Coordination of View Maintenance Policy Adaptation Decisions: A Negotiation-Based Reasoning Approach

Prasanta Bose[1] and Mark G. Matthews[2]

[1] Information and Software Engineering Department, George Mason University, Fairfax, VA 22030
bose@isse.gmu.edu
[2] The MITRE Corporation, 1820 Dolley Madison Blvd., McLean, VA 22012
mmatthew@mitre.org

Abstract. In mission critical applications of distributed information systems, autonomous information resources are coordinated to meet the information demands of client specific decision-support views. A major challenge is handling dynamic changes in QoS constraints of the clients and/or changes in QoS properties of the resources. This paper presents a negotiation-based adaptive view coordination approach to address such run-time changes. The three key ideas are as follows. a) A negotiation-based reasoning model for adapting view maintenance policies to meet changes in QoS needs and context constraints. b) A dynamic software architecture of the collaborating information resources supporting the client task of maintaining a specific view. c) Coordination mechanisms in the architecture that realize negotiated changes in the policies for view maintenance. The paper describes an initial prototype of the support system for the supply-chain task domain.

1 Introduction

In mission critical applications of distributed information systems, autonomous information resources are coordinated to meet the information demands of client specific decision-support views. . A major challenge is handling dynamic changes in quality of service (QoS) properties and constraints of the clients, information resources, and shared infrastructure resources. Current view coordination approaches are static in nature and cannot be dynamically changed to meet changing demands. Consider the following scenario from the supply-chain domain.

A decision-support view for inventory management is maintained from multiple autonomous information resources within the supply-chain. Customer order information from customer sites, product assembly information from manufacturer sites, and parts inventories from parts supplier sites are configured to support an "Order-Fulfillment" view used by inventory managers of the suppliers and consumers. As orders, product assembly requirements, and parts inventories constantly change, changes in the view must be coordinated to achieve consistency and to support management decisions.

There are multiple view change coordination policies available to support the above inventory management task. When selecting a policy to implement, one must consider tradeoffs between consistency, communications costs, and processing costs. Currently, the view coordination policy is set at design-time and is static. Suppose a high-cost complete consistency view maintenance policy was selected for implementation at design-time. Further suppose that several inventory managers are simultaneously executing intensive on-line analytical processing queries against the `Order-Fulfillment` view. The queries are competing with the view coordination policy for system resources. Under these conditions, both the queries and the view maintenance task are likely to suffer from poor performance. Short of shutting down, reconfiguring, and restarting the system, current view coordination approaches have no way of prioritizing preferences and dynamically responding to changing preferences and constraints.

1.1 Self-Adaptive Software: Requirements and NAVCo Approach

Self-adaptive software systems that can dynamically adapt internal mechanisms and components in response to changing needs and context are required for addressing the above problems. There are four major requirements that a self-adaptive software system must meet. i) *Detecting a change* in context or a change in needs. It should be able to monitor its behavior and detect deviations from commitments or the presence of new opportunities. It should be able to accept new needs from external sources and evaluate for deviations with respect to current commitments. ii*) Knowing the space of adaptations*. It must have knowledge of the space of self-changes it can choose from to reduce deviations. iii) *Reasoning for adaptation decision*. It should be able to reason and make commitments on the self-changes and commitments on revised goals. iv) *Integrating the change*. It should be able to package the change if required and do assembly/configuration coordination to insert the change into the existing system with minimal disruption to existing behaviors.

The NAVCo approach described in this paper considers a family of adaptive systems that involve information view management and makes specific design choices to meet the requirements. In particular, the approach considers the following. a) Changes in committed preferences and context assumptions to trigger the adaptation process. b) An adaptation space defined by a set of view coordination policy objects. c) Reasoning for change as a negotiation-based process involving client and information resource agents. d) Use of assembly plans for change integration. Our current change integration approach relies on forcing the active view coordination objects to a quiescent state and then, based on the results of the negotiation-based reasoning, dynamically switching to an alternate set of objects within the space of adaptation.

The following sections of the paper focus primarily on the negotiation-based coordination to decide on the adaptation. Section 2 presents background information. Section 3 presents the NAVCo approach and architecture. An initial prototype of the support system for the supply-chain task domain is described in Section 4. Related work is discussed in Section 5. A summary is presented and future work discussed in Section 6.

2 Multi-Resource View Coordination Architecture

Multi-resource view maintenance falls within the domain of distributed decision-support database systems. A simplified model of this domain is illustrated in Figure 1. As illustrated in Figure 1, a view (V) is maintained from a set of autonomous data sources (S_1, S_2,…,S_n). The view is a join of relations (r_1,r_2,…,r_n) within the data sources. The update/query processor and view coordination object execute a distributed algorithm for incrementally maintaining the view. As data within a source changes, the associated update/query processor sends notification of the update to the view coordination object in Figure 1 which in turn queries the other sources to compute the incremental effect of the source update. After the incremental effect of the update has been computed, it is propagated to the client view. Client applications, such as on-line analytical processing and data mining applications, execute queries against the view. The data sources also support transactional environments, which result in updates to source relations that participate in the view.

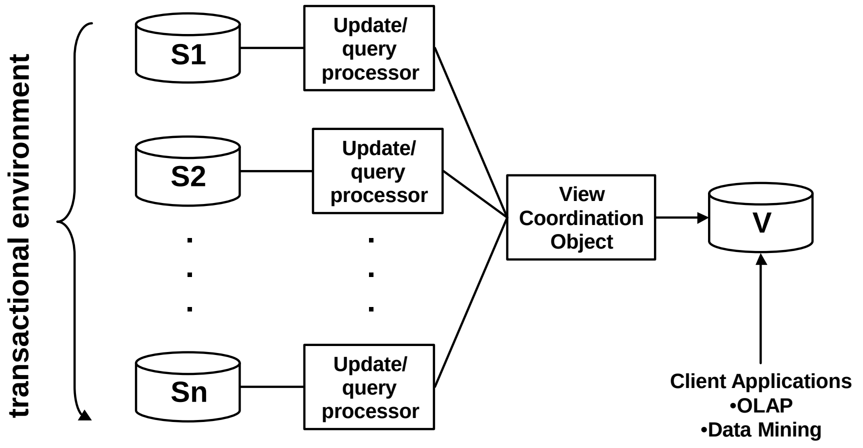


Fig. 1. Distributed Decision-Support Database System Domain

2.1 View Coordination Objects: Background

View coordination object (VCOs) are algorithmic objects that correspond to different policies for view maintenance. These view maintenance algorithms focus on maintaining a materialized view at the client in the presence of concurrent updates to the data resources. Four VCOs are briefly discussed and compared in this section. A complete description of these algorithms can be found in [1, 27] and is beyond the scope of this paper.

