AFRL-IF-RS-TR-2006-258
Final Technical Report
August 2006

REAL-TIME DISTRIBUTED ALGORITHMS FOR VISUAL AND BATTLEFIELD REASONING

University of Maryland at College Park

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J020

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2006-258 has been reviewed and is approved for publication

APPROVED: /s/

RICHARD C. BUTLER, II
Project Engineer

FOR THE DIRECTOR: /s/

WARREN H. DEBANY, Jr.
Technical Advisor, Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
AUG 2006	Final	Sep 99 – Jan 06

4. TITLE AND SUBTITLE

REAL-TIME DISTRIBUTED ALGORITHMS FOR VISUAL AND BATTLEFIELD REASONING

5a. CONTRACT NUMBER

5b. GRANT NUMBER
F30602-99-1-0552

5c. PROGRAM ELEMENT NUMBER
62301E

6. AUTHOR(S)

V.S. Subrahmanian, Larry Davis, James Reggia, Victor Basili and John Aloimonos

5d. PROJECT NUMBER
J020

5e. TASK NUMBER
16

5f. WORK UNIT NUMBER
01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Maryland at College Park
Office of Research Administration and Advancement
2100 Lee Building
College Park, MD 20742-5141

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 N. Fairfax Drive
Arlington, VA 22203-1714

AFRL/IFGA
525 Brooks Rd
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2006-258

12. DISTRIBUTION AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 06-559

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Information is key to the success of the next generation battlefield. There is a critical need to determine, in real-time, what the enemy is doing, and to interpret that information in the context of past related events. In this project we examined two aspects of this issue: development of a high-level task definition language for tasking a network of sensors to carry out given objectives, and interpreting recounted events so that past related scenarios could be automatically identified from a case database.

15. SUBJECT TERMS

Distributed Sensor Networks, Sensor Tasking, High-Level Task Definition Language, Graphical User Interface (GUI), Story Analysis, Story Interpretation, SensIT Nodes

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Richard Butler
U	U	U	UL	77	19b. TELEPONE NUMBER (*Include area code*)

Abstract

Information is key to the success of the next generation battlefield. Whether a combat operation is taking place in the ocean, or in the desert, or in an urban terrain, there is a critical need to determine, in real-time, what the enemy is doing, and to take appropriate actions based on knowledge of the enemy's movements. We examined two aspects of the critical need for battlefield information in this work: sensor tasking and automated interpretation of narrated events.

Analysts who wish to task a network of sensors to carry out their objectives should not have to understand the details of sensors in order to specify their tasks. In section I, we proposed a high level task definition language, which is rich enough to allow an analyst to (i) specify a region of interest to him, (ii) specify a time frame of interest, (iii) specify a set of conditions that the analyst wants monitored during the time frame and in the region above, and (iv) specify one or more actions to be taken in the event the conditions are satisfied. We developed graphical user interfaces that may be used to express such task specifications and developed algorithms to scale such systems.

The understanding of recounted events (stories, histories, etc.) so that one can determine the best course of action is a common and critical part of decision making in many areas, including anti-terror intelligence assessment and military operational command. The goal of the second part of this project was to identify aspects of story representation and analysis that can be readily implemented using existing technology. As described in Section 2 of this report, we assessed the availability of narratives in domains such as urban warfare and software engineering as well as the suitability of software tools for story analysis. A prototype story interpretation system that integrated rule-based and case-based processing was implemented and evaluated in the context of urban warfare stories. We conclude that automated story interpretation is feasible in targeted domains and that general purpose tools for this task can be developed, but a significant stumbling block to future progress is the limited availability of online story repositories.

Table of Contents

Table of Contents

List of Figures

List of Tables

Acknowledgements

We thank Sri Kumar at DARPA for his incisive comments throughout this effort. Dave Shepherd assisted us throughout the process of working with other contractors on the Sensor Information Technology (*SensIT*) effort. We worked closely with Gail Mitchell and Prakash Mangwani at BBN, Tom Hammel and Bernie Yetso of Fantastic Data - we thank them for their prompt response to our needs. In addition, we worked with numerous researchers who formed other vital parts of the *SensIT* team including Philippe Bonnet and Johannes Gehrke of Cornell, Mark Jones and Jae Park from Virginia Tech, Joe Reynolds, Steve Beck, and many others from BAE Systems, University of Tennessee, University of Wisconsin, and Pennsylvania State University.

Section 1: Tasking a Network of Sensors

1.1 Summary

Analysts who wish to task a network of sensors to carry out their objectives should not have to understand the details of sensors in order to specify their tasks. We proposed a high level task definition language, which is rich enough to allow an analyst to (i) specify a region of interest to him, (ii) specify a time frame of interest, (iii) specify a set of conditions that the analyst wants monitored during the time frame and in the region above, and (iv) specify one or more actions to be taken in the event the conditions are satisfied. We developed graphical user interfaces that may be used to express such task specifications and developed algorithms to scale such systems. We worked jointly with several members of the Sensor Information Technology (*SensIT*) team to develop a system that could be demonstrated on live sensor data, both from 29 Palms and from BBN's Waltham test bed.

1.2 Introduction

The DARPA Sensor Information Technology (*SensIT*) program has proposed the use of large ad-hoc networks of sensors to collect information about a current or potential battle zone. As sensors get smaller and smaller, the ability to deploy such large networks of sensors from the air gets more and more probable.

However, interacting with sensors today is a challenging task. Software support for tasking and processing information collected from sensors is in its infancy. In order to interact with sensors, an analyst or an end-user needs detailed information about the sensor hardware, and often needs detailed information about the sensors' Application Program Interface (API for short). There are many reasons why interacting with a network of sensors in such a way is a bad idea:

1. First and foremost, analysts and mission planners are not software programmers or computer experts. They are experts in analyzing intelligence and fighting wars. Expecting them to program software code, even in so called high level programming

languages like *Java*, is unreasonable - expecting them to program sensors that typically have much lower level programming languages is truly foolhardy.

2. Second, diverse sensors have very diverse programming interfaces. For example, temperature and pressure sensors are very different from motion sensors which, in-turn, are very different from image and video sensors. Even if an analyst or war fighter were to learn the APIs of one sensory source, this would be far from adequate as different sensors have varying APIs. Placing such a burden on an analyst/war fighter would leave him in a position where he could not focus on his primary task - that of successfully prosecuting a war.

3. Third, even if we had someone learn how to use APIs of all these diverse sensory devices, we would still be left with the challenge of harnessing all the information coming back from sensors. The ability to "mix and match" data coming back from diverse sensors is a major challenge. Expecting even a professional programmer to program these tasks one by one has the following problems:

- **Cost.** Programming each tasking request is a costly endeavor.

- **Time.** Programming each tasking request will inevitably lead to a significant delay between the time the analyst/war fighter makes the tasking request and the time the request is programmed in. This delay may be acceptable in applications such as field-testing a road vehicle (where the gap between the time the tests are conducted to deployment can run into years), but is totally unacceptable in the military setting where time is of the essence. Delays lead to potential casualties.

- **Lack of flexibility.** Hard coded solutions, where each task request is explicitly programmed, are inflexible. Suppose a task was programmed in *C* or *Java*. It is inevitably the case that task requests get changed over time. For instance, a request to monitor the road leading to a potential Iranian nuclear facility might need to be abruptly changed when it is noticed that there is an inordinate amount of helicopter traffic in the region. Retasking would require reprogramming. However, reprogramming programs written in modern programming languages is difficult. The

person doing the reprogramming needs to understand all the existing code - a major challenge. Even if the programmer thinks he has understood it, he may be wrong, leading to a reprogrammed version of the task that does not quite do what it is supposed to do. This in turn might have catastrophic results.

