different than any other capability to the agent; only the Gatekeeper considers this capability as required for coordination. At the same time, if an agent must communicate information to another agent, the Gatekeeper can leverage its knowledge of role input and output information to notify an agent which other agent(s) should be on the receiving end of that in-between information: e.g., by sending *a1* the message `sendTo(a2)`. This allows the sending agent to reduce extraneous message passing by sending messages only to the appropriate agent(s) instead of the entire group. After this capability discussion, the Gatekeeper may still reject an agent (if the role is satisfied with other team members) or it may assign the agent to the role with teammates `:rea(agt_x, SRole), teammates(<agt_x0 ...)`

The Monitor interaction protocol with Agents, shown in Figure 3, is straight-forward and simple by comparison – Agents are responsible for notifying the Monitor if they have lost any capabilities with a `:lost(Cap)` message. If the capability was on the enforcement list sent to the Monitor by the Gatekeeper, the Monitor will pass on that message. As mentioned earlier, we also require that the Monitor pro-actively and periodically communicate with each teammate to verify the agent is still functional. This message, `?stillAlive(Time)` is sent at some interval, which may be any unit such as minutes, hours, or after small task completions.

The Monitor interacts with the Gatekeeper, shown in Figure 2, and the Gatekeeper uses the new information to make any new team, role, or subrole assignments. The Gatekeeper requires the Monitor keep track of specific agents and capabilities with `:enforce (agt_x, Cap)`, meaning the Monitor should notify the Gatekeeper when `agt_x` loses capability `Cap`.

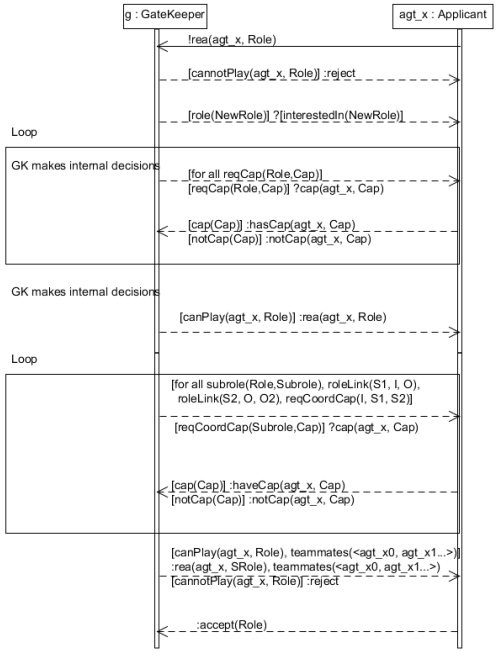

Figure 1: The Gatekeeper / Applicant interaction protocol.

We now define the high-level knowledge requirement for any

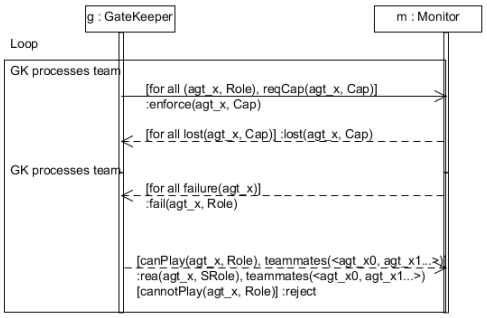

Figure 2: The Gatekeeper / Monitor interaction protocol.

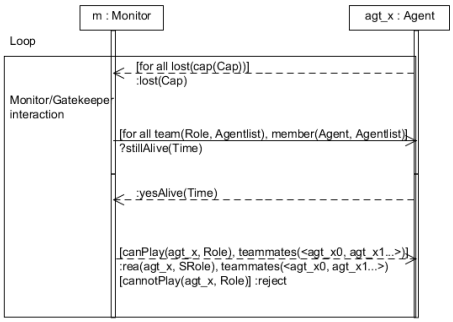

Figure 3: The Monitor / Agent interaction protocol.

agent who might want to be in a team:

- Communication. An agent must determine what to communicate about, given the capability division (from Gatekeeper).
- Reasoning. An agent must be able to reason about capabilities in relation to role-capabilities and agent interaction in relation to role-capabilities.
- Organization formation.
  - Done before task execution, by the Gatekeeper:
    - Knowledge of what capabilities agents have
    - Knowledge of dependencies between capabilities: those which come due to interaction between agents in different roles, and those which come from performance requirements for one agent from its role
  - Done during task execution, by each agent:
    - Knowledge of who is going to employ which capability (the internal 'organization' per team)
    - Ability and follow-through of communicating relevant information, based on dependencies, 'task division,' and current state
    - Knowledge of the Monitor, such that agents can monitor performance and reorganize if needed; each Agent must take instructions from the Monitor

4.1 Gatekeeper

Because team members may have to coordinate, the Gatekeeper must be aware of roles, subroles that compose the role, and any dependencies between subroles. If two different Applicants have the required capabilities for two different subroles, the agents may have to coordinate action or information from one subrole to the other because of a dependency (for example, send each other information about the contents of a room or building). To that end, the Gatekeeper must be able to reason about information dependencies between the two agents and verify they have the capabilities needed to pass along the required information or perform the additional actions. This verification must happen after realizing the potential subrole assignments but before finalizing any assignments. For added robustness, the Gatekeeper should first verify if role dependency information might instead be gathered from the environment, using perception, to minimize additional agent-to-agent communication. The Gatekeeper will not check references or verify capabilities. We assume each Applicant has accurate knowledge about his capabilities and does not lie about his wishes to play a role or his capabilities. We assume that if an agent requests a role from the Gatekeeper, it is willing to follow the instructions it receives, and continues to communicate and act in the best interests of the assigned role.

To properly determine if agents can fulfill a role, the Gatekeeper requires some initial structured information about the roles. All agent roles should contain input, actions, and output. A reasonable form would be:

- required input: I_1
- required actions: $[action_1, action_2, \dots]$
- delivered results: I_2, state-change.

Such definitions should ignore matters of internal policy; e.g., inputs (the collection of all information required to execute actions), actions (tasks or processes done by the agent to or in the environment), and results (information creation or changes within the environment caused by the agent) should ignore which actions are alternatives for unexpected behaviors or strategies for accomplishing tasks. We simply desire all inputs, all actions, and all results of actions (or outputs). In addition, note that the required input for some actions may be the expected output or result of another action. Thus the required actions speak to the needed capabilities for the agent performing the role, especially if extra coordination will be needed. We may also implement the role as a series of subroles, possibly based on a one-to-one or one-to-many association with the actions required of the role.

Consider a Gatekeeper that is aware of agent abilities and potential subrole assignments `potAsgn(agent, subrole)` for each agent (an agent with all required subrole capabilities could perform the subrole). Now we have the input of potential assignments – for sake of brevity, given as three subroles of the role, of which one agent *a1* satisfies the first and second, and another agent *a2* satisfies the third. The Gatekeeper's mental model of the world at this point should look like:

Input:

- Agents *a1* and *a2* have some list cap$[c_1, c_2, ...c_n]$, where c is a capability
- potAsgn(*a1*, subrole1)
- potAsgn(*a1*, subrole2)
- potAsgn(*a2*, subrole3)
- role-description(role, specification)

Actions:

- team-formation(*a1*, *a2*): because cap(*a1*) $\cup$ cap(*a2*) is a superset of goals(Goal)
- role-creation: role(newrole1), role(newrole2) – name of new roles is arbitrary but sensible, taken from knowledge of agents' capability(ies)
- role-assignment: assigned(*a1*, newrole1), assigned(*a2*, newrole2)

- for all goals in Goal, match input to results – determine if an information flow exists such that every subtask can receive the required input so that the over-all goal is accomplished; this includes reasoning that multiple agents may require additional dependencies, such as extra communication, to satisfy the goal
  - **if** input(G2, I) && result(G1, I) **then** poss-dep-on(G1, G2)
  - **if** poss-dep-on(G1, G2) && role-assignment(*a1*, G1) && role-assignment(*a2*, G2) && $a1 \neq a2$ **then** sharing-req(output(G1), *a2*, *a1*) && req-cap(*a1*, send-comm(I)) && req-cap(*a2*, receive-comm(I)))
  - Here, goal $G1$ outputs information I that is the same information required for input to $G2$. Thus, $G2$ has a possible dependency on $G1$, which requires any role assignments of, say, *a1* to $G1$ and *a2* to $G2$, that *a1* is capable of communicating I and that *a2* is capable of receiving that communication.

