Elements of an Agent-based Mediative Communication Protocol for Design Objects

Jamal A. Abdalla, American University of Sharjah (jabdalla@ausharjah.edu)

Summary

Integrated structural engineering system usually consists of large number of design objects that may be distributed across different platforms. These design objects need to communicate data and information among each other. For efficient communication among design objects a common communication protocol need to be defined. This paper presents the elements of a communication protocol that uses a mediator agent to facilitate communication among design objects. This protocol is termed the Mediative Communication Protocol (MCP). The protocol uses certain design communication performatives and the semantics of an Agent Communication language (ACL) mainly the Knowledge and Query Manipulation Language (KQML) to implement its steps. Details of a Mediator Agent, that will facilitate the communication among design objects, is presented. The Unified Modeling Language (UML) is used to present the Meditative protocol and show how the mediator agent can be use to execute the steps of the meditative communication protocol. An example from structural engineering application is presented to demonstrate and validate the protocol. It is concluded that the meditative protocol is a viable protocol to facilitate object-to-object communication and also has potential to facilitate communication among the different project participants at the higher level of integrated structural engineering systems.

1 Introduction

A typical integrated structural engineering system is likely to consist of several application modules such as modeling, analysis, design and detailing. Each module may contain a large number of design objects and the application modules themselves may be distributed across different platforms (Abdalla 1991, Fenves et al. 1990). In general, the design objects of the same or different application modules need to communicate data and information among each other (Kandlur et al. 1996, Abdalla and Powell 1995, IEEE 1994, ACM 1991). Such data will be communicated through the communication *Channels* that exist among design objects due to the inherent relationships among them (Abdalla 2002). Certain means of communications will be used to communicate the data and information among design objects. The means assumed in this paper are the *Messenger* objects which are mainly *Argument* objects and *Response* objects (Abdalla 2002). There are several types of communication protocols for object-to-object communication that have recently emerged such as the *Prescriptive* protocol and *Conversational* protocol (Abdalla 2002). Almost all object-oriented programming languages have a built-in *Prescriptive* communication protocol that uses the message sending paradigm as its back-bone. The *Prescriptive* protocol, though efficient, nevertheless it produces coupled software, among many other drawbacks. The *Conversational* protocol requires all design objects to have enough built-in intelligence for them to hold converstion inorder to be able to communicate information each other. Brief description of these protocols will be presented. The proposed *Mediative* protocol requires the intelligence to be built in the *Mediative agent* which is the main facilitator for communication among design objects

This paper presents the elements of the Mediative communication protocol. This protocol uses a Mediator agent to facilitate communication among design objects. The Mediative protocol uses certain design communication performatives and the semantics of an Agent Communication language (ACL) mainly the Knowledge and Query Manipulation Language (KQML) to

implement its steps (Finin et al. 1994). Details of a Mediator Agent, that will facilitate the communication among design objects, is presented together with some notes on its implementation. The Unified Modeling Language (UML) (Booch et al. 1999) is used to present the Meditative protocol and show how the Mediator agent can be use to execute the steps of the Meditative communication protocol. An example from structural engineering application will be presented to demonstrate and validate the protocol. It is concluded that the meditative protocol is a viable protocol to facilitate object-to-object communication and also has potential to facilitate communication among the different project participants at the higher level of integrated structural engineering systems.

2 Objects and Agents

In recent years, software engineering researchers have viewed agent-oriented paradigm as a natural supplement or even a successor to object-oriented paradigms (Wooldridge and Jennings 1995, Muller 1997). This stem from the fact that integrated engineering systems are characterized by distribution, that are dynamics in nature and requires high level of interaction, therefore they are more amenable to agents-which are more active than objects.

Although two decades or so had passed since the object-oriented model had taken the software community by storm, the object-oriented model is still evolving, and several enhancements can be made (OOPSLA 1987-2003; Booch 1994). The essential features of the object-oriented model have been well established. In short, all entities, whether physical or conceptual, are represented by *objects* that are instance of *classes*. Objects communicate by sending *messages* and receiving *responses*. Messages and responses are the basic ingredients of the object-oriented model for communication among objects.

