Evolving Social Commitment Based Protocols Using MAPC

Jason Heard
Computer Science and Information Systems
Mount Royal College
Calgary, Alberta, Canada
Email: jheard@mtroyal.ca

Rob Kremer
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
Email: rkremer@ucalgary.ca

Abstract—Currently, agent designers and programmers must work extensively to design, describe, test and implement the communication protocols used in multi agent systems. The design of the protocols is on an imperative basis. The potential exists for the designer and programmer to play a smaller role, though. This role would simply be to input a description of required activities, or to describe it declaratively. This paper describes MAPC (multi agent protocol creator) which makes it possible to evolve a protocol through the genetic programming of agent behaviours. These behaviours are capable of performing a communication-based task. Once the behaviours have been evolved, it is fairly simple to distil the behaviours into a social commitment based communication protocol.

Keywords—evolving communication, evolving protocols, social commitments, genetic programming, CASA

I. INTRODUCTION

Research on the creation of various types of protocols for use in multi agent systems is prolific. Protocols have been defined for auctions [1] and authentication [2] to cite two examples. In each of these cases, the protocols created have been the result of many hours of painstaking thought and skill in order to find a protocol that would meet several requirements. This paper describes MAPC (multi agent protocol creator), which was created to generate multi agent protocols.

To create a new protocol, the environment in which it acts must be clearly defined. For this work, the following method was used. The scenario in which the protocol must work is outlined. The scenario represents a generic situation, such as an auction house. In order to define this scenario, some of its constituents must be examined. In most protocols, there are classes of agents that all perform the same function within the protocol. To represent this, roles are used. A role represents one class of individuals within a scenario. In an auction house there are two roles: bidders and auctioneers. Formally, a role is defined as a set of role instances, where each role instance defines a specific set of attributes and capabilities for a given agent. A bidder with $100 and a desire for item #32 might be one instance of the bidder role. Just as roles are defined by a set of role instances, a scenario is formally defined as a set of scenario instances. These scenario instances are each a specific example of the general scenario. A scenario instance is defined by a set of role instances and a collection of global attributes. One instance of an auction house scenario might be three bidders (with their money and desires defined), one auctioneer (with the items to be sold described) and an amount of time allotted to complete all auctions.

Once a scenario has been outlined, a set of outcomes must be specified. The desired effects describe what the effects of the protocol should be. In an auction house, the desired effects might be that all items are sold within the allotted time, that all sales are fair (one and only one payment per item received), and that every bidder has a chance to bid (the bidder with the most money and desire receives the item). When, after the system has been executed, all of the outcomes are true, the given protocol is considered successful.

From the proceeding descriptions of the scenario and the desired effects MAPC produces a communications protocol. For this research, conversation policies have been chosen as the method to describe the resultant protocol. Conversation policies map actions of agents to the creation and deletion of social commitments [3]. When describing a communication protocol, conversation policies have several advantages over other models. Their greatest strength resides in their inherent social nature. Since they describe what agents are committed to do, they allow for an outside observer to determine if all agents are meeting the requirements of a protocol. In addition, they describe what agents must do (are committed to do) without describing how that would be accomplished. This allows a programmer implementing a conversation policy based protocol to use many different programming styles and systems.

The creation of the protocol involves two main steps. First a set of behaviours are evolved, one for each role in the scenario. Second, the evolved behaviours are distilled into a new protocol. The goal of the first step of evolving behaviours is to generate behaviours that could be plugged into any given scenario instance with the result that the desired effects will occur. This is done using genetic programming to create a set of conversation policies compatible with CASA, the agent architecture used in this research.

Once the behaviours have evolved which will produces the desired effects in any scenario, the second step of reducing these behaviours into a protocol takes place. Technically any set of conversation policies defines a protocol, but the desired result is a concise protocol. Since MAPC uses genetic
programming, the behaviours typically contain many unneeded policies, and even needed policies often contain many unneeded parts, referred to as introns [4]. In this final phase, a protocol is created by removing all of the superfluous portions of the conversation policies.

This research presents a simpler method for creating multi agent communications protocols as compared to traditional methods. It provides a system (MAPC) for evolving protocols for many different scenarios. This is unique in the field because it combines evolution with higher level protocols, and not just with communication actions (as in artificial life).