4. Fourth, when analysts create tasks, those tasks involve not just accessing the sensor data, but also correlating that data with non-sensory sources. For example, an analyst correlating data about movement of enemy forces might want to pipe that data into a prediction program, which in turn might need to access an intelligence database showing enemy assets in the region. The high level task that the analyst is interested in is not just an examination of the low level sensor data - rather, he wants that sensor data analyzed in a manner that he specifies and he wants the results of that "processed" sensor data to be the result of his task request.

The above points make it clear that a system that enables analysts to task a network of sensors needs to satisfy many fundamental criteria.

1. **Task definition.** First and foremost, the notion of a task must be formally defined. We all know informally what a task is. However, for a computer system to find ways of accomplishing war fighters' tasks, such a formal definition is required.

2. **Task definition language.** The system needs a formal language in which tasks can be articulated - such a language must be declarative in the sense that the war fighter merely needs to specify what the task should accomplish, not how the task should be accomplished. In short, the analyst might merely say that he wants to monitor levels of traffic in and around the Al-Natan suspected nuclear facility. He shouldn't have to tell the system to turn sensor 1 on, orient it 45 degrees SW, set it to low battery consumption and do similar things with sensors 2 through 100.

3. **Tasking GUI.** Even though declarative languages are far more easy to program and modify than imperative languages like *C* and *C++*, they still require a learning curve for the war fighter - time that is not the best possible use of the war fighter's unique skills. A tasking graphical user interface (GUI) which is easy to use and understand is critical. By

interacting with such a GUI, the war fighter or analyst specifies a task in the task definition language (even though he may not be aware of it).

4. **Task Engine.** The job of the task engine is to take one or more tasks and arrange for their execution. This is done by determining which of many possible diverse resources can be used to perform the task at hand. Given that hundreds of tasks may be executed concurrently (for the same or for different war fighters), the ability to allocate tasks to sensors that have the required capabilities in a way that maximally supports all the tasks is needed.

5. **Task Merging.** Scaling the task engine requires the ability to study how to merge tasks together when it is clear that the tasks share a common component. For example, if we wish to monitor traffic on two roads, we may wish to place a sensor at the intersection of the two roads rather than place one on each as this reduces the overall amount of resources (sensors) used. Of course, this must be done judiciously (e.g. by examining the road network, etc.).

The work we have performed as part of the DARPA *SensIT* program includes the following.

1. We have proposed a formal definition of a task.

2. We have proposed a declarative tasking language in which tasks may be expressed.

3. We have developed a graphical user interface through which analysts may express tasks in the above declarative tasking language.

4. We have developed algorithms to implement the Task Engine.

5. We have developed algorithms for Merging tasks.

6. We have conducted experiments assessing the efficiency of many of these methods.

7. We have developed significant software components that were used during the *SensIT* experiments and that were used to build an integrated demonstration (jointly with other

SensIT contractors such as BBN, BAE Systems, Fantastic Data, Penn State University, Cornell University, Virginia Tech., to name a few).

The rest of this report will provide further details about these individual contributions as well as the methods we used, assumptions we made, and procedures we followed.

Note. Our original proposal also included several tasks related to image and video processing. However, those portions were not funded by this program.

1.3 Methods, Assumptions and Procedures

The following pieces of work were done internally at the University of Maryland.

1. Definition of a task.
2. Description of a declarative tasking language in which tasks may be expressed.
3. Algorithms to implement the Task Engine.
4. Algorithms for Merging tasks.
5. Experiments assessing the efficiency of many of these methods.

In this section, we describe the methods, assumptions and processes we put in place to develop the scientific underpinnings of these components. The scientific results, a description of our demonstrations and experiments, and a discussion of the process of joint experimentation and demonstrations are described later in Section 4.

1.3.1 Task Definition

Though several abstract definitions of the task exist in the AI planning literature, none of them is adequate for the purposes of tasking a network of sensors, because tasking networks of sensors is not the same thing as an AI plan.

We approached the problem of task definition as follows:

- First, we came up with a set of sample tasks that an analyst/user may wish to specify that require sensing capabilities.

- Second, we examined all these sample tasks with a view to determining whether they shared any commonalities. In other words, is there a common abstraction that captures all these diverse tasks as special instances of the abstraction?

- Third, we analyzed the abstraction for possible shortcomings and/or limitations.

Our abstraction of a task had four major components.

1. Where should the task be performed? This component specifies a geospatial region where the task is to be performed.

2. When and how frequently should the task be performed. Some tasks need to start now and be continuously running forever, while other tasks may have a finite time horizon, with updates on the task required only every hour.

3. What should the task look for? This component specifies whether the task is looking for certain vehicle activity patterns somewhere or whether it is looking for acoustic patterns or something else altogether.

4. What should the task do when it finds what it is looking for. For example, if the task finds that enemy activity near the Al-Natan nuclear site is dramatically increasing, should it just notify the analyst? Should it correlate the sensor readings with background intelligence data, analyze it using a threat module, and send the resulting fused knowledge to the analyst?

We came up with a formal mathematical definition of a task that incorporated the above four components and we verified that this mathematical definition can capture the example tasks we had started out with.

1.3.2 Declarative Sensor Tasking Language

Clearly, the definition of a language for tasking sensors depends upon our definition of a task. As our notion of a task had four components - what to monitor, when to monitor, where to monitor, and what to do when the task monitoring condition is satisfied, we needed to make the following choices.

- **Choice of a geospatial region definition:** There are many possible ways of specifying geospatial regions. We assumed that all geospatial regions of interest would be rectangles with two horizontal and two vertical sides.

- **Choice of a temporal specification:** Again, there are many possible ways of specifying times of interest. In the types of sensor tasking examples we saw, we chose to specify time periods via a triple - a start time (for the monitoring) an end time (for the monitoring) and a frequency describing how often the task condition should be evaluated.

- **Task condition:** The hardest part of our research was determining how to specify conditions to be monitored. This is because the conditions being monitored could span multiple sensory nodes as well as, potentially, third party data sources. We showed how the notion of a *code call condition*[12] can be used to specify task conditions.

- **What actions to take when task conditions are satisfied:** There are many choices in specifying how to react to a task monitoring condition being satisfied. We examined some alternative possibilities and decided that the criterion of having a declarative, easy to modify specification was best met with rules of the form "if ⟨*task condition*⟩ then DO ⟨action⟩". The key therefore was defining a syntax for task conditions.

1.3.3 Algorithms to implement Task Engine

The job of the task engine is to determine which sensor will execute which task or subtask. In order to execute a task or a subtask, the sensor in question must be at the right place at the right time and furthermore, should have the right capabilities. In addition, the load on the sensor should not be so high that it cannot satisfy other tasks it has previously been charged to perform.

Our approach to this problem was as follows.

- First, we came up with a mechanism to define what tasks each individual sensor can perform and how those tasks can be accessed by external programs.
- Second, we came up with a formal definition of a Task Assignment Problem (TAP for short). This formulation of a TAP assumes the existence of a model of load on the network, but it does allow different tasks to take varying amounts of time.
- Third, we analyzed the complexity of the TAP problem using computational complexity theory. This allows us to assess the complexity of the problem independently of any specific algorithm (of course an algorithm is good only if it falls within the complexity classification of the problem).
- Based on the above analysis, we developed alternative algorithms that could be used to task a network of sensors.
- Finally, we developed heuristic algorithms to solve this problem as well. These algorithms may not find an optimal way of assigning sensors to solve a given problem, but they are guaranteed to run much more effectively than an algorithm that attempts to solve the combinatorial optimization problem involved in solving the TAP exactly.