Agents are autonomous software components (Muller 1997, Tveit 2001), according to Woodridge and Jennings (1995), a software agent, in the weak sense, are those that possess the following properties: (1) *autonomy*, i.e., operate without intervention and have control over their states and actions; (2) *reactivity*, i.e., *perceptive* and are aware of their environment and have the ability to respond in a timely manner to the changes and actions that occur; (3) *pro-activeness*, i.e., take the initiative and are able to exhibit goal-directed behavior; and (4) *social ability*, i.e., *co-operative* or have the ability to interact with other agents and objects with some kind of language. In addition to these, agents in the strong sense, may possess additional characteristics that include: (5) *mobility*, i.e., ability to move around; (6) *rationality*, i.e., ability to perform in optimal manner to achieve goals; (7) *benevolence*, i.e., obey; and (8) *veracity*, i.e., truthful. There are several software agents that are currently in use such as the animated paperclip agent in Microsoft office, computer viruses, web spiders, artificial players in computer games, among others.

In spite of the fact that Agent-oriented programming has been viewed as an extension to object-oriented programming, nevertheless, there are many differences between objects and agents as follows (Wooldridge and Jennings 1995): (a) objects are mainly passive entities while agents are active and autonomous entities with beliefs, commitments and interactions capabilities; (b) the decision about whether to execute an action lies with the object that sends (invoke) the method in the case of object-oriented model, however, in the case of agents, the decision lies with the agent that receive the message or request ; (c) Agents have their own thread of control-continually observing their environment, updating their internal state, selecting and executing actions as they desire.

3 Agents Communication Languages and Protocols

Knowledge Query and manipulation Language (KQML) is a language and a protocol for exchanging knowledge and information to support communication between software agents. Based on ideas from speech act theory, Finin et al. (1994) proposed a semantic description for KQML that associated descriptions of the cognitive states of agents with the use of the language's primitives (performatives). KQML consists of three layers – content layer, message layer and communication layer. The content layer specifies the proposal of the message based on specific ontology – common terms and their real world meaning that is common among the communicating agents. The message layer provides a set of performatives that can be sent between agents such as ask, reply, tell. The communication layer defines the protocol for delivering the message and its contents. KQML performatives are classified into seven categories: (1) basic queries – for asking basic questions (ask-if, ask-one, ask-in, etc); (2) multi-response queries – for handling long answers (stream-in, stream-out, etc.); (3) responses – for simple answers to queries (sorry, reply, etc.); (4) generic informational – for informing other agents without preceding query (achieve, tell, cancel, etc.); (5) generators – for synchronizing matters (ready, standby, next, disrecard, etc.); (6) capability-definitions for managing services between agents (subscribe, advertise, monitor, import, etc.); and (7) networking – for net-organizational purposes.

Agent Communication Language (ACL) was developed by the Foundation for the Intelligent Physical Agents (FIPA) to offset the shortcomings of KQML such the lack of precise semantics of the defined performatives. Several agent communication languages have emerged over the years. Examples are Actors (Agha 1996, Agha et al. 1995, Genesereth and Ketchpel 1994).

4 Objects Communication Protocols

The framework for communication protocols between a client (requester) and a server (provider), as an extension to the well known Contract Net Protocol (Parunak 1987), involves at least six stages as shown in Figure 1: (1) preparation of the proposal or request by the client; (2) sending of the proposal or request from the client to the server; (3) acceptance of the proposal by the server; (4) execution by the server of what is proposed by the client; (5) preparation by the server of the result of the proposal; (6) returning of the results of the execution back to the client by the server; (7) acceptance of the results by the client. In object-to-object communication the client is the message sender object and the server is the message receiver object. The proposal is the message sent (*Argument* object) and the result of the proposal is the response received (*Response* object). The seven stages outlined above are the fundamentals of communication protocols between any two parties. Based on this framework the Mediative protocol will be presented.

Before presenting the details of the *Mediative protocol*, the following rules and principles which govern the use and ownership of *Response* and *Argument* objects should be emphasized.