Results:

- Send messages to agents, *a1* (Agent1) and *a2* (Agent2), asking whether they have the required communication capabilities. If they do, then actualize the role assignments.
- Gatekeeper informs agents about role assignments; if the agents accept, the role may be finalized.
- Gatekeeper informs the Monitor of all agent-capability pairs that are required in this team.
- If Gatekeeper receives information about agent partial or total failure from the Monitor, it may recompute the information flow and send any new role assignments to the Monitor.

4.2 Monitor

The Monitor agent is much simpler – given a series of capabilities and agents to enforce, the Monitor listens for any reported failures. In volatile environments, we also want to ensure agents are still functioning, and the Monitor must periodically poll each team member.

Input:

- enforce(*a1*, cap(Cap))
- enforce(*a1*, cap(Cap2))
- enforce(*a2*, cap(Cap3)) ...

Actions:

- notification: if believe `(lost(a1, cap(Cap)))`, notify Gatekeeper of loss, `lost(a1, cap(Cap))`
- notification: if believe `failure(a1)`, notify Gatekeeper of total failure, `failure(a1)`
- notification: if believe `(newAssignment(aX))` and `enforce(aX, c1)`, `enforce(aX, c2)` ..., notify team of new assignments (may include reassignment of existing team members).

Results:

- The notification may trigger a response from the Gatekeeper (a new enforcement message), in which case the Monitor adds that input.

In this way, the Monitor is removed from the decision process, and only needs to be as complex as is required for locally communicating with the nearby agents and communication with the Gatekeeper (possibly at long distances).

5. BW4T IMPLEMENTATION

BlocksWorld for Teams (BW4T) from Delft University of Technology [15] adds complexity to the standard BlocksWorld problem by expanding the environment from a simple agent and tabletop to a map with multiple rooms, limiting visibility of block locations, and varied numbers of robotic or human players on a map. The map contains rooms, hallways, zero or more blocks of various colors inside a room, a non-zero number of agents, and a target sequence of blocks to find throughout the rooms and deliver to a target location.

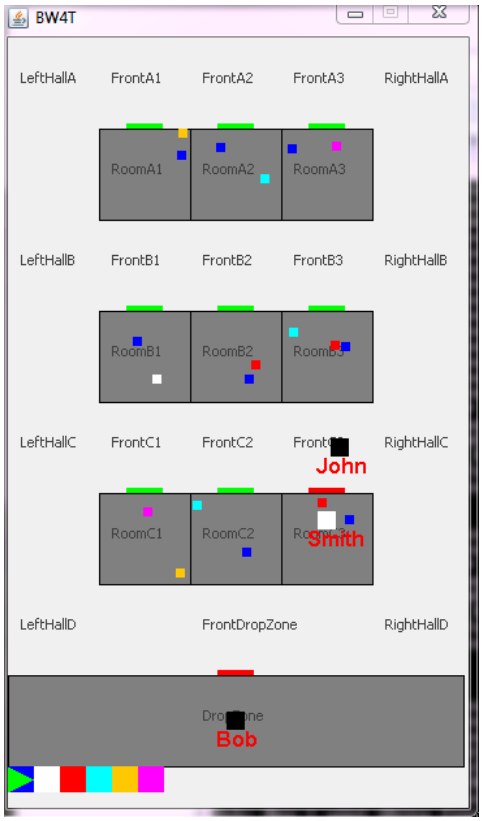

Figure 4: BlocksWorld For Teams (BW4T)

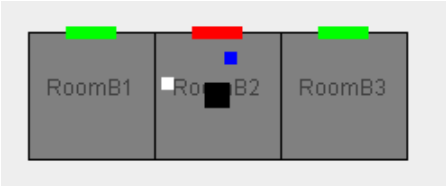

Figure 5: An agent (black) in a room; it can see objects (a white box and a blue box) in this room, see that its current room is occupied, and see that neighboring rooms are unoccupied.

The agent's goal is to deliver a sequence of blocks in the assigned order. To accomplish the goal, an agent must explore the rooms, find the blocks, and deliver them in the required order to the DropZone. The complete agent which does this is a BlocksWorld for Teams Player, or BW4T Player, and we will expand this specification in the next section.

BW4T allows up to one agent in a room at a time, any number of agents in a hallway, and any goal color sequence of blocks for the

agents to find and deposit in the DropZone on the south end of the map. Boxes that are dropped outside rooms and outside the DropZone disappear. Agents cannot see other agents, but they can hear other agents' messages, and they can see when a room is occupied.