MAPC is also novel in its use of the social commitment paradigm. The system will use genetic programs to fill in the social commitment policies which define a agent communication protocol.

This paper is outlined as follows. Sec. II provides background information on the multi agent system (CASA) used in MAPC as well as a comprehensive overview of the evolutionary framework (ECJ) used for genetic programming. Sec. III provides the method for describing a scenario in MAPC. Sec. IV outlines the language used to describe the desired effects of a protocol. Sec. V describes the system used to evolve a behaviour which can meet the desired effects. Sec. VI gives an implemented scenario and a solution generated by MAPC. Sec. VII describes some related work. Sec. VIII offers some concluding remarks as well as some ideas for future work.

II. AGENTS, CASA, AND ECJ

This section provides important background information to aid the reader in understanding the evolutionary system outlined in this paper. This system will initially evolve behaviours for roles within a scenario through the use of genetic programming. In order to do this, it is helpful to define an agent and the concept of behaviour. Sec. II-A defines these terms. The multi agent system used as a basis for the evolutionary system is CASA (Collaborative Agent System Architecture). CASA provides a communication framework that makes use of social commitments and conversation policies. Sec. II-B describes the portions of the behaviour that will not evolve because they are part of every CASA agent. In addition, the conversation policies used in CASA are defined and described here. Finally, the genetic programming is performed using ECJ (Evolutionary Computation for Java), a flexible evolutionary framework. Sec. II-C describes ECJ and how it is used to evolve conversation policies.

A. Agent and Behaviour

For this paper, an agent (Ag) will be defined by a set of possible situations or environments (Sit), a set of possible actions (Act) and a set of possible internal data representations (Dat). This can be shown as a three-tuple.

\[ Ag = (Sit, Act, Dat) \]

The behaviour \( f_{Ag} \) of an agent is therefore a function which takes the current situation and the internal data of the agent and from that generates an action.

\[ f_{Ag} : Sit \times Dat \rightarrow Act \]

In the evolutionary system, a portion of the behaviour will be performed by the CASA system, and the remainder will be evolved.

B. CASA

MAPC makes use of CASA [5]. CASA is a multi agent system written in Java which is primarily communications-based. CASA agents make use of a standardized message format to communicate. Agents communicate within CASA using a common message format based on the FIPA message standard [6]. Each message contains several fields, including sender, receiver, language, contents, performative, and act. The sender and receiver define who has sent the message, and who should receive the message, respectively. These are given as URLDescriptors, which all agents are required to understand. The language and the contents fields of the message are specific to various situations and may not be understood by every agent. Finally, the performative and act fields define the context of the message within the system and the conversation in particular. These values are used by the conversation policies described below to map the message onto a set of social commitment operators.

CASA uses conversation policies to define its communications protocols. The conversation policies use the performative and act fields in each message to determine which conversation policy applies to a message. The performative gives the speech act of the message. The act, however, defines the overall topic of the conversation. For example, to request that another agent calculate a sum, an agent might send a message with a performative of “request” and an act of “add”. The other agent could then respond by agreeing to the request using the performative “agree”, declining the request using the performative “refuse”, or, in the case of misunderstanding or a scrambled message being received, with the performative “not-understood”. In each case, the response would have the same act, since that is what the agents are discussing.

The performatives are arranged in a subsumption lattice as shown in Fig. 1. In the diagram, the arrows indicate an “is a” relationship. So, the arrow from reply to inform and ack (short for acknowledge) indicates that a reply is also an inform and an ack. This means that a message that has a performative of reply triggers the reply, inform and ack conversation policies.

The performative lattice is given to all agents. In addition, there are behaviours built into CASA which will automatically process incoming and outgoing messages if given a set of conversation policies. The majority of the actions carried out by an agent are done through these conversation policies. So, the evolutionary step within the system evolves behaviours by evolving a set of conversation policies. Each conversation policy will be capable of creating and fulfilling social commitments. These social commitments will contain code

\(^1\) The format of the messages is either XML or KQML-like.
which will perform the majority of the actual work in the system. In addition to these conversation policies, each agent is capable of performing a set of actions on start-up and when it is idle (not processing incoming messages or outstanding social commitments). The evolving behaviours will therefore also include a section for the evolution of these areas of the agent.