1.3.4 Algorithms for Task Merging

A single user or analyst can generate numerous tasks, which in turn can generate numerous subtasks. Given a set of tasks $\{t_1, \ldots, t_n\}$ to perform, one way is to

sequentially execute the tasks in some order. Unfortunately, this may not be the most efficient way of executing the tasks. For example, we may have two tasks:

1. Task t_1 counts the number of vehicles going in or out of the suspected Al-Natan nuclear plant.

2. Task t_2 counts the number of Heavy Equipment Transports (HET) going in or out of the suspected Al-Natan nuclear plant.

Clearly, it is possible to merge both these tasks into one task – every time a vehicle is detected, increment the vehicle count by 1 and if the vehicle is an HET, then increment the HET count by one as well.

Our work in this topic included the following.

- First, we studied existing algorithms for merging activities. People have studied the possibility of merging activities in many fields (e.g. designers of disk servers try to merge requests to read addresses on disk to reduce the search time on disk; designers of multimedia video servers try to service multiple clients by merging them into one stream and so on). We determined that the algorithms closest in spirit to what we wanted to do were on Multiple Query Optimization (MQO) in databases.

- Our first step was determining whether algorithms used for multiple query optimization in databases worked. Unfortunately, we quickly determined that the optimization algorithms themselves take a prohibitive amount of time to run.

- Our experiments seemed to indicate that algorithms based on the A^* algorithm would work fine as long as only a relatively small number of queries (under twenty, but preferably near ten) were being merged.

- As a consequence, we were forced to consider alternative strategies. Suppose we had N task conditions to evaluate and only a maximum of M (where M is much smaller than N) task conditions could be merged efficiently. We considered the idea of splitting our M requests into at least $\lceil M/N \rceil$ buckets (there can be more buckets than

this) such that each bucket contains N or fewer task conditions. The idea is that the conditions inside a bucket are all similar and/or share common computations. By grouping together similar task conditions in one bucket, we hope to get a big savings in merging the items inside that bucket. By having multiple buckets, each being merged, we wanted to derive a significant savings compared to the case where the task conditions are independently evaluated.

- We first designed algorithms to optimally split the set of N task conditions into such buckets.

- We then analyzed the complexity of this problem and realized it was Non-deterministic Polynomial-time hard (NP-hard).

- As a consequence, we later developed heuristic algorithms for this problem.

1.3.5 STM System

One of the key components of our effort was to build a Sensor Task Manager (STM) implementing the theory developed in the work described in the preceding sections. The STM is a collection of modules documenting this effort.

The STM modules themselves involve the design and implementation of a wide variety of state of the art algorithms - these are described in full detail in Section 4, along with a rationale for why they were needed and who used them. In this section, we merely list the key modules.

1. Task specification language module.

2. Task Generation library development.

3. Cougar communications development conduit to work with Cornell University's (another *SensIT* contractor) Cougar system.

4. Simulation database. This was used to run experiments.

5. Sensor Network Gateway Development - this was used to talk to the sensor network.

6. *SensIT SH4* Gateway Task and Action Module. This was used to communicate with the underlying SensIT data query processes.

7. Imager control, manual trigger (*SH4*) module development. This component was used to interface with Sensoria's imager nodes.

8. In-Field alternate track generation support extension work. This was used to generate coerce tracks from target detection data.

9. In-Field external imager trigger prediction support.

10. Cache-borne task specification table development. This was used to specify tasks in cache.

11. *OpenMap SensIT* GUI. UMD developed a graphical user interface using tasks which could be specified by analysts/war fighters. We built this GUI using BBN's *OpenMap* graphical information system (GIS) library.

12. *OpenMap* GUI applet development. This was developed in order to support use of our *OpenMap* based GUI over the web by other *SensIT* contractors.

13. Modified classification code and shape display. This was used to render target shapes in the GUI.

14. Track history GUI line and color display. This component showed tracks aggregated from detections data on the GUI.

15. Cache-borne codebook table development. The codebook database stores the entity/value representations for the underlying *SensIT* network (i.e. 105 = "M1-A Tank", or 23 = "Passive InfraRed sensor"). This component implemented the network codebook internally (in-cache), allowing client interface functionality without the previously required external database/network connections.

16. *OpenMap* GUI imager query display.

1.4 Results and Discussion

In this section, we describe, in greater detail than we have done thus far, the key results underlying our approach to querying and tasking sensor networks. Over the almost four years of funding received under the *SensIT* program, we addressed and solved many problems related to the theory and implementation of systems to task sensor networks. In this section, we present one subsection for each problem encountered. These range from theoretical problems to practical implementation and/or demonstration problems. For each such problem, we present the following:

1. **Problem statement:** First, we present a statement of the problem. This is usually done in English. Where appropriate, some mathematical notation and/or figures may be used.

2. **Problem solution:** We then describe our solution to the problem. Again, this is done in English rather than in mathematical terms or in hardcore algorithmic terms. Where appropriate, we discuss the pros and cons of the solution.

3. **Collaborators:** We describe with whom we collaborated.

4. **Users:** We list those (in *SensIT*) who have used the research.

5. **Discussion:** Where appropriate, we list additional remarks/discussion.

1.4.1 Task definition

Problem Statement: Despite the fact that users have a good idea of what constitutes a sensory task, we had not previously come across a definition of a sensory task. Our problem was to define a sensory task appropriately, given the context of the *SensIT* program.

Problem Solution: Our solution to this problem involved several steps.

1. First, we had extensive discussions with our team members as well as selected *SensIT* participants in order to discern what people in the *SensIT* community thought a task was.

2. Several example tasks were proposed during the above discussions.

3. Based on these discussions, we defined a task to be a 5-tuple (*Map, Rect, Time, Cond, Act*) where:

- *Map* is the id of some map;
- *Rect* is a 4-tuple (*llx, lly, urx, ury*), where (*llx, lly*) represent the lower left corner of a rectangle and (*urx, ury*) represents the upper right hand corner of a rectangle. The rectangle itself has sides parallel to the x and y axes (i.e. the rectangle cannot be "crooked.").
- *Time* is a triple (*st, et, fr*) where st is the start time, *et* is the end time, and *fr* is a frequency. Intuitively, this says that the time frame during which the task is "active" is from *st* to *et*, and that the task condition must be checked at least every *fr* units of time.
- *Cond* is a condition that is expressed in some syntax and which allows the user to express conditions spanning multiple sensors and/or multiple data sources.
- *Act* is a set of condition/action pairs. Intuitively, if *Cond* is true, and if the pair (*C,A*) is present in *Act*, then we take action *A* if some further condition *C* is true.

Collaborators: The work on this was done by T.J. Rogers and V.S. Subrahmanian.

Users: A version of this work was implemented in the Sensor Task Manager, components of which were used by

- British Aerospace (BAE)/Nashua
- BBN Technologies (BBN)
- Fantastic Data (FData)
- Penn-State University (PSU)

- University of Tennessee (UTK)
- University of Wisconsin (UWI)
- Virginia Tech (VTech).

In the rest of this report, we will often use the term "*Group of 8"* to refer to this set of 7 *SensIT* contractors plus the University of Maryland.

Discussion: There is one simplifying assumption in the above definition, namely that users can only specify rectangular regions of interest. However, this is not a big limitation. If the user wants to specify interest in a single point on a map, he can set *urx = llx* and *ury = lly*. If the user wants to specify a non-rectangular region, he can usually approximate it via a set of rectangles. In addition, in STM, we have access to *OpenMap* which allows zooming in and out. This, combined with the possibility of using a set of rectangles, prevents the requirement that all regions of interest be rectangular, from being particularly restrictive.