View coordination objects can be differentiated based on the level of consistency provided. Four levels of consistency (convergence, weak, strong, and complete) have been defined [1]. Here we consider VCOs that provide either a strong or a complete level of consistency. Strong consistency requires the order of the view states to match the order of source updates. Strong consistency allows global states to be skipped (i.e., every source update need not result in a distinct view update). Complete consistency is more restrictive requiring every source state to be reflected as a distinct view state. With complete consistency there is a complete order-preserving mapping between view states and source states.

The Strobe algorithm is an incremental algorithm that achieves strong consistency. The Strobe algorithm processes updates as they arrive, sending queries to the sources when necessary. However, the updates are not performed immediately on the materialized view; instead, a list of actions to be performed on the view is generated. The materialized view is updated only when it is certain that applying all of the actions in the action list as a single transaction at the client will bring the view to a consistent state. This occurs when there are no outstanding queries and all received updates have been processed.

The Complete-Strobe (C-Strobe) algorithm achieves complete consistency by updating the materialized view after each source update. The C-Strobe algorithm issues compensating queries for each update that arrives at the VCO between the time that a query is sent from the VCO and its corresponding answer is received from a source. The number of compensating queries can be quite large if there are continuous source updates.

The SWEEP algorithm achieves complete consistency of the view by ordering updates as they arrive at the VCO and ensuring that the state of the view at the client preserves the delivery of updates. The key concept behind SWEEP is on-line error correction in which compensation for concurrent updates is performed locally by using the information that is already available at the VCO. The SWEEP algorithm contains two loops that perform an iterative computation (or sweep) of the change in the view due to an update.

The Nested SWEEP algorithm is an extension of the SWEEP algorithm that allows the view maintenance for multiple updates to be carried out in a cumulative fashion. Nested SWEEP achieves strong consistency by recursively incorporating all concurrent updates encountered during the evaluation of an update. In this fashion, a composite view change is computed for multiple updates that occur concurrently.

The performance of VCOs can be compared based on the communications and processing costs required to maintain a given level of consistency. Communications costs can be measured with respect to the number and size of messages required per update. Processing costs must be measured with respect to the processing burden that the algorithm places on both the client and the data sources.

Table 1 compares the communications and query processing cost of the four view maintenance algorithms discussed above. The cost of the algorithms is dependent on the number of data sources, n. The costs of the C-Strobe and Nested SWEEP algorithms are highly dependent on a workload characterization factor, a where $0 \le a \le 1$, which reflects the rate of updates received. If updates arrive infrequently $a=0$ and if updates arrive continuously $a =1$. The client processing cost of a delete update in the Strobe and C-Strobe algorithms is highly dependent on the number of pending updates, p. The costs in Table 1 depict the case in which the VCO is co-located with the client.

Table 1. View Coordination Object Comparison

Algorithm	Consistency Level	Update Type	Comm Cost	Client Cost	Server Cost
Strobe	Strong	delete	1	1+p	0
		insert	2n-1	(n-1)+1	1
C-Strobe	Complete	delete	1	1+p	0
		insert	2(n-1)+ 2a(n-1)!+1	(n-1)+ a(n-1)!+1	1+a(n-2)!
SWEEP	Complete	delete/insert	2n-1	(n-1)+1	1
Nested SWEEP	Strong	delete/insert	2(1-a)(n-1)+1	(1-a)(n-1)+1	(1-a)

2.2 Order-Fulfillment Scenario Revisited

To better understand the costs and demands of the algorithms we consider a supply-chain view coordination scenario with databases at client, manufacturer, and parts supplier locations.

A customer orders database is maintained at client locations for orders that get handled by an automobile manufacturer. Each time an order is placed for an automobile, a tuple is inserted into the `Orders` relation. When an order is filled, it is deleted from the `Orders` relation and an appropriate tuple is inserted into a `FilledOrders` relation. There is only one end-item (automobile) allowed per order. If a customer wishes to order two automobiles, then two separate orders are generated. The database schema for the `Orders` relation is as follows:

```
Orders(OrderID(PK),CustomerID,ModelNumber)
```

A product assembly database is maintained at manufacturer locations. This database maintains a `ProductAssembly` relation that captures the dependency of each assembled product on supplier parts. Each end item (automobile) that is ordered must be assembled from a set of sub-items. Each time the manufacturer offers a new automobile model, new tuples are inserted into the `ProductAssembly` relation. When a manufacturer discontinues a model, the tuples associated with the automobile model are deleted from the `ProductAssembly` relation. The database schema for the `ProductAssembly` relation is as follows:

```
ProductAssembly(ModelNumber(PK),PartNumber(PK),
QuantityRequired(PK))
```

A parts inventory database is maintained at parts supplier locations. This database maintains an inventory of the number of automobile parts (sub-items) that are in stock in the warehouse. As the quantity of parts at the warehouse changes, the corresponding tuples are updated in the `PartsInventory` relation. Tuples are inserted into the `PartsInventory` relation as the warehouse begins storing new parts and tuples are deleted as the warehouse discontinues the storage of particular parts. The database schema for the `PartsInventory` relation is as follows:

```
PartsInventory(PartNumber(PK),QuantityAvailable)
```

A decision-support database is maintained at management locations to support inventory management decisions. This database maintains a materialized view (MV) of the inventory of parts available to fill automobile orders. The database schema for the materialized view is as follows:

```
MV=Orders|x|ProductAssembly|x|PartsInventory
=(OrderID, CustomerID, ModelNumber, PartNumber,
QuantityRequired, QuantityAvailable)
```

The materialized view is maintained as orders are placed and filled, manufacturer models and model-part dependencies change, and as supplier inventories change. Supplier and consumer inventory managers utilize this view to perform on-line analytical processing tasks in support of their inventory management decisions.