C. ECJ

The genetic programming in MAPC is done using ECJ [7]. ECJ is an evolutionary framework written in Java that is highly flexible and extendable. In addition, ECJ provides genetic programming capabilities with a tree-based genome which are used in MAPC. Within ECJ, the evolutionary process is set up by creating (or using pre-made) classes responsible for various portions of the evolutionary process. The main classes that control the evolutionary process are shown in Table I. The evolution state holds all of the information about the current evolution and controls the overall evolutionary process. MAPC uses the SimpleEvolutionState class, which performs an evolution as follows:

1) Ask the initializer to create an initial population.
2) Ask the evaluator to evaluate the population (modifying the population’s fitnesses).
3) If an ideal individual was found, or the maximum number of generations has been reached, skip to 8.
4) Ask the exchanger to perform a pre-breeding exchange (which may change the population).
5) Ask the breeder to breed the next generation and replace the population with the result.
6) Ask the exchanger to perform a post-breeding exchange (which may change the population).
7) Increment the generation number and then go to 2.
8) Ask the finisher to finish up.

Exchanging is not performed in MAPC, so the SimpleExchange class is used, which performs no actions. Therefore steps 4 and 6 are essentially skipped. More details of the genetic programming used in MAPC will be described in Sec. V.

III. Scenario Description

The goal of MAPC (to create protocols) and the method used to work towards that goal (genetic programming) each require a different view of the situation in which they will be used. Protocols are typically designed to work in a variety of situations. Genetic programs must be tested in specific situations with specific criteria for success. In order to provide these specific situations, the concepts of scenarios and roles were defined for use in MAPC. The specific scenario instances contained within a scenario are used during genetic programming to provide specific environments in which to test and evaluate the potential behaviours, which will control each of the role instances in the scenario instance. The evolved behaviours should not just be suitable for specific agents, but should work in a variety of similar situations. The concept of roles is meant to capture these groups or situations within a protocol. This section will outline how a scenario is defined.

To define a scenario in MAPC, two things must be given: the scenario name and descriptions for all of the roles in the scenario. In addition, any scenario may also have scenario variables and/or scenario actions. The scenario name is used in the output of MAPC and also aids in discourse. The method for describing the roles is more complex and will be covered below. Scenario variables represent information that will be set for any particular scenario instance that is not specific to one role or role instance.\(^2\) Typically, scenario variables do not include values that are constant for the entire scenario. Scenario actions, on the other hand, describe domain-specific actions that all agents within any scenario may perform.

For each role in a scenario, a role name and the possible number of role instances in a particular scenario instance must be provided. In addition, any role may have role instance variables and role-specific actions. The role name is used to refer to the role when discussing desired effects, when debugging, and in the final output. The number of instances of a role is used when a scenario instance must be generated. The role instance variables represent information that can be

\(^2\)Scenario variables vary within the scenario, but within any particular scenario instance they are constants and cannot change.
different for each instance of a role in the scenario. Role-specific actions describe the domain-specific actions that are appropriate for agents in a given role.

The following simplistic example illustrates the information required to define a scenario. A MAS designer is creating a system with two types of agents: paint agents and finder agents. Each paint agent corresponds to one and only one colour. The finder agents must determine if there is a paint agent available that corresponds to a specific colour. This must be determined through communication, which requires a protocol. Assuming the designer is using MAPC to aid in the creation of this communications protocol, the scenario needs to be defined. This scenario shall be called “ Colour Query”. To represent the colour that must be found, each scenario instance has a designated colour (grey or white). To represent this, there is a scenario variable called environmentColour that can be either grey or white. The two roles in this scenario are “paint” and “finder”. To keep the example simple, there is always one paint agent and there can be two or three finder agents. The paint agent can also be grey or white—this is a role instance variable (paintColour) for the paint role. Finally, the finder agents may decide if the environment colour can be found in a paint agent or not. This is done using an action available to the finder agents called “found”. The found action requires a single boolean argument (whether the colour was found or not). This scenario is represented visually in Fig. 2. In the figure, there are eight scenario instances, each in a separate box. Within the scenario instances, the role instances (agents) are shown as circles with a cloud of personal data and capabilities. The finders are shown as double-lined clouds and the colour agents are shown as single-lined clouds. The environmentColour variable is illustrated with the background colour of the scenario instance, and the paintColour variable of the colour agent in each scenario is illustrated with the colour of that agent’s cloud. So, in the top-right scenario instance, the environmentColour variable is set to white, the colour agent’s paintColour variable is set to grey, and there are three finder agents. As is shown, each scenario instance defines the values for all scenario and role instance variables.