1.4.2 Backend Task Definition Language

Problem Statement: The notion of a task, as described above, is extremely complex. Though the *Rect* and *Time* components are easy to represent computationally, task conditions and actions pose a major challenge. How can we write task conditions that span the data coming from multiple sensory devices and correlate those with data from non-sensory background sources?

Problem Solution: We considered a number of alternative solutions. First, there has been a huge amount of work in sensor fusion (such as that of LSU and PSU in the *SensIT* program). In general, sensor fusion aims at integrating analog signals from sensors. However, analysts cannot be expected to write queries involving analog signals. Based on the types of sensor nodes picked for the *SensIT* program, we realized that the type of sensor nodes available to *SensIT* all had Application Program Interfaces (APIs). APIs are sets of functions that can be invoked by external programs.

This proved key to the realization that the mechanism of *code call conditions*[5,12] can be used to express task conditions across multiple diverse sensory devices. Our STM uses the concept of a code call condition, scaled back somewhat to allow easy articulation of task conditions by war fighters and/or analysts (using a GUI) to express task conditions. In short, rather than using sensor fusion methods directly, we found that accessing sensor APIs was more convenient for tasking sensor networks.

Collaborators: This work was done by T.J. Rogers and V.S. Subrahmanian at the University of Maryland.

Users: A version of this work was implemented in the STM program. All Group of 8 members implicitly used this technology (though it was transparent to them).\

1.4.3 Task Merging

Problem Statement: Given a set of tasks that need to be performed by a single sensor, is there a way to integrate the tasks together so that the load on the sensor is relatively low?

Problem Solution: In view of the fact that code call conditions can be used to express task conditions, we were able to show that *any* algorithm for heterogeneous Multiple Query Optimization (MQO) can be used to merge task conditions. There are several such algorithms.

We conducted detailed experiments to see which of these techniques we should use. First, we studied variations of Sellis' MQO algorithm. It turned out that these algorithms did not scale very well to tasks even on simple relational databases. As a consequence, we concluded that using Sellis' MQO algorithm would *not* work for effective task merging.

In addition, we developed methods to bucket sets of tasks based on spatio temporal nearness. For this, we developed a heterogeneous spatial relational algebra. Given a region on the ground that the analyst is interested in, this algebra can be used to determine which sensors are in the region in question, as well as which sensors are near in a temporal sense.

Collaborators: Marat Fayzullin, V.S. Subrahmanian, and T.J. Rogers.

1.4.4 Task partitioning

Problem Statement: As we saw from the preceding subsection on task merging, standard MQO algorithms for task merging just don't work effectively. As a consequence, we were faced with the problem of how to achieve task merging effectively.

Problem Solution: Our solution to this problem was one of the major contributions of this research. We noticed that if we have N task conditions to merge, but at most M task conditions can be merged efficiently (this means that merging pays off because the time to merge plus the time to evaluate the merged task conditions is less than the time required to sequentially execute the task conditions), then we can try to create a set of "buckets" each of size M or less. The idea is that each bucket contains tasks where merging pays off *handsomely*, i.e. the savings realized by merging the tasks inside the bucket are substantial. The problem then was to determine how best to place the tasks inside the buckets. Our solution to this problem included the following novel contributions.

1. First and foremost, we developed a formal statement of the task partitioning problem using operations research methods in conjunction with the concept of code call conditions.

2. Next, we proved that the task partitioning problem is Non-deterministic Polynomial-time complete (NP-complete). This proof shows that the well known multiple knapsack problem can be reduced in polynomial time to the task partitioning problem. NP-complete problems are widely viewed in computer science as being impossible to solve efficiently (proving this formally is arguably the single most challenging open problem in computer science today).

3. This negative result caused us to rethink and re-evaluate our strategy. Should we really use bucketing methods like this? Or should we try something else?

4. We decided to go ahead and consider solutions to the operations research formulation of the problem that are *suboptimal*, but can be computed very fast, as well as solutions that find an *optimal* task partitioning, but are slow (because of the NP−completeness of the problem rather than because of any fundamental problem with the algorithm).

5. We first came up with a variation of the *A** search algorithm, called ***A*-based***, to solve the optimization problem *exactly*.

6. In addition, we came up with a *Branch And Bound*, called ***BAB*** algorithm, to solve the problem exactly.

7. After this, we came up with the unique notion of a *cluster graph*. The basic idea behind a cluster graph is that given a set of tasks, we create one node for each task in the set. We draw an edge between two nodes, if merging the two tasks leads to a savings. If so, we label the edge with the savings.

8. We showed that cluster graphs can be computed very fast, especially if we choose to estimate the savings.

9. We then developed a number of *greedy algorithms* to efficiently process cluster graphs for computing partitions of task sets (i.e. to decide which tasks go into which bucket). These greedy algorithms were called:

- ***Greedy-Basic:*** A basic greedy algorithm manipulating cluster graphs.
- ***Greedy-WU:*** Greedy with weight-update, which updated weights after each iteration.
- ***Greedy-NMA:*** A greedy algorithm which did not allow tasks to be moved from one bucket to another.
- ***HillClimb:*** A hill climbing algorithm.

10. Once we finished this, we decided to run experiments to assess the scalability of these approaches.

In the first experiment, we compared the running time of our algorithms (Table 1 lists the results). It turns out that *A*-based* ran out of memory, when the number of activities exceeded 10, and ***BAB*** ran out of memory, when the number of activities exceeded 11. The reason for this is that the OPEN list maintained by *A** quickly grows overwhelmingly large. Hence, the tables only show results for 11 activities. Although the *A*-based* algorithm is faster than ***BAB***, it runs out of memory faster. Both algorithms have much longer running times compared to the heuristic-based algorithms. This is expected, as the problem is NP-hard, and both *A*-based* and ***BAB*** algorithms find optimal solutions.

Table 1: Running times of the Algorithms (in milliseconds).

No of Activities	A*-based	BAB	Greedy-Basic	Greedy-NMA	Greedy-WU	Hill Climb
5	17.24	19.84	5.28	5.04	7.92	6.44
6	53.4	58.26	6.46	6.2	10.72	8.54
7	142.72	158.44	8.2	7.58	14.84	11.02
8	479.02	503.5	10.02	9.4	19.26	13.8
9	1431.7	1657.5	11.94	11.56	25.14	17.16
10	4189.44	3432.38	15.68	13.8	31.4	20.92
11		4149.56	18.7	16.14	38.12	24.96

As the heuristic algorithms were very fast, we increased the number of activities to be partitioned. We first fixed the overlap degree and overlap probability constant at 0.2, and ran two experiments. In the first experiment, we kept the number of activities within a cluster constant at 25, and in the second, the number of activities within a cluster was 50.

In this experiment, the ***HillClimb*** algorithm ran out of memory around 400 activities and did not scale up very well. Recall that the ***HillClimb*** algorithm uses the same expansion function as the *A*-**based*** algorithm, but keeps the best sub-partition at each node. When the number of activities in the input set exceeds 350-400 activities, the algorithm generates too many children nodes and runs out of memory. Therefore, we only show the scalability results for the greedy algorithms. This result suggests that the cluster graph representation is a very effective heuristic.