2.3 Run-time Policy Switching: Cost-Benefit Analysis

As illustrated in Table 1, VCO cost is highly sensitive to the workload characterization (i.e., the volume and types of updates received). To illustrate the effect of workload on VCO cost, consider the order fulfillment scenario discussed earlier. Assume that there are four data resources and one client view. Further assume that over a period of time the system experiences the following workload.

? Period 1 -high vol., high insert, 100 inserts and 0 deletes over X seconds
? Period 2 - low vol., balanced, 50 inserts and 50 deletes over 3X seconds
? Period 3 - medium vol., high delete, 0 inserts and 100 deletes over 2X seconds

The cost of each algorithm over these periods can be calculated using the formulas in Table 1. The value of the parameter p is assumed to be 0 for low traffic, 10 for medium traffic, and 100 for high traffic. The value of the parameter a is assumed to be 0 for low traffic, 1/3 for medium traffic, and 1 for high traffic. The cost of the four algorithms over Periods 1-3 is illustrated in Table 2:

Table 2. View Coordination Costs in the Supply-Chain Example

Algorithm	Consistency Level	Comm Cost	Client Cost	Server Cost
Strobe	Strong	1200	1750	150
C-Strobe	Complete	2400	2350	350
SWEEP	Complete	2100	1200	300
Nested SWEEP	Strong	1250	775	158

With current technology a single algorithm is implemented during system configuration and can not be changed without shutting down and reconfiguring the system. The selection of an algorithm can have a profound effect on the processing and communications requirements to support the view. Tradeoffs must be made at design-time with respect to consistency versus client, server, and communications costs. If, however, the algorithm can be dynamically changed at run-time, these tradeoffs can be made continuously as preferences and constraints change. As illustrated in Table 3, the ability to dynamically switch algorithms can result in significant cost savings and improved performance in a constrained environment.

The first row in Table 3 shows that communications cost can be minimized by initially implementing the Nested SWEEP algorithm and then dynamically switching to the Strobe algorithm between periods 1 and 2. This results in a reduction of 450 messages over a static implementation of the Strobe algorithm.

The second row in Table 2 shows that client processing cost can be minimized by implementing the Nested SWEEP algorithm during periods 1 and 3, and the Strobe algorithm during period 2. This results in a reduction of over 1000 queries over a static implementation of the Strobe algorithm. This would free up valuable resources for the processing-intensive analysis users and result in a significant performance improvement for those users.

Table 3. Results of Dynamic Switching of View Coordination Algorithms in the Example

Preferences/ Constraints	Comm Cost	Client Cost	Server Cost	Period 1	Period 2	Period 3
Minimize comm cost	750	1525	75	Nested Sweep	Strobe	Strobe
Minimize client cost	950	625	108	Nested Sweep	Strobe	Nested Sweep
Minimize server cost	750	1525	75	Nested Sweep	Strobe	Strobe
Minimize comm cost and complete consistency	1200	1750	150	Sweep	C-Strobe	C-Strobe
Minimize client cost and complete consistency	1350	825	175	Sweep	C-Strobe	Sweep
Minimize server cost and complete consistency	1200	1750	150	Sweep	C-Strobe	C-Strobe

3 The NAVCo Approach

The NAVCo approach to adapting view coordination policies in response to changes in needs of the clients or change in constraints imposed by the resources is based on negotiation reasoning between the client and resource objects. The approach introduces negotiation reasoning models and adaptive policy reconfiguration mechanisms to the existing view coordination application architecture. The architecture, shown in Figure 2 as a UML class diagram, introduces a negotiation layer to perform dynamic negotiation-based selection of coordination policies. Additional software mechanisms are introduced to bring about the dynamic switching of the coordination objects.

The key elements of the architecture are: a) Models and reasoning support for model-based coordination negotiation via Role Negotiation Agents (RNAs) and a Negotiation Facilitator Agent (NFA) that communicate via the shared coordination negotiation data space (CNspace). b) Models and support for switching based on negotiated switching decisions. The change reasoning and change coordination views are integrated via a shared data space whereby the negotiation facilitation agent communicates with team level coordination agents via the CNspace [6]. The Change Coordination Agent (CCA) aids in coordinating the switching.

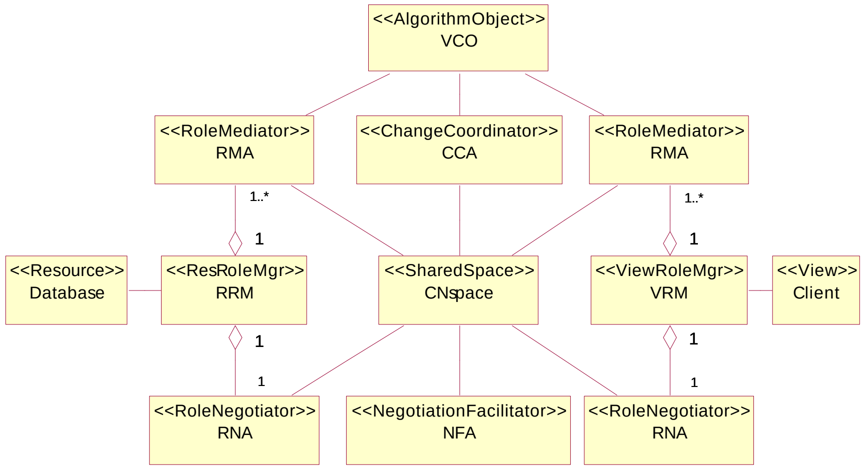


Fig. 2. Architecture To Support Adaptive View Coordination

3.1 Negotiation Reasoning

The model for negotiation reasoning used in our approach is based on the WinWin [2, 3] model used in requirements negotiation. In such a model, the participating agents collaboratively and asynchronously explore the WinWin decision space that is represented by four main conceptual artifacts. i) *WinCondition* - capturing the preferences and constraints of a participant. ii) *Issue* - capturing a conflict between WinConditions or their associated risks and uncertainties. iii) *Option* - capturing a decision choice for resolving an issue. iv) *Agreement* - capturing the agreed upon set of conditions which satisfy stakeholder WinConditions and/or capturing the agreed options for resolving issues. The object model for the WinCondition object developed for negotiating VCOs is shown in Figure 3. The object explicates attributes relevant to expressing preferences and constraints for the view coordination problem.