(1) The arguments for any message from a design object to another design object are passed via an object of type *Messenger* class, called the *Argument* object.

(2) The response items are passed back by an object of type *Messenger* class, also, called the *Response* object.

(3) A transaction begins when the *Argument* object is constructed and ends when the *Response* object is destroyed. *Argument* and *Response* objects are thus temporary.

(4) Transactions will frequently be nested. That is, the message receiver object may itself send messages to other objects, and hence will act as an intermediate message sender object. The message receiver object may create several sub-transactions before it returns the final *Response* object to the original message sender object.

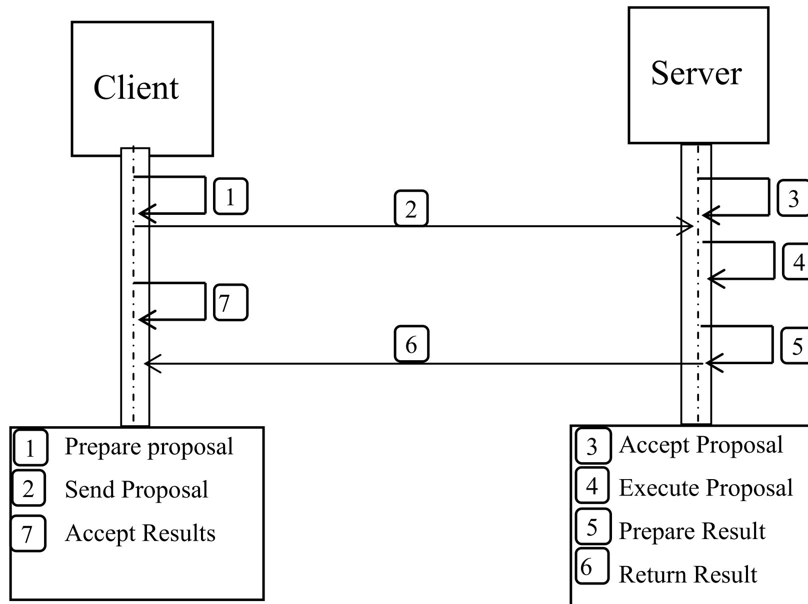


Figure 1 UML Sequence Diagram for a Framework for Client-Server Communication Protocol

4.1 The Prescriptive Protocol

This protocol of communication assumes that the message sender object and the message receiver object knows each other's needs. I.e., the message sender object knows exactly what arguments the message it intends to send needs in order to perform its task, and also the message receiver object knows the format of the response items that is requested by the message sender object. The steps of the prescriptive protocol are graphically depicted in Figure 2.

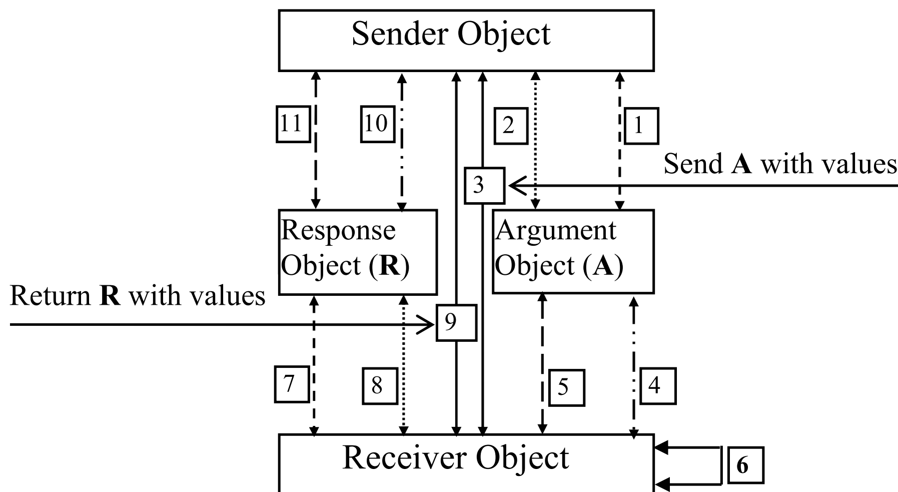
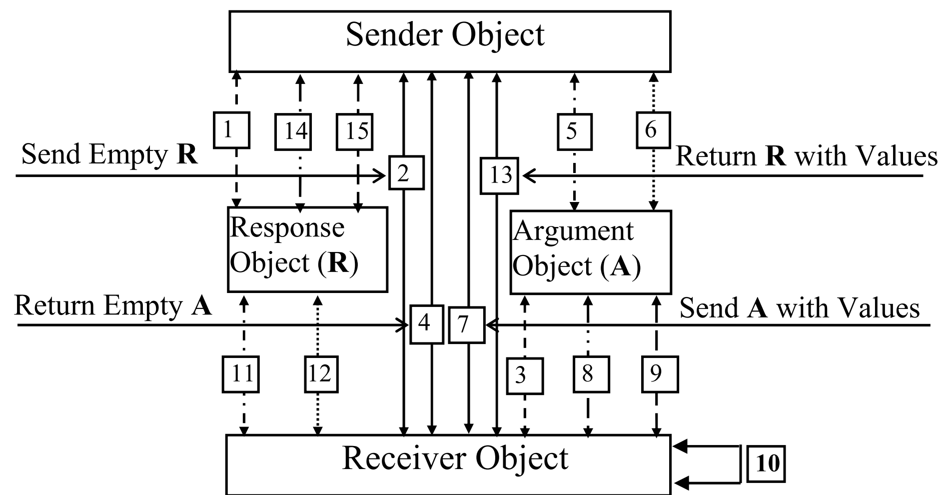