When the number of activities within a cluster is 25, the running times of the algorithms are very close, and they handle 1000 activities in about 140 secs. On the other hand, when the number of activities within a cluster is increased to 50, the execution time of the ***Greedy-WU*** algorithm is much larger than that of the other two greedy algorithms. This is expected because the ***Greedy-WU*** recomputes edge weights after each iteration. Moreover, all algorithms run much slower in the second experiment.

As the greedy algorithms may compute suboptimal solutions, we want to see how much "worse" the solutions produced by these algorithms were compared to the optimal solution. We used the following metric to compare the quality of plans produced by the algorithms. When we have a set **A** of activities, and a partition **P** of **A**, the cost reduction percentage realized by **P** is given by:

$$\frac{\sum_{a \in A} c(a) - \sum_{A_i \in P} c(A_i)}{\sum_{a \in A} c(a)} \quad (1)$$

The rationale behind using percentage savings instead of absolute numbers is the fact that we have simulated the cost estimation function.

We first wanted to see how our savings were affected by changes in the amount of overlap between activities.

In general, the solutions produced by the ***Greedy-Basic*** algorithm are about 0.8–13% worse than the optimal solutions. The solutions produced by the ***Greedy-NMA***

algorithm are 0.8–20% worse, and the ones produced by the ***Greedy-WU*** algorithm are 0.06–9.4% worse than the optimal solutions. The quality of the solutions produced by the ***HillClimb*** algorithm is not as good as the greedy solutions, and they are about 2.4–23% worse than the optimal solutions. As we increase the overlap degree in the input activity set, the difference between the savings generated by the heuristic algorithms and the ones computed by the optimal algorithms also increases.

Although we could not get the optimal solution after 11 activities, we still ran the greedy algorithms up to 1000 activities to see the quality of partitions they produced. We observed that the performance of the greedy algorithms does not degrade, and stays about the same as we increase the number of activities in the input sets. The ***Greedy-NMA*** algorithm produces the worst results as it explores the smallest number of alternatives when generating partitions. The quality of the partitions produced by the ***Greedy-Basic*** and the ***Greedy-WU*** algorithms are comparable. When the number of activities within a cluster is smaller, and hence there are fewer edges in the cluster graph, the ***Greedy-Basic*** algorithm does better than the ***Greedy-WU***. However, when the number of activities within a cluster increases, the ***Greedy-WU*** algorithm catches up with the ***Greedy-Basic*** algorithm.

These results show that although the ***Greedy-WU*** algorithm updates weights after each iteration, and hence uses more accurate cost saving estimates, it still uses a greedy strategy, and is not able to capture the maximal savings. Recall that the ***Greedy-WU*** algorithm does not move activities around once it inserts them into components. The ***Greedy-Basic*** algorithm on the other hand, tries to adapt its savings estimates as the computation of the output partition proceeds by moving activities across components. It turns out that this heuristic is more effective in capturing the real savings than in updating weights. Hence, ***Greedy-Basic*** produces higher quality partitions. Moreover, its running time is better than ***Greedy-WU***. As a result, we believe that ***Greedy-Basic*** is the best choice among the three.

Both *A**-***based*** and ***BAB*** are unable to perform at all, when there are more than 10-20 concurrent activities. On larger sets of activities, they both quickly run out of

memory — in contrast, the greedy algorithms scale up very effectively when many activities occur. Furthermore, the ***HillClimb*** algorithm also does not scale up very well, and produces worse quality results than the greedy algorithms. Therefore, the ***Greedy–Basic*** algorithm is also the best algorithm among the heuristic based algorithms.

Collaborators: Fatma Ozcan, V.S. Subrahmanian

1.4.5 Assigning sensors to tasks

Problem Statement: There are many cases where a multiplicity of sensing devices can perform the same task. For instance, we could have multiple sensors, each of which can track vehicles moving in a given region - these could be motion sensors or image sensors or vibration sensors. Given a set of sensors, each of which is capable of performing a set of low level or high level tasks, how should tasks or subtasks be allocated to them so that the performance of the sensor network as a whole is optimized?

Problem Solution: We considered a number of alternative solutions. However, based on the solution to the task merging problem, we decided to proceed as follows.

1. First and foremost, we provided a formal, mathematical definition of the problem, which used a mix of operations research/optimization style terminology, as well as terminology involving sensor capabilities.

2. We introduced the notion of a service table, describing which agents provide what services. In addition, this table provides information on related parameters, such as average computation time for the answer, average latency, average size of answers returned, etc. These parameters provide valuable statistics in determining how to assign tasks to sensors.

3. Next, we showed that the problem of optimally assigning sensors to handle a given set of tasks in NP-complete. Most computer scientists believe that NP-complete problems cannot be exactly solved efficiently. As a consequence, we were forced to consider methods that rapidly provided suboptimal solutions as compared to methods that took exponential amounts of time to provide an exact solution.

4. Next, we proposed an architecture which uses a "generic" cost function — this is very general and hence, we may plug in different cost models for individual costing of network operations, sensor node computations, etc., and use the cost function to merge them.

5. In addition, we provided algorithms that: given a desired task condition *C* to be solved, will produce a way of assigning sensors to the atomic subtasks of *C*, as well as a way of ordering the atomic subtasks of *C* so as to optimize some performance criterion.

Collaborators: Fatma Ozcan, Leana Golubchik, and V.S. Subrahmanian.

1.4.6 Low-bandwidth tasking framework

Problem Statement: In addition to the logic oriented declarative tasking framework, we also proposed an alternative task specification language syntax for prompting/extracting data from a distributed, low-power wireless, sensor node network. Considering the target platform, efficiency (read low-bandwidth) was to be a major consideration in the final encoded solution.

Problem Solution: Drawing on experience from ongoing work on agent technology research platform (Interactive Maryland Platform for Agents Collaborating Together – *IMPACT*), the UMD team quickly cast the basic query syntax as a set of task, action, and constraint references, which are wrapped by a simplified token-based hierarchical structure syntax (XML, but without all the embedded descriptive tags). While the "task" entry (one per specification) equates to a specific data generating program function (such as detect or track), the accompanying set of (0 or more) "action" references map to specific program procedures which, in-turn, do something useful with the task generated data. The (optional) constraint entries typically apply as run-time data filters (pushed down as deep as possible), while evaluating the data generating task functions.

Collaborators: UMD only – This in-house effort was authored by V.S. Subrahmanian and T.J. Rogers.

Users: All *SensIT* teams (i.e. Group of 8) who used UMD's final Graphical User Interface (GUI) as part of a system demonstration ultimately used this underlying language and syntax. The list includes British Aerospace (BAE)/Nashua, BBN Technologies (BBN), Fantastic Data (FData), Penn-State University (PSU), UMD, University of Tennessee (UTK), University ofWisconsin (UWI), and Virginia Tech (VTech).

Discussion: To promote the maximal dynamic system configuration, UMD proposed to the *SensIT* quorum that network introspection calls should be included as part of the *SensIT* platform Application Programming Interface (API). The availability of such functions would minimize the quantity of hard-coded (so-called "magic") values embedded in the various software layers and thus readily support network reconfiguration (without requiring recompilation). Using such functions, the client interface would query the network regarding its capabilities and present that to the user (rather than making assumptions based on hard-coded values). UMD also proposed that the various target and sensor references should be encoded in the task specification results by suitable numeric values; the actual textual references could then be readily retrieved by standard SQL queries to a networked database engine.

1.4.7 Task generation (*TaskGen*) library

Problem Statement: At this point in the *SensIT* program, our query and task language/syntax was destined for use by Virginia Tech (the client GUI developer) as part of a communications layer for extracting data from the *SensIT* network interface (slated for development by Cornell). At the time, our existing query language/syntax documentation was descriptive only; additionally, no example code was available from which developers could learn or extend.