To simplify the process of describing scenarios, a scenario table format has been created. In a scenario table, there are two or more sections separated by double horizontal lines. The first section is always the scenario name. The next two sections are optional and give scenario variables and then scenario actions for the scenario, if any. The remaining section(s) each describe a single role, including the number of agents of that role in any scenario instance, role instance variables (if any) and role-specific actions (if any). The Colour Query scenario is described in Table II.

IV. DESCRIBING THE DESIRED EFFECTS

Every multi agent protocol is designed to meet some need of a designer. In order to automatically create a protocol in MAPC, designer needs must be codified so that the evolutionary process can evaluate individual behaviours. In MAPC, this is done by programming one or more fitness functions. These fitness functions are written in Java and return a float value greater than or equal to 0. They generate standardized fitness values; the most fit individual will have a fitness value of zero, and less fit individuals will have a fitness greater than zero.

To calculate the fitness for a given execution, the fitness functions need several pieces of information. As the system executes, scenario variables and role instance variables (whose values are defined by the scenario instance) can be read by the agents. After execution, these values are also available to the fitness functions. Each role instance (or agent) within the execution essentially creates a sequence of messages and actions. As described above, the actions may have parameters which describe how they were performed. In the case of the Colour Query scenario, the finder agents can perform a “found” action. Nothing in the scenario description restricts a finder agent from contradicting itself (performing a found (true) action and also a found (false) action). The fitness functions can iterate through the list of messages (individually or grouped into conversations) or through the list of actions, and therefore have access to the messages and actions performed by each agent. In addition to variables, actions, and messages from the execution, fitness functions also have access to the trees (one for each role) representing the behaviour of each role.

To illustrate the creation of fitness functions for MAPC,
consider again the Colour Query scenario. In the Colour Query scenario, the desired effects are that:

1) all finder agents make some determination about whether or not the colour is found,
2) no finder agents contradict themselves about whether or not the colour is found, and
3) the finder agents all pick the correct output (true if the paint agent’s colour matches the environment colour; false otherwise).

Each of these effects are tested in separate fitness functions. The first fitness function is given in Fig. 3. Each of the other desired effects are computed in a similar manner. The combined fitness for the execution is then the sum of the three fitness functions.

V. EVOLUTIONARY PROCESS

Once the scenario and desired effects have been outlined, MAPC can be used to create a new protocol. This is done using genetic programming. Genetic programming (GP) was initially introduced by Koza in 1992 as a method for automatically programming a computer [8]. Genetic programming uses genetic algorithms to program a computer. A programmer must simply define the goal of the program, and the genetic programming system will write the program to reach that goal.\(^3\)

A. Genome Representation

The formats of a genetic program and a traditional program are significantly different. Genetic programming works best when the program format is well suited to the crossover and mutation operators of the genetic algorithm. Koza originally introduced a tree-based programming format that accommodates these needs [8].

Tree-based genomes make use of a tree structure to represent a genetic program. They were originally used to mimic the parse trees that are used to represent programs in compilers. Each operation and constant in the genetic program is a node in a tree. Functions, such as the addition function, take the values of their children as their inputs and evaluate to a value. Terminals, such as the numeric values 1 and 2, act as the leaves in the tree.

The advantage of tree-based genomes is that as long as there are the correct number of children for each node, a program that contains syntax errors cannot be represented. MAPC also makes use of an important improvement on tree-based genomes: strongly typed genetic programming [9]. This means that genetic operators on these trees need only ensure that the correct number and types of children exist for each parent, and not that the correct syntax is followed. In addition, unless global memory is built into the functions and terminals, each node in the tree only has access to the result(s) of its children, if any. This makes the flow of information from the leaves to the root fairly straightforward. This can aid in understanding how a genetic program executes.