NAVCo incorporates three types of negotiation reasoning schemes that extend the WinWin model to consider a reactive model of negotiation. The first scheme, illustrated in Table 4, is used during the initial establishment of the task and subsequent negotiation of the initial policy. This scheme is triggered when a new WinCondition of a client is submitted. The WinCondition contains the task parameters and any client preferences and constraints. The negotiation facilitator agent then generates issues and the associated options and then sends the options to the client and resources role negotiation agents for evaluation. An agreement is reached and propagated to the change coordinator agent for implementation if all the role negotiation agents accept the option. Otherwise the client or resources can trigger further negotiation through the submission of revised WinConditions.

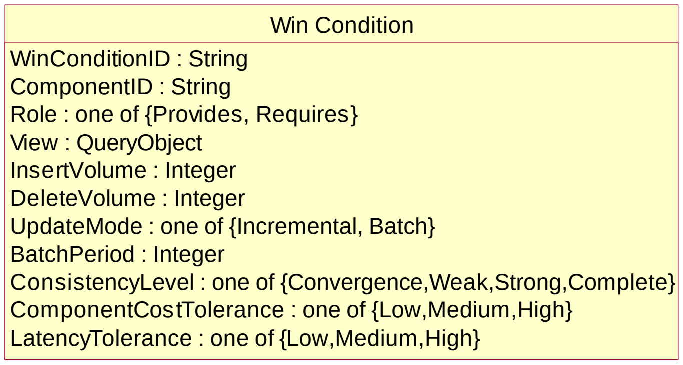

Win Condition
WinConditionID : String
ComponentID : String
Role : one of {Provides, Requires}
View : QueryObject
InsertVolume : Integer
DeleteVolume : Integer
UpdateMode : one of {Incremental, Batch}
BatchPeriod : Integer
ConsistencyLevel : one of {Convergence,Weak,Strong,Complete}
ComponentCostTolerance : one of {Low,Medium,High}
LatencyTolerance : one of {Low,Medium,High}

Fig. 3. The WinCondition Object Model

Table 4. Task Driven Negotiation Protocol

```
1. Client RNA submits a WinCondition to NFA. The WinCondition identifies the
task preferences and constraints of the Client
2. The NFA analyzes the posted WinCondition and identifies Issue(s)
3. The NFA generates potential Options that Resolve the Issue(s)
4. The Resource and Client RNAs evaluate the Option(s)
5. If  an option is accepted by all RNAs
     Then {Agreement = Accepted Option
            Agreement propagated to CCA for implementation}
   Else {Client and/or Resource RNAs post revised WinConditions
           Go To Step 2 }
   End If
6. If timeout_event received
   Then initiate priority driven protocol
   End If
```

The second scheme (illustrated in Table 5) is conflict-driven and is used for run-time dynamic renegotiation of policies. This scheme can be triggered by any client or resource through the submission of a revised WinCondition representing changing component preferences and/or constraints. The negotiation facilitator agent then analyzes the revised WinCondition against the current set of WinConditions and the current agreement to identify issues and associated options. If this results in an option other than the current agreement, a negotiation among the components ensues.

The third scheme is priority-driven and is triggered by the occurrence of a timeout_event during the execution of either the task-driven or conflict-driven protocols. NAVCo supports two priority-driven schemes, competitive and cooperative. In both schemes, a list of possible preferences is identified (e.g., x_1= complete consistency, x_2= low communications, etc.). Each component maintains a weighted list of the local preferences (e.g., $a_{i,j}$ is the weight assigned by component i to preference j). Each component is also assigned a global weight (e.g., w_1=1.0, w_2=0.95, etc.). Table 6 details the steps in the competitive scheme and Table 7 details the steps in the cooperative scheme.

Table 5. Conflict Driven Negotiation Protocol

```
1.  Resource or Client RNA submits a revised WinCondition to the NFA. The
revised WinCondition reflects a local change in preferences and/or constraints
2.  The NFA analyzes the revised WinCondition against the set of current
WinConditions to generate Issue(s) resulting from (pairwise) conflicting
interaction
3.  The NFA generates potential Options that resolve the Issue(s)
4.  If there is no change in Option (i.e., Option = current Agreement)
    Then {NFA marks the Issue as resolved}
     Else {Resource and Client RNAs evaluate the Option(s)
.             If an Option is accepted by all RNAs
              Then {Agreement = Accepted Option
                      Agreement propagated to CCA for implementation}
              Else {Client and/or Resource RNAs post revised WinConditions
                      Go To Step 2 }
              End If
      End If
5. If timeout_event received
   Then initiate priority driven protocol
   End If
```

Table 6. Competitive Priority Driven Negotiation Protocol

```
1. Each RNA generates a weighted list of local preferences
   (e.g., $a_{1,1}x_1$, $a_{1,2}x_2 \ldots a_{1,j}x_j$ for component number 1)
2. As part of WinCondition, each RNA submits top weighted preference to the
NFA (e.g., $a_{1,1}x_1$, may be submitted by component 1)
3. NFA applies global component weighting factors to submitted preferences
   (e.g., $w_1a_{1,1}x_1$ would be the weighting for component 1's preference)
4. The NFA selects the component preference with the highest overall weighting
   (i.e., the preference associated with $\max(w_ia_{i,j})$ is selected)
5. The NFA identifies the policy that most satisfies the selected preference
6. The selected policy is propagated to CCA for implementation
```

Given the above reasoning methods three major questions arise. 1) How do *issues* get generated? 2) How do *options* get generated? 3) How do *options* get evaluated?

The NAVCo approach exploits the context and the view coordination problem domain to address the questions as follows. a) Given one or more WinConditions, *issue generation* involves formulating a query to identify VCO specification objects that satisfy the WinConditions. Here the issue is formalized as a query object. b) Given the formulation of the issue, *option generation* involves evaluation of the query to retrieve plausible VCO specification objects and their refinements. c) Given the options, *option evaluation* involves checking for consistency of an option against a database of committed WinConditions.