Problem Solution: We endowed our query system with a set of *Java* classes which provided a fairly simple but complete API; given the target region coordinates (client thread input parameters), the API spawns a pop-up dialog through which users can specify the *SensIT* network query parameters (and invoke the process execution) with minimal key-board dependency (important for palm-top GUI's). This solution included

two “base-class” definitions which encoded generic API’s for building both communications conduit and action library components; this provided a mechanism by which the system could be easily extended to support communications with new (or alternate) hardware/software platforms and define additional “actions” for processing the task generated data. Of note, our solution included several default class extensions which rendered some sample synthetic results for basic testing with upper software layers (which could be developed by third parties).

Collaborators: UMD and Virginia Tech developed the interface as a piece of “middleware” for GUI applications; modifications, updates and corrections were applied according to feedback provided by Virginia Tech’s Mark Jones, Jae Park, and their crew of graduate students.

Users: UMD and Virginia Tech (initially). Later the entire Group of 8.

Discussion: During the numerous *SensIT* gatherings (teleconferences, PI meetings and such), we continued to promote the need for introspective API’s at the network communications layer to minimize the possibility of hard-coded configuration values; these API calls would provide information regarding available task and action definitions, as well as any value to text code mappings (i.e. 105 = “M1A Tank”, or 23 = “Passive InfraRed sensor”) imposed by the underlying software layers. At this point, we also began suggesting that all software layer architects provide some sort of simulation data for basic testing with higher software layers; the availability of simulation data at every layer would greatly simplify debugging segments of the *SensIT* platform while the whole is unavailable. At this point in the development process, there was generally an overall lack of data (real or simulated) for testing the various layers (either independently or in functional groups).

1.4.8 Cougar communications conduit

Problem Statement: The Virginia Tech team needed help using the (newly defined) UMD and Cornell API’s for linking their GUI to the underlying SensIT network platform (a collection of Microsoft *WindowsCE* served sensor nodes). One of Cornell University’s

major components was to develop an external API for communicating with the underlying SensIT network hardware processes (then based on the *WindowsCE* platform – a "*PocketPC*" platform precursor). Cornell had developed *WindowsCE* embedded "Query proxy" processes and was developing connectivity for this with its home-grown "Cougar" distributed database system. The researchers at Virginia Tech were busy with their GUI development, and generally required help with the Cougar required *JDBC* API. In short, the Virginia Tech team needed some pointers for dealing with the Cougar communications layer.

Problem Solution: We installed a local copy of Cornell's Cougar database server (for initial testing without the *SensIT* node network) and encoded the necessary Cougar API calls as a plug-in communications conduit in our task generation library. Of note, since network introspective calls were not generally supported by the Cougar API, we chose (for this conduit) to read them from locally maintained database files via standard API calls (i.e. local codebook values defined in *MS–Access* tables and accessed via *JDBC/ODBC* data queries).

Collaborators: Cornell University (Phillippe Bonnet), Virginia Tech, UMD.

Users: UMD and Virginia Tech.

Discussion: The Cougar interface provided support for a few simple queries and returned simple canned data. Of note, the *WinCE* query proxy code (necessary to test actual connectivity with the underlying *SensIT* node network) was generally unavailable until after our initial field test integration (SITEX01).

While both Cornell and UMD attended the field integration exercise (SITEX), their time-slots did not overlap; the Cornell team had already departed the exercise prior to UMD's arrival. Hence, the Cornell query proxies were unfortunately not part of the default node software load and thus were unavailable for UMD's testing. As a consequence, the Virginia Tech, UMD, Cornell University *SensIT* node connectivity was not testable during SITEX01.

1.4.9 Simulation Data Base (SimDB)

Problem Statement: Following SITEX01, UMD found itself still without any reasonable data to test its interfaces and query control integration with Virginia Tech's GUI. While Cornell's query proxy code had recently become available, the working node network was not (UMD had two nodes which was far below the minimum number necessary to form a reasonable sensor network).

Problem Solution: UMD adapted SITEX01 ground-truth data into the "SimDB" conduit interface; this provided a limited capability to "play-back" recorded target positions as if it were rendered from a *SensIT* node network query. During field experiment runs, BBN routinely collected ground-truth data via a GPS equipped laptop. UMD obtained these ground-truth files and populated a database from several (about six) of the more interesting test run files. UMD then built a conduit interface (SimDB), which instantiates target *SensIT* network queries as persistent SQL queries over the ground-truth data. In short, "live" tracks are simulated by stepping through the recorded ground-truth data.

Collaborators: UMD and BBN.

Users: UMD and Virginia Tech. This work allowed us to fine-tune the GUI/TaskGen integration and experiment with numerous display capabilities and modes.

Discussion: This work also provided perhaps the first integrated proof-of-concept demo capabilities for the Virginia Tech GUI.

During the next *SensIT* PI meeting, the quorum generally agreed that the current *WinCE* platform performance was abysmal. Sensoria, the hardware provider, agreed to research and field an alternate platform (later identified as a Hitachi *SH4 Linux* variant). While most of the PI groups agreed to move readily to the new platform, Cornell declined to commit to a query proxy *SH4* port for the next field integration exercise. Discussed options included building an interim "gateway" server on one of the *SensIT* nodes and using Fantastic Data's web database (the distributed data cache) as a means of moving queries and data around the network. Of note, the Virginia Tech GUI was demonstrated to several key individuals (Dr. Sri Kumar and assorted BBN researchers) during the PI

meeting. While the network communications channel was now in question, the demo (over ground-truth data) proved the existing Virginia Tech GUI as a viable end–user/soldier interface.

1.4.10 Preliminary *SensIT* network Gateway (*SH4*) interface

Problem Statement: Following the PI meeting, UMD accepted the task of exploring the feasibility of implementing an interim *SensIT* network gateway for use in the next field integration exercise.

Problem Solution: UMD built a *Linux* PC workstation, loaded the available *SH4* cross-compilation tools, and then began to develop the gateway server. As this "component" actually spans two separate network realms (remote *Java* client vs. embedded *SensIT* node process), the solution must be viewed as two separate development issues.

To solve the lower level issue, we developed a *Linux SH4* based TCP/IP command server, which ultimately manipulates low-level function pointers to provide answers for client rendered one-time and persistent queries. This interface supports *SensIT* network introspection so that client interfaces can discover the underlying network capabilities (and present that representation in the GUI). This interface also implements a generic API for the various task functions and action procedures necessary to support the desired user queries (this makes the implementation readily extendible).

To solve the client issue, we built a conduit interface (Gateway), which instantiates target *SensIT* network queries as gateway TCP/IP commands and converts the resulting data stream back into *TaskGen* result objects.

Collaborators: UMD only.

Users: UMD initially, but eventually included Group of 8.

Discussion: While this proof-of-concept piece solved the initial client to/from *SensIT* network communications void, it remained unconnected with the actual low-level data query processes. These lower level connections required more experimentation with the evolving Sensoria *WinsNG* and Fantastic Data APIs (see next section).

1.4.11 Preliminary *SensIT SH4* gateway task and action module set development

Problem Statement: The current gateway interface lacked connectivity with the actual underlying *SensIT* network data query processes. Moreover, the *SensIT* quorum had yet to agree to a base set of supported user queries or shared (cache) data format schemas.

Problem Solution: UMD implemented a base set of gateway task code extensions, which mapped user queries to the underlying, distributed, Fantastic Data cache processes. This base set of tasks included the user queries "Ping" (non-persistent/retrieves current node status reports), "Detect" (persistent/retrieves current and updated target detection reports), and "Track" (persistent/retrieves current and updated target tracking reports).