Figure 2 The Prescriptive Communication protocol

4.2 The Conversational Protocol

This protocol of communication is envisaged when the message sender object does not know what arguments the message it intends to send needs to accomplish its task, and the message receiver object does not know what data the message sender wants and in what form the response should be (i.e., no previous knowledge about each other's need). This protocol is of conversational type in which the message sender object and the message receiver object will engaged in a dialogue in order for each to provide the other with what it actually needs. The steps of the conversational protocol are graphically depicted in Figure 3.



Legend

Get Values	Set Values	Initialize Slots
Extract Slots	Destroy object	Any Message

Figure 3 The Conversational Communication protocol

4.3 The Mediative Protocol

This type of protocol involves a third party or an agent, the *Mediator* agent that mediates between the two communicating objects. The *Mediator* agent knows more about the needs of the two communicating objects than they do know about each other's need. Therefore, the communication between design objects, sender and receiver, and the *Mediator* agent is done using the Prescriptive protocol based on the assumption that the *Mediator* object and the design objects know about each others need. However, communication between two design objects is not done directly, instead through the *Mediator* agent. Figure 4 shows the Mediative protocol and the corresponding steps involved.

4.3.1 Elements of the Mediative Communication Protocol

Based on Figure 4, the steps of the Mediative protocol can be outlined as follows:

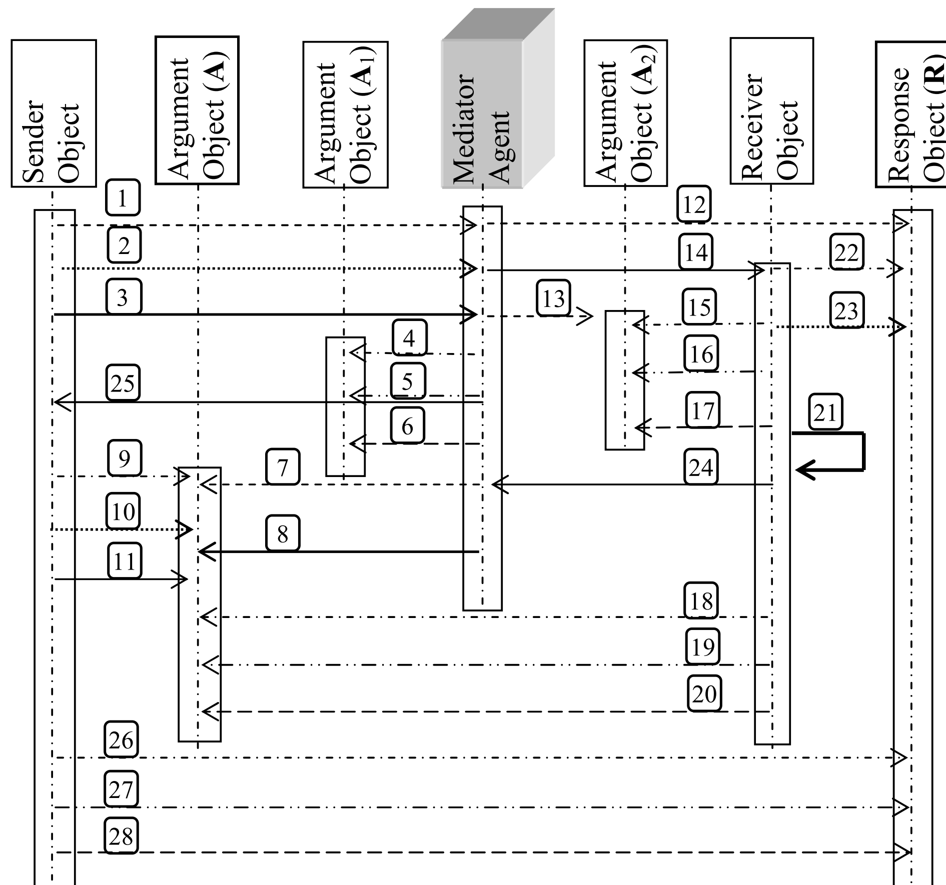
1. The sender object constructs the first *Argument* object using the Construct method and initializes its data slots using the Initialize method.
2. The sender object then populates the first *Argument* object using the Set method. The message sender object then sends a message to the *Mediator* agent expressing its desire to send a particular message and inquiring about the arguments needed for executing this particular message.

3. The *Mediator* agent extracts the data slots from the first *Argument* object using the Extract method. The *Mediator* agent then retrieves data values from the first argument object using the Get method.

4. The *Mediator* agent, knowing the message and what arguments it needs, constructs the second *Argument* object using the Construct method and initializes its empty data slots using the Initialize method. It then returns this empty *Argument* object to the sender object as a response to its message (**A**).

5. The message sender object, with the given information from the Mediator agent, extracts the data slots from the second *Argument* object using the Extracts method.

6. The sender object, knowing the data slots *Argument* object, populates them with data values using the Set method. The sender object then send the message to the *Mediator* agent (*Argument* object with data values).



Legend

Get Values	Set Values	Initialize Slots
Extract Slots	Destroy object	Any Message

Figure 4 UML Sequence Diagram for the Mediative Communication Protocol

7. The *Mediator* agent, given the response data and form, constructs a *Response* object using the Construct method and initializes its empty data slots using the Initialize method. It then sends the message, with the populated *Argument* object and the empty *Response* object to the receiver object.
8. The *Mediator* agent constructs the third *Argument* object using the Construct method and initializes its empty data slots using the Initialize method.
9. The *Mediator* agent then populate the third *Argument* object with values using the Set method. It then sends the message, with this *Argument* object to the receiver object. This *Argument* object contains the populated second *Argument* object and the empty *Response* object.
10. The message receiver object, with the given information from the *Mediator* agent, extracts the data slots from the third *Argument* object using the Extracts method.
11. The message receiver object then retrieves the data values from the third *Argument* object using the Get method and destroys it using the Destruct method.
12. The message receiver object then extracts the data slots from the second *Argument* object using the Extract method.
13. The message receiver object then retrieves the data values from the second *Argument* object using the Get method and destroys it using the Destruct method.
14. The receiver object then extracts the data slots from the *Response* object **R** sent by the *Mediator* agent using the Extract method.
15. The message receiver object then proceeds in its routine calculations assuming that all the data it needs is now available. The receiver then populates the *Response* object with the requested response values using the Set method and returns the message to the *Mediator* agent with the populated *Response* object R. The *Mediator* agent will then return the populated *Response* object **R** to the message sender object.
16. The message sender object extracts the data slots from the *Response* object using the Extract method.
17. The message sender object retrieves the data values from the *Response* object using the Get method and destroys the *Response* object using the Destruct method. This signals the end of the transaction.