Table 7. Cooperative Priority Driven Negotiation Protocol

1. Each RNA generates a weighted list of local preferences (e.g., $a_{1,1}x_1$, $a_{1,2}x_2 \ldots a_{1,j}x_j$ for component number 1)
2. As part of WinCondition, each RNA submits entire list of weighted preferences to the NFA
3. NFA applies global component weighting factors to submitted preferences
4. The NFA sums the weights associated with each preference (e.g., $(w_1a_{1,1} + w_2a_{2,1}+\ldots+ w_na_{n,}1)x1$ would be the cooperative sum for preference 1)
5. The NFA selects the preference with the highest overall cooperative sum
5. The NFA identifies the policy that most satisfies the selected preference
6. The selected policy is propagated to CCA for implementation

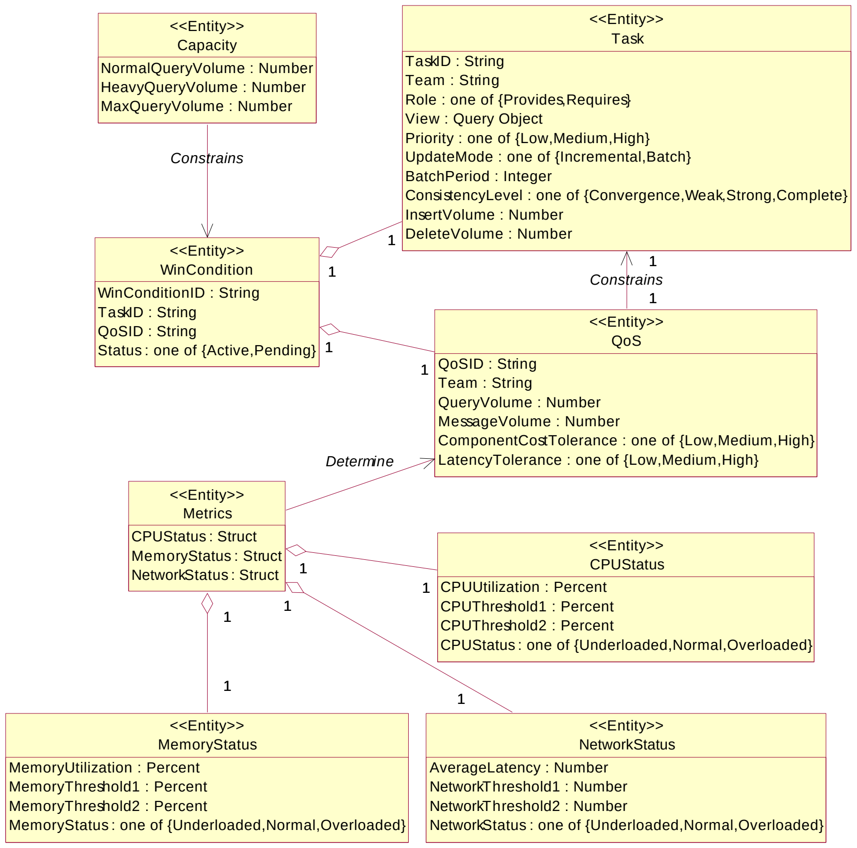
3.2 Models to Support Negotiated Selection of VCO

In order to support the reasoning approach outlined above, NAVCo requires the following. a) Declarative models of preferences and constraints at the clients and resources as a database of facts. b) Rules for issue generation, option generation, and option evaluation. We briefly describe below the data models and some examples of the rules that have been formulated and prototyped in our initial experiments.

The class diagram shown in Figure 4 captures the data model underlying the information maintained by the role negotiation agents of the clients and resources. The model in essence articulates the WinCondition as consisting of two parts. a) The t*ask part* is of type "provides" for a resource or of type "requires" for a client. The task part explicates the role to be played, prioritization of tasks, task preferences, and update volume and distribution submitted in support of the task. b) The *QoS constraint part* articulates the constraints imposed on the task. The QoS schema specifies the component workload to support the task and the component QoS constraints based on the status of component resources captured as QoS metrics. The data model also specifies global integrity constraints.

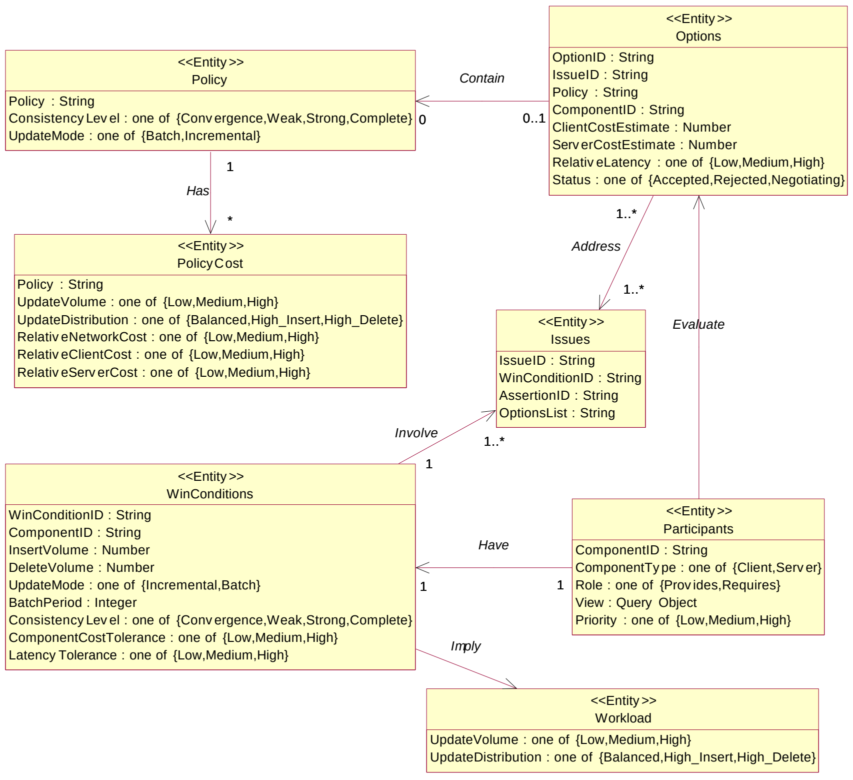
The data model specifying the content of the information in the negotiation facilitation agent is given in Figure 5. The data model captures VCO specifications and associated costs. The data model also contains models of the WinConditions, Issues, and Options that get posted or generated by the NFA. Some of the important data elements are a) identification, characteristics, and costs of available coordination policies, b) task-specific meta-data, and c) overall team-level workload characterization, preferences, and constraints.