Collaborators: UMD and Fantastic Data.

Users: UMD and Virginia Tech (initially), and Group of 8 later.

Discussion: The Gateway conduit ↔ *SH4* gateway ↔ data cache layers effectively replace the functionality previously provided by Cornell's Cougar database ↔ Query Proxy processes.

In addition, at this point the inter-module development process remained hindered by a general lack of play-back test data. Moreover, most PI groups lacked sufficient node inventory to establish a working SensIT network. As such, basic inter-process debugging remained difficult at best.

The BBN Pre-SITEX02 integration meeting also highlighted, perhaps, a support oversight encountered by several PI groups. Future remote collaborative efforts should either clearly announce a lack of developer support or provide such an interim service. For in-house development and testing, UMD and several other PI groups had installed the necessary *Linux* development environment on stand-alone PC's. Here, we happily implemented, tested, and cross-compiled the necessary source code for *SH4* node execution. As expected, once all was well, we committed the various software packages, via Internet connection, to BBN's CVS repository. Upon preparing for the

Pre–Integration meeting, many of us assumed we'd just take our *Windows* laptops to the meeting, plug into BBN's development network, setup a work environment, and continue coding where we left off. Alas, this was not to be. BBN's Wind-Tunnel *Linux* setup was not equipped sufficiently to support such a bevy of needy programmers. Moreover, the primary available network connections were NOT connected to the Internet and as such, lacked access to the BBN repository server. As for UMD, we overcame this problem by building a temporary work environment on Fantastic Data's *Linux* laptop from backup files, which we had fortunately brought on a zip disk.

1.4.12 Imager control, manual trigger (*SH4*) module

Problem Statement: During the Pre-SITEX02 integration meeting, Sensoria revealed the first imager node (a node with a software controlled digital camera). BBN was anxious to get someone to adapt an interface to it and encouraged UMD to give it a try.

Problem Solution: Just prior to SITEX02, we experimented with an imager node and developed both a client GUI control panel (allowing remote modification of the imager configuration) and a gateway task module to manually trigger the imager and return the information regarding the rendered image(s).

Collaborators: UMD and Sensoria.

Users: UMD and Virginia Tech.

Discussion: At first glance, the Imager API appears rather simple; triggering the imager, in-fact, is simple. Obtaining a list of the rendered images, however, turned out to be non-trivial. On closer inspection, we discovered that the imager node actually implements two computers. One unit controls the camera and the other is **similar** to the standard *WinsNG* node (commands and data get shared between the two units via internal Ethernet protocols). Unfortunately, the embedded computers in our test imager node(s) do not time synchronize at boot-up. This confounds the image name lookup following a trigger event as, a) image transfer is not immediate, b) the trigger command does not prompt the client process when the image transfer is complete, and c) time stamped image file names

cannot be guessed, since the camera clock, from which the time stamp applies, is not synchronized with the base *WinsNG* client process clock.

Of note, the current file name discovery code, written by one of BBN's field reps (Jimmy) during SITEX02, works by reading the local timestamp at trigger time, waiting a given period (for expected file transfer), and then trying to find the image files whose time stamp closely matches that of the recorded trigger time. While this method works a majority of the time, it remains somewhat intermittent; this is to say that the imager trigger task fails to report the generated image name about 20% of the time.

On the very last day of SITEX-02, we managed to see live end-to-end connectivity using the Virginia Tech GUI. Detection and tracking result updates were painting to the screen and we finally managed to process a system predicted imager trigger which, in-turn, captured the dimly illuminated target vehicle passing before the imager node. Unfortunately, a majority of the SITEX02 participants had already departed. These final events were only witnessed by those monitoring the network from the base-camp.

1.4.13 In-Field alternate track generation support extension

Problem Statement: Part-way thru SITEX02, BBN requested a gateway compatible mechanism, which would allow them to both export *SensIT* network target detection records and import externally rendered target track update records.

Problem Solution: UMD implemented a two-fold solution. First, we implemented/adapted a gateway export action module, which wrote detection query results to an externally accessible (outside the *SensIT* node network) data file. Next, we implemented/adapted a gateway import task module, which would read BBN's *MatLab* generated track result records (from the externally accessible data file) and insert the track updates into FData's distributed data cache. With these two components running (simultaneously but from separate gateway processes), we once again had complete data flow channel from the Signal processing layer, all the way up to the client GUIs.

Collaborators: UMD and BBN.

Users: Virginia Tech, UMD, Fantastic Data, BBN.

Discussion: This component allowed us to continue testing the numerous software layers in PSU's absence.

1.4.14 In-Field external imager trigger prediction support modifications

Problem Statement: Happy with the recent external tracker adaptation success, BBN desired similar gateway modifications for responding to an externally driven image trigger prediction process.

Problem Solution: UMD implemented/adapted a gateway import task module. Initially similar to the Track Import task rendered (see section 4.13), this module controlled an *SH4* process, which would monitor the time and trigger the imager according to the predicted target proximity time stamp.

Collaborators: UMD and BBN.

Users: UMD and BBN.

1.4.15 Cache-borne task specification table

Problem Statement: Following the Santa Fe PI meeting, the *SensIT* quorum decided to drop the Virginia Tech GUI from the base software configuration (also called *SenSoft*). This decision quickly caused some internal *SensIT* group fragmentation and lead Fantastic data to experiment with an enhanced signal processing control mechanism; Cornell, in-turn, voiced interest in cache access to our query specification entries. UMD

proposed injecting its gateway query specification data into the distributed data cache for easy 3rd party access.

Problem Solution: UMD modified the gateway server module to create (if necessary) and maintain a set of cache-borne query specification tables, derived from the gateway client requests.

Collaborators: UMD only.

Users: UMD, Fantastic Data initially, Group of 8 eventually.

1.4.16 Initial *OpenMap* based *SensIT* GUI

Problem Statement: BBN was interested in the feasibility of using an *OpenMap* based GIS interface to interact with UMD's gateway interface libraries in an effort, perhaps, to reuse much of what had already been developed and tested. The UMD team, suddenly faced without a map-based GUI (as the *SensIT* quorum had previously decided not to use the Virginia Tech GUI) for displaying, making sense of, and debugging the often numerically intensive query results, was also curious regarding *OpenMap* adaptability as a Virginia Tech GUI replacement.

Problem Solution: UMD developed an initial *OpenMap* data layer interface, which served as a proof-of-concept piece for interacting both with the BBN's *OpenMap* and UMD's task generation (*TaskGen*) interface. Shortly thereafter, we implemented our initial stand-alone, multi-layer, *OpenMap* GUI through which the user could define, execute, and observe the results of *SensIT* network target detection and track results.

Collaborators: UMD, BBN, Virginia Tech.

Users: UMD initially, Group of 8 eventually.

Discussion: Shortly after fielding the initial *OpenMap* interface, we revisited Virginia Tech's GUI and noted many of its useful display features. Many of these were later mimicked in our *OpenMap* GUI.

1.4.17 *OpenMap* GUI applet

Problem Statement: Several *SensIT* groups voiced interest in experimenting with our *OpenMap* based GUI shortly after our initial release. Unfortunately, at that time, it was very much still a work-in-progress and we did not favor the notion of supporting client configuration and update issues on such an "Alpha" package; delivering this utility as a web embedded applet, however, would serve to reduce the maintenance headaches, demonstrate client portability, and global access possibilities.

Problem Solution: The UMD team quickly repackaged the *OpenMap* GUI as a *Java* applet, learned the separate Netscape/Explore HTML applet embed syntax, and experimented with the system property settings, which are necessary for an applet to function as a stand-alone application.