4.3.2 Language Description of the Steps of the Mediative Protocol

Using the protocol performatives, the steps of the Mediative protocol can be represented. There are three Argument that need to be constructed for the executing the steps of the Mediative protocol. Let $\mathbf{A_1}$ = Argument object for the first message; **A** = Argument object for the second message, which is the primary argument object; and $\mathbf{A_3}$ = Argument object for the third message, **R** = Response object. The steps of the Mediative protocol can be represented as shown in Table 1 Below.

4.3.3 Example Illustrating the Mediative Protocol

To illustrate the Mediative protocol, a message that involves interaction with several design objects with be used as an example. Consider the equation for checking the flexural design strength M_u at a given cross section of a Rectangular Reinforced Concrete Beam (*RCBeam*) object with effective depth d, width b_w, area of steel A_s, steel yield strength f_y and concrete crushing strength f'_c.

Table 1. Language Description of the Steps of the Mediative Protocol

```
Sender:
      (Beginning of First Transaction)
      Step 1: Construct A1 and Initialize its Data Slots
      Step 2: Set Data Values of A1
      Step 3: Send message to Mediator indicating its intend.
      (This analogous to steps 1-6 of Prescriptive protocol)
Mediator: (after receiving the first message from Sender)
      Step 4: Extract Slots of A1
      Step 5: Get Data Values from A1
      Step 6: Destruct A1
      Step 7: Construct  A and Initialize its Data Slots
      Step 8: Return A as a Response to Sender
      (End of First Transaction)
Sender: (after receiving first response from Mediator)
      (Beginning of Second Transaction)
      Step 9: Extract Data Slots from A
      Step 10:Set Data Values of A
      Step 11:Send A with the Message to Mediator
Mediator: (after receiving the second message from Sender)
      (Beginning of Third Transaction)
      Step 12:Construct R and Initialize its Data Slots
      Step 13:Construct A3 and Initialize its Data Slots
      Step 14:Send the Message to the Reciever with A3
      (A3 containing both A and R)
      (A with Data Values, R with Data Slots only)
Receiver: (after receiving the message from the Mediator)
      Step 15:Extract Data Slots from A3
      Step 16:Get Data Values from A3
      Step 17:Destruct A3
      Step 18:Extract Data Slots from A
      Step 19:Get Data Values from A
      Step 20:Destruct A
      Step 21: Execute the Method
      Step 22:Extract Data Slots from R
      Step 23:Set Data Values of R
      Step 24:Return Response R to Mediator
      (End of Third Transaction)
Mediator: (after receiving the response from the Receiver)
      Step 25:Return Response R to Sender
Sender: (after receiving second response from Mediator)
      Step 26:Extract Data Slots from R
      Step 27:Get Data Values from R
      Step 28:Destruct R
      (End of Second Transaction)
```

As given by (ACI 2002), the flexural design strength can be written as follows:

$$M_u \leq \Phi M_n = \Phi \rho b_w d^2 f_y \left(1 - 0.59 \rho \frac{f_y}{f'_c} \right) \tag{1}$$

where: M_n = Nominal Flexural Strength, Φ = Resistance Factor, $\rho = \frac{A_s}{b_w d}$ = Steel ratio.

The message for checking the flexural strength (`CheckFlexuralSrength`), will be sent to a *RCBeam* object, from the design application *Driver* object, to check the flexural design strength at one or a number of cross sections for a given object (Abdalla 1996, Abdalla 2002). Figure 5 shows the ingredients of the message for checking flexural strength, where the data values needed to compute the final response of the message are the nodes and leaves of the message tree. The argument to the `CheckFlexuralStrength` message is an Argument object (A) which contains information about the location where the flexural strength need to be checked. The response to the message is a Response object (R) containing the $M_u / \Phi M_n$ ratio at this location. The ingredient data for executing this message are shown schematically in Figure 5.