The rules for issue and option generation and option evaluation are modeled as database trigger rules that analyze WinCondition updates to identify issues and options and to trigger option evaluations. The trigger rule in Table 8 creates an `Issue`, whose semantics is that of a query assertion to select a VCO policy, in the `Issues` table when there is an update to the `WinCondition` table. The rule accesses relevant constraints imposed by a task specific WinCondition that must be met by a VCO. A trigger rule for option generation, as shown in Table 9, adds entries to the `Options` table and is triggered by issue entries in the `Issues` table.



<<Entity>>
Capacity
NormalQueryVolume : Number
HeavyQueryVolume : Number
MaxQueryVolume : Number
Constrains
<<Entity>>
Task
TaskID : String
Team : String
Role : one of {Provides,Requires}
View : Query Object
Priority : one of {Low,Medium,High}
UpdateMode : one of {Incremental,Batch}
BatchPeriod : Integer
ConsistencyLevel : one of {Convergence,Weak,Strong,Complete}
InsertVolume : Number
DeleteVolume : Number
1
1
<<Entity>>
WinCondition
WinConditionID : String
TaskID : String
QoSID : String
Status : one of {Active,Pending}
1
1
Constrains
1
<<Entity>>
QoS
QoSID : String
Team : String
QueryVolume : Number
MessageVolume : Number
ComponentCostTolerance : one of {Low,Medium,High}
LatencyTolerance : one of {Low,Medium,High}
1
Determine
<<Entity>>
Metrics
CPUStatus : Struct
MemoryStatus : Struct
NetworkStatus : Struct
1
1
<<Entity>>
CPUStatus
CPUUtilization : Percent
CPUThreshold1 : Percent
CPUThreshold2 : Percent
CPUStatus : one of {Underloaded,Normal,Overloaded}
1
1
1
1
<<Entity>>
MemoryStatus
MemoryUtilization : Percent
MemoryThreshold1 : Percent
MemoryThreshold2 : Percent
MemoryStatus : one of {Underloaded,Normal,Overloaded}
<<Entity>>
NetworkStatus
AverageLatency : Number
NetworkThreshold1 : Number
NetworkThreshold2 : Number
NetworkStatus : one of {Underloaded,Normal,Overloaded}

Fig. 4. RNA Data Model of Preferences and Constraints



<<Entity>>
Policy
Policy : String
ConsistencyLevel : one of {Convergence,Weak,Strong,Complete}
UpdateMode : one of {Batch,Incremental}
Contain
0
0..1
<<Entity>>
Options
OptionID : String
IssueID : String
Policy : String
ComponentID : String
ClientCostEstimate : Number
ServerCostEstimate : Number
RelativeLatency : one of {Low,Medium,High}
Status : one of {Accepted,Rejected,Negotiating}
1
Has
*
<<Entity>>
PolicyCost
Policy : String
UpdateVolume : one of {Low,Medium,High}
UpdateDistribution : one of {Balanced,High_Insert,High_Delete}
RelativeNetworkCost : one of {Low,Medium,High}
RelativeClientCost : one of {Low,Medium,High}
RelativeServerCost : one of {Low,Medium,High}
1..*
Address
1..*
Evaluate
<<Entity>>
Issues
IssueID : String
WinConditionID : String
AssertionID : String
OptionsList : String
Involve
1
1..*
<<Entity>>
WinConditions
WinConditionID : String
ComponentID : String
InsertVolume : Number
DeleteVolume : Number
UpdateMode : one of {Incremental,Batch}
BatchPeriod : Integer
ConsistencyLevel : one of {Convergence,Weak,Strong,Complete}
ComponentCostTolerance : one of {Low,Medium,High}
LatencyTolerance : one of {Low,Medium,High}
Have
1
1
<<Entity>>
Participants
ComponentID : String
ComponentType : one of {Client,Server}
Role : one of {Provides,Requires}
View : Query Object
Priority : one of {Low,Medium,High}
Imply
<<Entity>>
Workload
UpdateVolume : one of {Low,Medium,High}
UpdateDistribution : one of {Balanced,High_Insert,High_Delete}

Fig. 5. NFA Data Model of Preferences and Constraints

Table 8. An Example of an Issue Generation Rule modeled as a Trigger Rule

```
TRIGGER <Issue generation> on INSERT into WinConditions
(INSERT into Issues (…)
    WHERE Issue.Assertion =
   (SELECT Policy
     FROM PolicyCost |x| Policy
WHERE UpdateVolume = WinCondition.UpdateVolume
AND UpdateDistribution =WinCondition.UpdateDistribution
AND ConsistencyLevel = WinCondition.ConsistencyLevel
AND UpdateMode=WinCondition.UpdateMode
AND RelativeClientCost < =WinCondition.ComponentCostTolerance))
```

Table 9. An Example of an Option Generation Rule modeled as a Trigger Rule

```
TRIGGER <VCO-Option-with-Evaluation>
on INSERT into Issues
(INSERT into Options (…)
WHERE Policy=Evaluate(Issues)
AND ClientCostEstimate=
[estimated client update volume based on policy and workload]
AND ServerCostEstimate=
[estimated server query volume based on policy and workload]
AND RelativeLatency=
(SELECT RelativeNetworkCost
FROM PolicyCost WHERE Policy=Issues.Option))
```

3.3 Mechanisms for Dynamic Switching of VCO

Figure 6 depicts an object collaboration diagram for the dynamic switching of view coordination objects. The event sequence is further elaborated below.