Figure 4 shows nine rooms, RoomC1 – C3, RoomB1 – B3, and RoomA1 – A3, as well as the DropZone. Underneath the DropZone, the user sees the required sequence of blocks to be delivered, as well as which blocks have been delivered. In this case the sequence is blue, white, red, cyan, orange, and magenta; the blue block has been delivered (indicated by a green triangle). Agent Smith has picked up a white block in RoomC3, indicated by the agent's change in color from black to white. An agent cannot see blocks until it moves inside of the room, as shown in the cross section of the environment in Figure 5.

If the agents have memory, they can remember what blocks they have seen and where, enabling them to retrieve specific colored blocks without repeating the exploration phase. Obviously when multiple agents run around looking for blocks and do not share all information they have about the blocks found, memory has a limited value. Another agent could go into a room and pick up a block, rendering the information stored in an agent's memory out of date. Agents are able to communicate with each other through predefined questions, answers, and statements, such as "Who is in RoomB2?", "I am in RoomB2," and "Someone, we need a Blue block." [15]. Humans can participate in the BW4T world by controlling one agent, which has the same environmental constraints as any other agent; the human-controlled agent will communicate messages the same way.

An agent starts without knowing where any blocks are, so it must choose a room to look in for the next color block. If it finds the correct color, it can pick up the block and carry it to some location (either the DropZone to deposit, or to a nearby room for storage). Then the agent decides again – is the task finished? Should it explore another room? Does it know where the next color is? When other agents communicate information, an agent's job is easier because it no longer needs to explore some rooms (if it just heard what blocks are in a particular room) or because it no longer needs to get every block in the sequence (if another agent has just informed the team it will pick up the next block or has delivered a block).

5.1 Gatekeeper and Monitor implementation

We now demonstrate the Gatekeeper and Monitor agents in the BW4T environment. The previous two sections give a generic version of the specifications required; we will now outfit the specification with BW4T specific roles, subroles, and linking information. Because the goal is to solve the BW4T task by finding and delivering blocks in a specific order to a predefined location, we first require a BW4T Player specification for the Gatekeeper which contains role and subrole information, required capabilities, and role-checking rules. We are conerned with demonstrating how the BW4T Player task can be accomplished when no single agent has all the abilities required to find and deliver a block; therefore, we implement the Gatekeeper, Monitor, and protocols with two different agent types. One agent type can only reason about the next block and identify blocks; it has no 'picking-up' abilities. The second agent type can only pick up a block and deliver the block; it has no abilities to reason out the target color or to identify block colors.

First, we populate some of the mental model of the Gatekeeper and agent knowledge with BW4T specific information, such as the goal and subgoals. Next, we discuss the actual implementation.

The BW4T-player role can be described by three subroles:

- determineTargetColor:
  - input: capabilities to perceive task sequence, perceive current index in task sequence
  - actions: calculate from index and goal sequence
  - result: target-color (next color in sequence)
- find:
  - input: target-color (next color in sequence)
  - actions: goTo roomR; if unknown what blocks are in room R; until color seen
  - result: [block(id, color, roomR)]
- deliver:
  - input: rooms that contain a block of the target-color
  - actions: select room R that has target-color
    - goTo room R
    - pickUp block with target-color
    - goTo dropzone
    - drop target-color
  - result: environmental trigger used by the determineTargetColor subrole (new index in task sequence)

The Gatekeeper mental model looks much like the generic version given in Section 4 – only we can change the Input section to include some specified list of abilities that result in, for example, agent *a1* being able to do the $determineTargetColor$ and $find$ roles, and agent *a2* being able to do the $deliver$ subrole).

The BW4T specification must include, for the purpose of team formation:

- input (determine-target-color, []), result(determine-target-color, info(next-target-color, (C)))
- input (find, [info(next-target-color(C))]), result([block(X, C, roomR)])
- input (deliver, [info(next-target-color(C)), [block(X, C, roomR)]]), result(deliver, state(done(C)))

We use this as input to the Actions section. Together with the Input of *a1* and *a2* abilities, we have new role-creation and role-assignment information:

- role-creation: role(determine-target-color-find), role(deliver)
- role-assignment: assigned(*a1*, determine-target-color-find), assigned(*a2*, deliver)
- for all goals in BW4T-player, match input to result – each role link is defined as "roleLink(subrole, input, output (results))" or as "roleLink(subrole, input)"
  - roleLink(determineTargetColor, [sequenceIndex, sequence], nextBlock).
  - roleLink(searcher, nextBlock, locBlock).
  - roleLink(deliverer, locBlock).
  - **if** links exist, roleLink(Subrole1, Input1, Output1), roleLink(Subrole2, Output1, _), **then** insert(reqCap(Subrole1, send-comm(I))), insert(reqCap(Subrole2, receive-comm(I))) or reqCap(determineTargetColor, receive-comm([sequenceIndex, sequence])), reqCap(determineTargetColor, send-comm(nextBlock))), reqCap(searcher, receive-comm(nextBlock)), reqCap(searcher, send-comm(locBlock)), reqCap(deliverer, receive-comm(locBlock)). (In this environment, a successful block delivery triggers an environmental *index* change, which *determineTargetColor* uses as input.)

5.2 Example code

We implemented the Gatekeeper, Monitor, and two agents in BW4T environment using the GOAL IDE which uses Prolog, and give a few code snippets here. For example, the role-subrole relation *subrole(bw4tPlayer, determineTargetColor)* input – actions – result information is captured by the Prolog clause *roleLink (determineTargetColor, [(sequenceIndex, [index]), (sequence, [blocks])], nextBlock)* The $sequence$ of $blocks$ tells the agent the sequence of required blocks; the current $sequenceIndex$ at value $index$ tells the agent which block in the sequence is next. Note that the $determineTargetColor$ subrole input – action – result sequence is just one input for the Gatekeeper. It must know required capabilities for each subrole; e.g., its knowledge base will also contain

- reqCap(determineTargetColor, ableToPerceive (sequenceIndex, [index]))
- reqCap(determineTargetColor, ableToPerceive (sequence, [blocks])
- reqCap(determineTargetColor, ableToReason (nextBlock, []))

The subroles $searcher$ and $deliverer$ have similar $subrole$ and $roleLink$ rules. As another example, take agents *a1* and *a2*. Some of *a1*'s capabilities are

- cap(ableToPerceive(sequenceIndex, [index]))
- cap(ableToDo(goTo,[place]))
- cap(ableToComm(send, [at, blockID, color, room]))
- cap(ableToReason(nextBlock, []))

A few of *a2*'s capabilities are

- cap(ableToDo(goToBlock,[blockID]))
- cap(ableToDo(pickUp,[]))
- cap(ableToComm(send, [at, blockID, color, room]))
- cap(ableToComm(receive, [locBlock, blockID, room]))

These two agents are affected by role links

- roleLink(searcher, nextBlock, locBlock)
- roleLink(deliverer, locBlock, na)

Initially there were no communication requirements (if an agent can perform all tasks alone, it does not need to talk to itself). However, the agents assigned to $searcher$ and $deliverer$ are different, which triggers additional requirements:

- reqCap(searcher,ableToComm(send,locBlock))
- reqCap(deliverer,ableToComm(receive,locBlock))

6. RESULTS

Using the protocols illustrated above, we performed several experiments to demonstrate the Gatekeeper, Monitor, and agent interactions:

- A Gatekeeper with a single agent able to accomplish all tasks
- A Gatekeeper with two agents which, when combined, can accomplish all tasks
- A Gatekeeper and Monitor with four agents (2 that can determine target color and search, and 2 that can deliver blocks) able to accomplish all tasks
- A Gatekeeper and Monitor with four agents (as above) able to accomplish all tasks, in which one agent fails in mid-task
- A Gatekeeper and Monitor with four agents able to accomplish all tasks, in which both agents able to search for colors fail in mid-task

Overall, the experiments showed that the Gatekeeper and Monitor were able to determine a potentially-successful team, monitor the capabilities of agents, and determine when a team would or would not be able to finish the goal.

A sample of the Monitor agent's beliefs after receiving Gatekeeper enforcement notices can be seen in Figure 6, and a sample of beliefs of an agent assigned to search and determine-target-color can be seen in Figure 7.