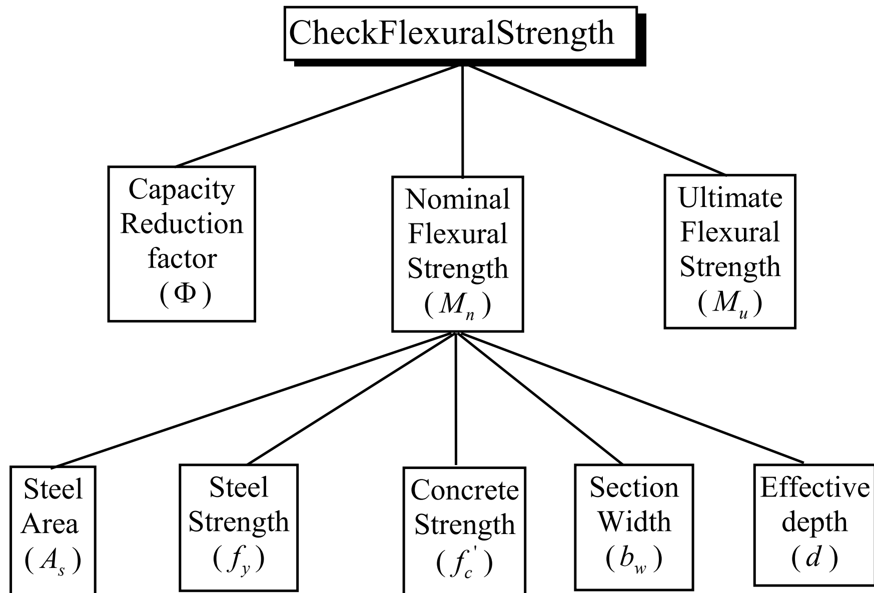


Figure 5 Use Case Diagram for the Ingredients for Checking Flexural Strength at a Section

The steps of the Mediative protocol can be described using the following set of messages. As shown in Figure 6, it takes three messages before the final response is returned:

```
A = Mediator CheckFlexuralStrength(A1)
R = Mediator CheckFlexuralStrength(A)
R = RCBeam   CheckFlexuralStrength(A3)
```

Using KQML performatives the last message can be expressed as follows. This syntax can be expanded to write all the steps of the mediative protocol.

```
(CheckFlexuralStrength
     :content <Argument Object A3>
     :language <Protocol performatives>
     :ontology <word>
     :reply-with <Response Object R>
     :sender <Driver Object>
     :receiver <RCBeam Object>
 )
```

The first message is from a *Driver* (sender) object to the Mediator agent with intention to send the `CheckFlexuralStrength` message to *RCBeam* (receiver), **A** is the returned Argument object with slots. The second message is from *Driver* (sender) to the *Mediator* agent with populated **A** and populated **R** is returned. The third message is from the *Mediator* agent to *RCBeam* (receiver) with **A** containing populated **A** and empty **R** and returns **R** with values. It is clear that the third message is nested within the second message, i.e., it will be sent before the response of the second message is returned to the message sender object.

Figure 6 shows a message tree where the data values needed to evaluate the message are the nodes and leaves of the tree. However, an *RCBeam* object has channels of communication with other objects that hold this data, but these channels are through the *Mediator* agent. Therefore, channels of communications are established between the *RCBeam* object and the objects related to it via a *Mediator* agent. As a result, messages must be send to these objects to get the data values as shown above. Remember also, three messages, similar to the those above, are sent to retrieve each data value needed for computing the final response as shown in Figure 6.

Figure 6 Collaboration Diagram of the Mediative Protocol for Message CheckFlexuralStrength

5 Summary and Conclusions

This paper clearly identified the needs for having a standardized protocol to provide a uniform scheme for object-to-object communication. It also identified the major elements of the meditative communication protocol. Performatives for executing the steps of the Mediative protocols, for object-to-object communication, have been defined. The steps of the Meditative communication protocol have been outlined based on Messenger objects (Argument and Response) and protocols performatives. Although the Mediative protocol provides a uniform mean for object-to-object communication, however, it involves some computational overhead that results from the construction and destruction of Argument and Response objects and population and retrieval of data from these objects. This is likely to hinder the performance and compromise efficiency.