1. Once an option has been successfully negotiated the NFA writes a dynamic switching plan (DSP) into the CNspace. The DSP identifies the VCO that has been negotiated and includes a plan for dynamically switching between VCOs.
2. The CNspace sends a notification event to the CCA upon receipt of the DSP from the NFA.
3. The CNspace sends notification events to each RMA upon receipt of the DSP from the NFA.
4. Upon receipt of the notification event, the CCA reads the DSP from the CNspace and begins executing the plan.
5. Upon receipt of the notification event, each RMA reads the DSP from the CNspace and begins executing the plan. RMAs associated with resources begin queuing updates at this point.
6. The CCA sends a message to the current VCO to destroy itself. The VCO will continue to execute its algorithm until its queue of unprocessed queries is empty. At this point the VCO will send a return variable to the CCA indicating that it is about to destroy itself.
7. After the VCO is destroyed, the CCA using meta-data contained in its knowledge base will create a new VCO of the type indicated in the DSP. The meta-data contains task specific information to include the identity and location of team participants.
8. Upon instantiation, the VCO will bind itself to the RMI Registry.
9. The CCA writes a status event to the CNspace. The status event contains the name of the instantiated VCO.
10. The CNspace sends notification events to each RMA.
11. Upon receipt of the notification event, each RMA reads the status event from the CNspace.
12. Each RMA establishes a dynamic binding to the VCO through the use of the RMI Registry. Resource RMAs begin sending updates to the VCO.

4 Prototype

The adaptive view coordination architecture has been modeled using Rationale Rose 98 Enterprise Edition. Use cases, class diagrams, object collaboration diagrams, and sequence diagrams have been developed. Initial prototypes have been developed for both the negotiation and application views (layers). Prototypes for role mediator, role negotiation, change coordinator, and negotiation facilitator agents have been developed. Each prototype agent consists of a Java application and a Microsoft Access database.

All agent-to-agent coordination is accomplished through the use of the CNspace, which is implemented using JavaSpaces technology. WinConditions, options, dynamic switching plans and other objects are written as entries into the CNspace. The CNspace notify and read methods are utilized to route the entries to the appropriate agents. The prototype agents currently utilize input and output text files to simulate interactions with clients and resources. Initial results show that the NAVCo reactive reasoning methods can exploit the JavaSpaces based design environment to make negotiated decisions on the policy objects.

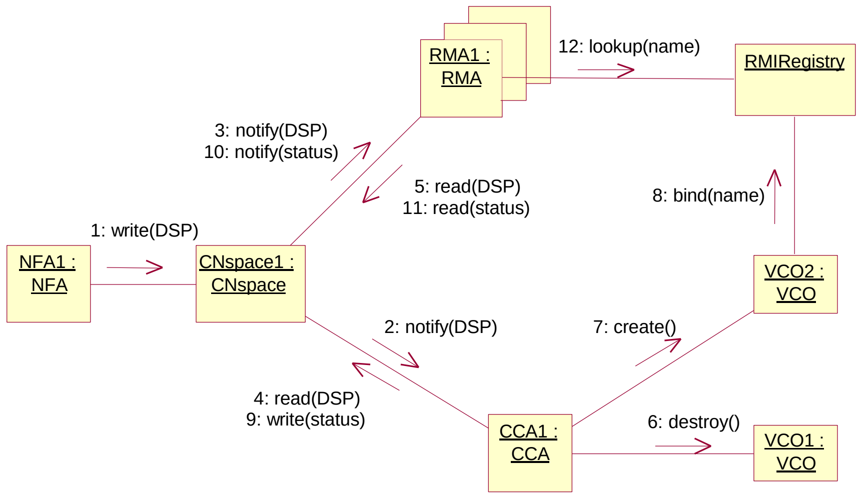


Fig. 6. Object Collaboration Diagram for Dynamic VCO Switching

5 Related Work

There has been a significant amount of work conducted in the area of view maintenance resulting in a spectrum of solutions ranging from a fully virtual approach where no data is materialized at one extreme to a fully replicated approach where full base relations are copied at the other extreme. These two extreme solutions are inefficient in terms of communications cost at one extreme and storage cost at the other extreme. Incremental view maintenance policies, such as ECA [26], Strobe [27], and SWEEP [1], are a hybrid of these two extremes. Incremental view maintenance policies maintain a materialized client view consisting of the relevant subset of data from the base relations at the data sources. Client decision-support applications, such as on-line analytical processing and data mining, then directly access the data contained within the materialized view at the client. The ECA family of algorithms is designed for a centralized database system, while the Strobe and SWEEP families are designed for distributed systems. Our research focuses on developing a self-adaptive architecture for distributed decision-support database systems. The NAVCo architecture described in this paper supports run-time policy changes between the Strobe and C-Strobe algorithms. The architecture is robust enough and can be scaled to support additional algorithms, such as SWEEP and Nested SWEEP.

The NAVCo work builds on the negotiation research performed by the community in requirements negotiation as well as automated negotiation. Negotiation is a complex and difficult area of active research pursued by researchers in different fields of study. Research progress has been made in different approaches to negotiation: a) Human factors approach where the major focus is understanding methods and techniques employed by humans to negotiate so as to manage the human factors of pride, ego, and culture [7, 19, 20]. The work on understanding people factors in requirements negotiation falls in this category. b) Economics, Game Theory and bargaining approach where research progress has been made on theoretical models of process driven negotiation [18], outcome driven negotiation, and self-stabilizing agreements to achieve some equilibrium [14]. Research on negotiation focuses on the group decision context where the power to decide is distributed across more than one stakeholder/agent as opposed to group decision making where a single decision maker relies on a set of analysts [15]. Two key aspects of the negotiated decision studied in most of the research are conflict and interdependence of decisions. Conflict has been used constructively in cooperative domains to explore negotiation options [3]. c) Artificial agents approach where the focus has been on developing computational agents that negotiate to resolve conflict [5], to distribute tasks [22, 24], to share resources [28], and to change goals so as to optimize multi-attribute utility functions [23]. In general, the models for agent cooperation and negotiation consider negotiation between multiple agents driven by global utility functions or independent local utility functions. The WinWin [2, 3] model used in NAVCo consider both types of drivers typical of negotiating teams having local preferences as well as global constraints.

The NAVCo approach is similar in spirit to the work on architecture-based run-time evolution [16]. Our approach and reasoning tools differ from [16] in terms of the nature of automation. The work in [16] focuses on providing a support environment where the necessary analysis for dynamic change and consequent operationalization can be performed. Automated switching based on automated negotiation reasoning motivates the NAVCo approach and prototype discussed in the paper.