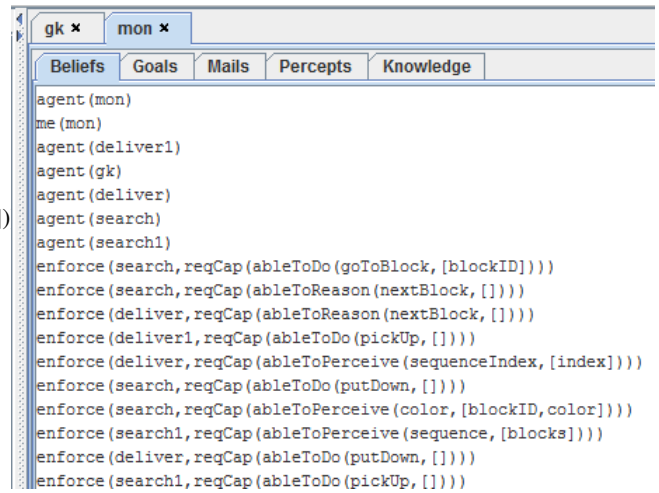

Figure 6: The Monitor agent's beliefs after receiving Gatekeeper messages.

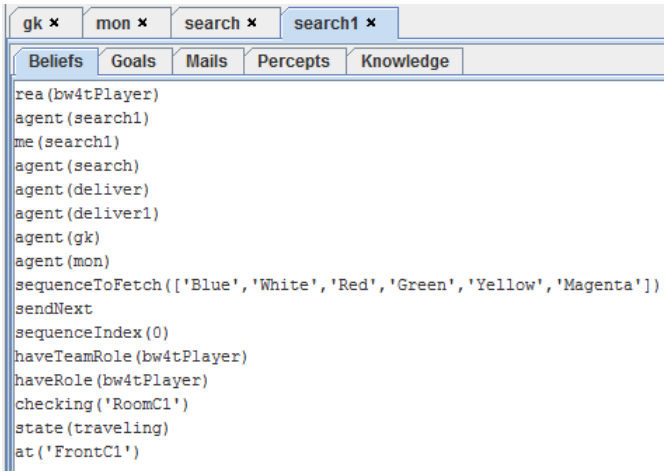

Figure 7: A Searcher's beliefs before exploring.

The two simple tests, a Gatekeeper with a single agent able to accomplish all tasks, and a Gatekeeper with two agents that accomplish all tasks, proved the basic concept that the Gatekeeper can analyze agent ability and assign teams. The next test, a Gatekeeper and Monitor with four agents, was also able to accomplish the BW4TPlayer role. For this test, note that we could approach the agent combination in several ways. First, we might have one determine-target-color agent, one searcher agent, and two deliverer agents. However, this means one agent would sit idle until the time came to announce the next color block to fetch, which is a waste of resources. Second, we could use two determine-target-color agents assigned to the searcher subrole, along with two deliver agents.

This means that the two agents who were able to determine-target-color and search were each assigned both roles. This is the method that we used. A third and simpler team structure could have only assigned one agent to the determine-target-color subrole and two agents to the searcher subrole (with two deliverer agents). A fourth method might be to have two pairs of searcher-deliverer agents, in which each agent searches out and delivers a subset of the target sequence, like the odd or even indices. As mentioned earlier, we are more concerned with initial interactions and targeting messages to agents that need specific input.

More interesting are the last two tests, in which agents break mid-task. The first, in which a Gatekeeper and Monitor with four agents (2 determine-target-color plus search, 2 deliverers) able to accomplish the BW4TPlayer role, with one agent that fails in mid-task, tested the Monitor's behavior. We were able to properly control the information (an agent capability failure) flow from Agent to Monitor to Gatekeeper. One of the agents with determine-target-color plus search was no longer able to identify blocks (analogous to camera failure in the field, for example) after the first block was delivered. That means that as soon as the failure occured, the agent was no longer able to assist in searching out blocks, because it could no longer identify block colors (which is how the required block sequence is defined). Because at least one agent was still able to perform each role (searching for blocks), the over all BW4T player goal was accomplished, and no further action was required.

The last test, in which a Gatekeeper and Monitor with four agents able to accomplish all tasks, in which both agents able to search for colors fail in mid-task, was the most illustrative. We accidentally discovered that since the searcher agents' failure occured after they had already explored most of the rooms, the deliverer agents were still able to finish the task. Because the agents were still able to find the blocks already known about and know the next block to fetch (because this ability was not impacted), the task ended in success. This occurred even though role requirements during run-time would have declared the agents unable to satisfy the role. Thus, in some cases, continuing to do the task even under a few agent failures can be a useful approach. There are some cases, however, when continuing to do a task even under partial system failure would be horrible – for example, during a surgery, a robot system is probably better off stopping the surgery completely than continuing to cut into a person even though the goal, installing a pace-maker, would not be satisfied.