To construct and modify the tree-based individuals, all of the possible functions and terminals must be defined. The primary functions and terminals are as follows. (Note that the term argument will be used to indicate the inputs to the functions and not the alternative tree-based child.):

- The agent function is always the root of every behaviour. It has three arguments. The first is a section of start-up code that will be executed when the agent first starts. This is expected to initialize communications. The second is a section of code that will be executed when the agent is idle. This can be used for additional processing as the agent runs. The third argument is a policy that defines how this agent responds to one type of message that is sent or received. The structure of these policies is described below. Multiple policies may be attached to an agent by using the policy-sequence function.
- The policy function defines a single policy for an agent. Its first two arguments define which messages trigger this policy. The first argument is the performative and the second is the act. The third argument is the social commitment operator (rule) that will be executed when this policy is triggered by a message. Multiple operators may be attached to a single policy by using the rule-sequence function.
- The add-commitment function creates a new commitment for this agent. The first argument is the debtor for the new commitment, and the second argument is the creditor. The commitment has the performative and act indicated by the next two arguments. Finally, the fifth argument represents the action that the agent will perform when it decides to execute this commitment.

\(^3\)This is in contrast with the traditional method of programming, in which a programmer write a program that meets the goal.
The fulfill-commitment function fulfills all commitments with the given debtor, creditor, performative, and act.

The msg.performative terminal, along with msg.act, msg.sender, msg.receiver, and msg.contents extracts the indicated portion from the relevant message. When one of these terminals is inside the code of a policy, but not in a social commitment action, the relevant message is the message that caused the policy to execute. When the terminal is used within a social commitment action, the message is the message that caused the execution of the policy which created that social commitment. MAPC ensures that these terminals are never found in the start-up or idle code of an agent.

The this-agent terminal returns a reference to the agent currently executing. This terminal cannot appear inside the policy definitions (as these should always be relative to the message sender and receiver).

The neighbour function returns a reference to another agent in the system. The sole integer argument is used to determine which agent is selected. If the same agent executes the neighbour function with the same number multiple times within the same scenario, it is guaranteed to get a reference to the same agent each time.

The all-agents terminal returns a list of references to all agents in the current execution. This can be used as an argument to the send-message function to send a broadcast message to all agents.

The agents-by-role function returns a list of references to all agents in the current execution in the specified role. The system automatically generates a list of role constants for use as parameters to this function.

The send-message function causes the agent to send a CASA message with the specified performative, act, receiver(s), and integer contents.

The send-reply function causes the agent to send a reply message with the specified performative, act, and integer contents. This function can only exist in the action code for a social commitment, and it marks the message sent as a reply to the message which triggered the creation of the social commitment containing the action.

The if-else function takes three arguments. The first argument is a boolean value, while the second and third arguments are of type code. If the boolean value is true, the second argument is executed. If the boolean value is false, the third argument is executed.

An example of a successful Colour Query finder behaviour is shown in Fig. 4. In the example, the agent has a start-up function composed of a single send-message action. This is intended to trigger a policy in a paint agent. The agent’s idle function is blank (consisting of only a nop which does nothing). Finally, there are two policies which correspond to receiving a yes or a no response (determined by the act of the message) from the paint agent.

B. Genetic Operators

MAPC makes use of several standard genetic operators. These operators are broken into two categories: mutations and crossovers [4]. In addition, copying parents from one generation to the next (the reproduction operator) is also performed, though only when another operator fails to create a modified individual due to type and other constraints. The evolutionary system will determine whether a mutation or crossover should occur, and then one of those operators will be selected. Each of the operators will be described below.

There are two types of mutation operators, expansion mutation and subtree mutation, which are used to mutate a terminal or a subtree, respectively. Expansion mutation replaces a terminal node with a randomly generated subtree (which may be a terminal). The root node of the new subtree must have the same return type as the selected terminal.

Subtree mutation replaces a subtree with a randomly generated subtree (which may be a terminal). As with the expansion mutation, the root node of the new subtree must have the same return type as the root node of the selected subtree.