The requirements for self-adaptive software included in Section 1.1 are derived from the DARPA "Broad Agency Announcement on Self-Adaptive Software" (BAA-98-12, December 1997). Due in part to the DARPA BAA the area of self-adaptive software has been an active area of research over the last several years [9, 10, 11, 13, 17, 21]. Most of the current research relies on the use of control theory to some extent [11]. A reflective agent architecture has been developed to support the run-time change of filters for the aerial surveillance domain [21]. The approach taken in [21] utilizes both reflection and control system theory. The use of control theory as a feedback loop for change reasoning was proposed in [10]. The control theory approach taken by others appears to be most applicable for embedded systems and domains with hard real time requirements. Since our domain of distributed decision-support database systems does not display these characteristics, we have taken a different approach based on negotiation reasoning followed by change coordination.

6 Summary and Future Work

This paper develops a Negotiation-based Adaptive View Coordination (NAVCo) approach for a class of distributed information management systems. The NAVCo approach allows view coordination to be dynamically adapted at run-time to meet changes in QoS preferences and constraints. The paper presents the key ideas and models developed and prototyped in our initial experiments with the approach. The key ideas of the NAVCo approach are as follows. a) A negotiation-based reasoning method for adapting view maintenance policies to meet dynamic changes in context (e.g., constraints). b) A dynamic software architecture of the collaborating information resources supporting the client task of maintaining a specific view. c) Coordination mechanisms in the architecture that realize negotiated changes in the policies for view maintenance.

This paper focuses primary on the negotiation-based reasoning models used in NAVCO and only briefly describes the NAVCo change coordination mechanisms. The change coordination mechanisms described in Section 3.3 rely on forcing the view coordination objects to a quiescent state prior to the dynamic switching. Our current work is focused on developing more sophisticated change coordination mechanisms that can gracefully transition on-going workload between view coordination objects without forcing the objects to a quiescent state. The main challenge is to ensure that the transitions support certain safety and correctness properties during and after the dynamic switching.

References

1. D. Agrawal, A. El Abbadi, A Singh, and T. Yurek, “Efficient View Maintenance at Data Warehouses,” In Proceedings of the ACM SIGMOD ‘97, pp. 417-427, 1997.
2. B. Boehm, P. Bose, E Horowitz and M. J. Lee, “Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach,” In IEEE Proceedings of the 17th ICSE Conference, 1995.
3. P. Bose and Z. Xiaoqing, “WinWin Decision Model Based Coordination,” International Joint Conference on Work Activities Coordination and Collaboration,” 1999.
4. W. R. Collins et. al., "How Good is Good Enough?" Communications of the ACM, pp. 81-91, January 1994.

5. E. H. Durfee, V. R. Lesser, and D. D. Corkill, "Cooperation Through Communication in a Distributed Problem Solving Network," In M. N. Huhns ed., Distributed Artificial Intelligence, Chapter 2, 29-58.
6. J. Farley, "Java Distributed Computing," O' Reilly Press, 1998.
7. R. Fisher and W. Ury, "Getting to Yes," Houghton-Mifflin, 1981. Also Penguin Books, 1983.
8. R. Hull and G. Zhou, A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 481-492, June 1996.
9. G. Karsai and J. Sztipanovits, "A Model-Based Approach to Self-Adaptive Software," IEEE Intelligent Systems, volume 14, number 3, May/June 1999.
10. M. Kokar, K. Baclawski, and Y. Eracar, "Control Theory-Based Foundations of Self-Controlling Software," IEEE Intelligent Systems, volume 14, number 3, May/June 1999.
11. R. Laddaga, "Creating Robust Software Through Self-Adaption," IEEE Intelligent Systems, volume 14, number 3, May/June 1999.
12. T. W. Malone and K. Crowston, "The Interdisciplinary Study of Coordination," ACM Computing Surveys, Vol. 26, No. 1, pp. 87-119, Mar. 1994.
13. D. J. Musliner, R. P. Goldman, M. J. Pelican, and K. D. Krebsbach, "Self-Adaptive Software for Hard Real-Time Environments," IEEE Intelligent Systems, volume 14, number 4, July/August 1999.
14. J. F. Nash, "The Bargaining Problem," Econometrica 28, pp. 155-162, 1950.
15. J. F. Nunamaker, A. R. Dennis, J. S. Valacich and D. R. Vogel, "Information Technology for Negotiating Groups: Generating Options for Mutual Gain," Management Science, October 1991.
16. P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based Runtime Evolution," In Proceedings of ICSE 1998.
17. P. Oreizy et. al., "An Architecture Based Approach to Self-Adaptive Software," IEEE Intelligent Systems, volume 14, number 3, May/June 1999.
18. M. J. Osborne and A. Rubinstein, "A Course in Game Theory," MIT Press, MA, 1994.
19. M. Porter, "Competitive Strategy: Techniques for Analyzing Industries and Competitors," Free Press, NY, 1980.
20. H. Raiffa, "The Art and Science of Negotiation,, Harvard University Press, Cambridge, MA, 1982
21. P. Robertson and J. M. Brady, "Adaptive Image Analysis for Aerial Surveillance," IEEE Intelligent Systems, volume 14, number 3, May/June 1999.
22. R. G. Smith, "The Contract Net Protocol: High-level Communication and Control in a Distributed Problem Solver," IEEE Transactions on Computers, 29, pp. 1104-1113, 1980.
23. K. P. Sycara, "Resolving Goal Conflicts Via Negotiation," In Proceedings of AAAI, pp. 245-250, 1988.
24. M. P. Wellman, "A General Equilibrium Approach to Distributed Transportation Planning," In Proceedings of AAAI-92, San Jose, CA 1992.
25. T. Winograd, "Bringing Design to Software," Addison Wesley Publishers, 1996.
26. Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehouse Environment. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 316-327, May 1995.
27. Y. Zhuge, H. Garcia-Molina, and J. Wiener, "The Strobe Algorithms for Multi-Source Warehouse Consistency," In Proceedings of the International Conference on Parallel and Distributed Information Systems, December 1996.
28. G. Zlotkin, and J. S. Rosenschein, "Mechanism Design for Automated Negotiation and Its Application to Task Oriented Domains," Artificial Intelligence, Vol. 86, pp. 195-244, 1996.