7. CONCLUSIONS AND FUTURE WORK

Our results suggest the Gatekeeper and Monitor combination makes team formation more robust to agent failure, but more work remains to be done. After assigning the teams, the Gatekeeper agent should output the new role specifications for the new roles (in the previous examples, these would be the two new roles that accomplish the determine-target-color and search subroles, and the delivery subrole). During any particular sample environment run, the Gatekeeper should be able to store, print, or otherwise record the agent and role assignments as well as agent capabilities and failures. This would allow humans to determine what went well before and during the task execution.

Our results also suggest that we need to be more aware of when continuation in the face of agent failure is still helpful. There may be domains (robotic surgery) in which agents should stop in case of partial failures, but in others (humanitarian de-mining) agents should continue as long as possible. We also must add agents into the environment in the middle of the task, especially if one of the team members has become disabled and the new agent would be able to replace the broken agent. In this way we can fine-tune the Gatekeeper reassignment abilities to ensure goal success.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. NSF/IIS-1208413, NSF/IIS-1216287, and NSF/IIS-1216361. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] M. van Riemsdijk, V. Dignum, C. Jonker, and H. Aldewereld, "Programming role enactment through reflection," in *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2011 IEEE/WIC/ACM International Conference on*, vol. 2, aug. 2011, pp. 133 –140.

[2] H. Aldewereld, V. Dignum, C. Jonker, and M. van Riemsdijk, "Agreeing on role adoption in open organisations," *KI - Künstliche Intelligenz*, vol. 26, pp. 37–45, 2012, 10.1007/s13218-011-0152-5. [Online]. Available: http://dx.doi.org/10.1007/s13218-011-0152-5

[3] J. F. Hubner, J. S. Sichman, and O. Boissier, "Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels," *International Journal of Agent-Oriented Software Engineering*, vol. 1, no. 3, pp. 370–395, 2007.

[4] L. Padgham and P. Lambrix, "Formalisations of capabilities for BDI-agents," *Autonomous Agents and Multi-Agent Systems*, vol. 10, no. 3, pp. 249–271, 2005.

[5] X. Fan and J. Yen, "Modeling and simulating human teamwork behaviors using intelligent agents," *Physics of Life Reviews*, pp. 173 – 201, 2004.

[6] S. M. Fiore, E. Salas, H. M. Cuevas, and C. A. Bowers, "Distributed coordination space: toward a theory of distributed team process and performance," *Theoretical Issues in Ergonomics Science*, vol. 4, pp. 340–364, 2003.

[7] A. Howard, L. E. Parker, and G. S. Sukhatme, "Experiments with a large heterogeneous mobile robot team: Exploration, mapping, deployment and detection," *The Int'l Journal of Robotics Research*, vol. 25, no. 5-6, pp. 431–447, 2006.

[8] A. Rosenfeld, G. A. Kaminka, S. Kraus, and O. Shehory, "A study of mechanisms for improving robotic group performance," *AIJ*, vol. 172, no. 6–7, pp. 633–655, 2008.

[9] P. Salmon, N. Stanton, R. Houghton, L. Rafferty, G. Walker, D. Jenkins, and L. Wells, "Developing guidelines for distributed teamwork: Review of literature and the HFI DTC's distributed teamwork studies," *HFIDTC*, 2008.

[10] K. Sycara and G. Sukthankar, "Literature review of teamwork models," Robotics Institute, Carnegie Mellon University, Tech. Rep. CMU-RI-TR-06-50, November 2006.

[11] G. Dedre and A. L. Stevens, Eds., *Mental models*. Lawrence Erlbaum Associates, London, 1983.

[12] P. N. Johnson-Laird, *Mental models*. Harvard University Press, Cambridge, 1983.

[13] R. Klimoski and S. Mohammed, "Team mental model: Construct or metaphor?" *Journal of Management*, vol. 20:403, 1994.

[14] K. Hindriks and M. van Riemsdijk, "A computational semantics for communicating rational agents based on mental models," *Programming Multi-Agent Systems*, pp. 31–48, 2010.

[15] *BW4T2 Specification*, Delft University of Technology, 2011.