It is observed that the *Mediator* agent, like a switch board, facilitates the communication between objects. The *Mediator* agent constructs both the final Argument and Response objects and sets their slots. That is due to its knowledge of the needs of the communicating objects. The message sender object destroys the Response object and the message receiver object destroys the Argument object.

While the Mediative protocol suggested here is for object-to-object communication, however, this protocol can be modified and used for communication among project participants and application programs across different levels and layers of the integrated engineering systems. Certainly to achieve such integration in engineering systems, more elaborate work of sufficient rigor and wider scope is to be carried out in the area of formalization, specification, and testing of communication protocols.

6 References

Abdalla, J. A. (2002). "Towards an Object Communication Model for Structural Engineering Design." In the proceedings of The 9th International Conference on Computing in Civil and Building Engineering (ICCCBE-IX), Taiwan, April.

Abdalla, J. A. and Powell, G. H. (1995). "Object Design Framework for Structural Engineering." The Journal of Engineering with Computers, 11(4), 213-226.

Abdalla, J. A. (1991). An Object-Oriented Architecture and Concept for an Integrated Structural Engineering System. Proceedings of the Second International Conference on Application of Artificial Intelligence Techniques to Civil and Structural Engineering. Oxford, England, September 3rd-5th, 1991.

ACI (2002). Building Code Requirement for Reinforced Concrete, ACI committee 318, American Concrete Institute, Detroit, MI.

ACM (1991). Communication of the ACM, Special Issue in Collaborative Computing,34(12).

Agha, G. (1996). Modeling Concurrent Systems: Actors, Nets and the Problem of Abstraction and Composition. Application and Theory of Petri Nets 1996: Proceedings of the 17th International Conference, Lecture Notes in Computer Science, vol. 1091, pp 1-10, J. Billington and W. Reisig (Editors), Springer-Verlag.

Agha, G., Kim, W. Y. and Panwar, R. (1995). Actor Languages for Specification of Parallel Computations. DIMACS Series in Discrete Mathematics and Computer Science, vol. 18, pp 239-258, G. E. Blelloch, K. Mani Chandy and S. Jagannathan (editors), American Mathematical Society.

Booch, B. Rumbaugh, J. Jacobson, I. (1999). The Unified Modeling Language User Guide, Addison-Wesley.

Booch, G. (1994) Object-Oriented Design with Applications, Benjamin/Cummings Pub. Co. Redwood City, CA.

Fenves, S. J., Flemming, U., Hendrickson, C., Maher, M. L., and Schmitt, G. (1990) "An Integrated Software Environment for Building Design and Construction." Computer-Aided Design, 22, 27-36.

Finin, T., Fritzson, R., McKay, D. and McEntire, R. (1994). “KQML as an Agent Communication Language.” The Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94), ACM Press, November 1994.

Genesereth, M. R. and Ketchpel, S. P. (1994). “Software Agents.”Communication of the ACM, Vol. 38, No.7, pp. 54-67.

IEEE (1994). “Reliable Software and Communication I, II, and II.” IEEE Journal on Selected Areas in Communications, 12(1), 23-32.

Kandlur, D. D., Saha, D. and Wilkbeek-LeMair, M. (1996). “Protocol Architecture for Multimedia Applications over ATM Networks.” IEEE Journal on Selected Areas in Communication, 14(7), 1349-1359.

Muller, H. J. (1997). “Towards agent systems engineering.” KNOWLEDGE AND Data Engineering Vol. 23, pp. 217-245.

OOPSLA (1987-2003). Object-Oriented Programming, Systems, Languages and Applications.

Parunak, H. V. D. (1987). “Manufacturing experience with the contract net, in M. Huhns (eds), Distributed Artificial Technology (Pitman Pubs. And Morgan Kaufman, pp.285-310.

Tveit, A. (2001). A survey of Agent-Oriented Software Engineering. First NTNU CSGSC, May 2001.

Wooldridge, M. and Jennings, N. R. (1995). Intelligent Agents: Theory and Practice. The Knowledge Engineering Review, Vol. 10, No. 2, pp.115-152.