The crossover operator used is the standard subtree exchange crossover. Each crossover operation requires two parents (termed mother and father to aid in the discourse) and generates two children. Subtree exchange crossover uses a randomly selected subtree in the mother to replace a randomly selected subtree in the father to create one child. The randomly
selected subtree from the father replaces the randomly selected subtree in the mother to create the other child. The root nodes of the selected subtrees must both have the same return type. To accomplish this, one subtree is chosen, and then another is selected from the possibilities in the other parent. If no match is found, the crossover is aborted.

C. Fitness Function

In each cycle of the genetic algorithm, each individual behaviour must be evaluated. This is done by selecting a scenario instance from the scenario and executing the system for a given amount of time (1 second by default). After the execution, the desired effect function is executed to get an execution score. Behaviours that remain for several generations are evaluated multiple times, in different scenario instances each time.\(^5\) The fitness for an individual is then the maximum of all execution scores that that behaviour has accumulated. Any behaviour that has not been evaluated the maximum number of times is considered ideal if it has a fitness of zero. A behaviour is not considered to be ideal even if it has a fitness of zero if it has not been evaluated the maximum number of times. As a special case, a behaviour with a fitness of zero is repeatedly evaluated until the maximum number of executions has been reached (at which point the behaviour is considered ideal) or the fitness is non-zero. This method prevents the system from selecting for behaviours that “guess” the correct answer and get lucky in their scenario instance choice.

Several key values for the genetic programming used in MAPC have not been defined here as they are not directly relevant to the content of this paper. The values for these parameters are indicated in the Koza Tableau for MAPC (see Table III) for completeness. Refer to [4] for a complete explanation of these parameters.

VI. THE SELECTOR GAME SCENARIO

As MAPC was created, several simple scenarios were developed. One of them is the Selector Game. In this game, there is one selector and several choices. Each of the choices is either acceptable or unacceptable, as determined by a role instance variable. The desired effect is that the selector will select all of the acceptable choices, and none of the unacceptable choices. The technical details of the Selector Game are given in Table IV.

MAPC was able to generate results which met all of the desired effects of the Selector Game. One example of results is shown in Fig. 5. As with many systems that make use of evolution, the results were not what was originally envisioned for a solution. It was predicted that the Selector Game would prompt MAPC to create a query-like protocol, which the selector would use to ask each choice if it was acceptable. Instead, a simple advertisement protocol was developed, where the choice agents broadcast a message with the performative of “petition” and an act of “act2” if they are acceptable, and the selector reacted to any message of that form by selecting the message sender.

VII. RELATED WORK

Much research into the evolution of communication has been done under the banner of artificial life, but most of that research has been into simple utterances which are the equivalent of grunts, howls and chirps [10]. The small amount of research that has involved more complex or structured communication has typically been centred on what Wagner et al. describe as unsituated agents. Unsituated agents are agents that do not interact with an environment beyond the use of communications channels, and typically only exist to communicate some idea. The system outlined in this paper creates higher level communication protocols and will therefore extend the range of research in evolving communications.

There is very little research that deals with evolving communications protocols. The work that is most similar to that discussed in this paper is by Clark and Jacob, who present a novel method for the creation of security protocols [11]. They make use of the BAN logic named after its original authors Burrows, Abadi, and Needham [12]. BAN logic was designed to reason about protocol abstractions. BAN logic is related to the BDI (beliefs, desires and intentions) model of agents. As such, it has the same limitations as BDI: it requires agents to be truthful [13]. Clark and Jacob specifically state that their investigation assumes truthfulness from the participants. Therefore, each utterance by an agent is a non-empty subset of that agent’s beliefs. Clark and Jacob actually reduce this

---

\(^5\) A behaviour can be evaluated a preset maximum number of times (a user configurable option).
The system has been implemented as described in this paper, and can currently evolve solutions to simple communicative scenarios. Future work will follow three main paths. First, problem-specific initialization routines and genetic operators should allow the system to work with more complex scenarios. Second, it is hoped that MAPC will be able to trace through several executions of the final behaviour in order to find the portions of the behaviours which are essential to bring about the desired effects, thus eliminating unnecessary portions of the final behaviours. Third, work will be done to automate the creation of the fitness functions which define the desired effects so that the designer will only need to describe the requirements in some logical language.

ACKNOWLEDGEMENTS

The authors would like to thank NSERC (Natural Sciences and Engineering Research Council of Canada).

REFERENCES