Collaborators: UMD only.

Users: The Group of 8 team.

Discussion: Prior to Sun's *Java 2*, version 1.4, release, such an applet was generally impractical due to standard *Java* "Sand-box" issues. *Java 2*, v1.4, however, provides such system property override capability. Of note, additional ease-of-use could be provided were this code served via *Java*'s "*WebStart*" library. Here, users would not need to pre-set the system properties. Rather, they could simply invoke the package via URL reference (without a browser), and the *WebStart* interfaces would prompt the user for permission to run and then down-load and execute the necessary resources (without a browser).

In late August, 2002, UMD, Fanastic Data, and BBN collaborated to demonstrate one of the first widely distributed uses of BBN's Waltham *SensIT* node test bed. For this demonstration, BBN manned the node network, and provided target personnel and vehicles. Meanwhile, Fantastic Data controlled the data cache and signal processing layers from their San Francisco office. Finally, sitting in Dave Shepherd's Virginia office, UMD defined and controlled *SensIT* network queries via the live Web-launched *OpenMap* GUI. Fantastic data's collaborative track generator worked well and the GUI accurately painted numerous scheduled BBN target groups, and detected and tracked several unscheduled "targets of opportunity" as they moved through the sensor node field.

Roughly two weeks later, BAE/Nashua demonstrated their collaborative track generation utilities (for the same DARPA audience, i.e., Dr. Sri Kumar and Dave Shepherd) using the same widely distributed Waltham test-bed, data-cache controls, and the UMD *OpenMap* GUI scenario. BAE/Nashua reported that the tests were well

received. Of note, shortly after this, BBN soon began to regard the UMD *OpenMap* GUI as the defacto standard *SensIT* network GUI.

1.4.18 Modified classification code and shape display

Problem Statement: Shortly after our successful demo for Dr. Kumar, Fantastic Data had successfully integrated some of Univ. of Tennessee's target classification code. The current GUI target shape display was now antiquated. Rather than displaying the target as a moving box, the GUI users desired to vary the target shape according to the classification type.

Problem Solution: UMD looked closely at and attempted to mimic the target classification shapes as were previously rendered by the Virginia Tech GUI. Our solution implemented several display shapes to indicate target type categories as follows: Small and Large Wheeled Vehicle, Small and Large Tracked Vehicle, Motorcycle, and human. UMD also appended the newly defined target type classification codes in its codebook database.

Collaborators: UMD, Fantastic Data, Univ. of Tennessee and Virginia Tech.

Users: Group of 8.

Discussion: Following several remote test-bed (Waltham) demo success stories, *SensIT* groups from the University of Wisconsin and Penn State University began voicing interest in the *OpenMap* GUI usage. We advised them concerning the setup requirements and soon were fielding questions and comments from these two new user groups as well.

1.4.19 Track history GUI line and color display

Problem Statement: Requested GUI embellishments included an optional target track history display and an alternating track display color mode for easy visual separation of targets on a cluttered interest area display.

Problem Solution: Answering the history request, UMD provided an optional trailing line display, which indicates a target's previous positions (ordered over time). UMD also

implemented a rather simple target color display mechanism. Rather than plot all targets in the same color, the new method iterates through a set of roughly eight colors.

Collaborators: UMD, BAE/Nashua, Univ. of Wisconsin, BBN.

Users: Group of 8.

1.4.20 Cache-borne codebook table development

Problem Statement: Just prior to the final *SensIT* "CapStone" demo, BBN requested a *TaskGen* library modification that would allow reading codebook values directly from the *SensIT* network data cache. The goal was to eliminate the need for external codebook database references that were common with the Web launched GUI.

Problem Solution: To support this, UMD first modified the gateway to define and populate codebook values (if necessary) during gateway initialization. Next, we modified the *TaskGen* initialization routines to support this alternate codebook value resource path.

Collaborators: UMD and BBN.

Users: Group of 8.

Discussion: While these modifications were in-fact on-line for the CapStone demo, it had not been entirely tested. To avoid demo-day disasters, we chose to operate the *OpenMap* GUI with codebook values served from a local database (one of the original default configurations).

During the November 2002 *SensIT* CapStone demo, the UMD *OpenMap* GUI was used successfully by BBN, Fantastic Data, Univ. of Tennessee, BAE/Nashua, PSU, and Wisconsin. All demo displays worked reasonably well (content varied slightly according to underlying selected generation methods).

1.4.21 Finalized in-cache codebook table modification testing

Problem Statement: Following the CapStone demonstration, both BBN and researchers at Rome labs requested modifications to finalize the final cache-served codebook centric GUI.

Problem Solution: A closer look at the *TaskGen* initialization methods resolved this issue. Additional configuration file options made this feature an easily selectable operation mode.

Collaborators: UMD, BBN, Rome labs.

Users: UMD, BBN, Rome labs.

Discussion: Shortly after the final codebook modifications, BBN requested that UMD edit the Task Generation / Gateway Server section of the pending *SenSoft* application manual. UMD returned a heavily edited version, which corrected some of the usage misconceptions and provided greater detail regarding the component interoperability.

1.4.22 *OpenMap* GUI imager query display modifications

Problem Statement: Automated imager usage ultimately proved problematic due to constantly recurring data propagation latency issues. The trigger tasking remains functional, but the existing implementation did not include the utilities necessary to retrieve and display the rendered images.

Problem Solution: As part of a final system "tweak", UMD adapted some of the image retrieval code from the original Virginia Tech GUI, and provided a display and control mechanism for the *OpenMap* GUI.

Collaborators: UMD and Virginia Tech.

Users: UMD.

1.5 Conclusions

The last few years has seen a great increase in sensing, communications, and computational capabilities. The goal of the *SensIT* program has been to harness these major advances into a single architecture, which brings together diverse technological advances in these three disparate fields so as to effectively support the war fighter. The University of Maryland team focused on the task of developing techniques to task the sensor network.

Our first major contribution was to develop a formal model of a task (which is appropriate for sensors), together with a formal tasking language. This tasking language allows users to specify (i) the geospatial area where a task is to be performed, (ii) a temporal component specifying when and how frequently the task should be performed, (iii) a component that describes the conditions to be looked for, based on the sensor readings, but possibly correlated with other data sources as well, and (iv) what actions to take when those conditions are met.

Our second major contribution was a mechanism to process tasks effectively, so that the resources of the system can be intelligently utilized, thus supporting the scalability of the system as a whole. For this purpose, we developed methods to merge multiple tasks together, taking advantage of any overlaps amongst those tasks. We built a suite of algorithms for this problem called task partitioning algorithms, and we conducted detailed experiments on the effectiveness of these algorithms. Our experimental results show that Greedy-Basic is the best task partitioning algorithm to use.

Our third major contribution was an algorithm to take a given task and determine which of a given set of sensors should process that task. We showed that the problem of optimally allocating tasks to sensors is NP-complete. To address this problem, we were able to come up with heuristic algorithms.

Our fourth major contribution was the design and implementation of a comprehensive Sensor Task Manager (STM) program and GUI that allows individuals to

specify the tasks they are interested in having performed. The STM program handles execution of the tasks.

Our fifth major contribution was the development of numerous system components, often in conjunction with other *SensIT* contractors, so that the entire *SensIT* architecture (which involved components from multiple contractors) could function smoothly. Specific contributions include the *OpenMap SenseIT* graphical user interface, the gateway task and action module, imager computations, and cache based computations.

1.6 Recommendations

The current *SensIT* effort demonstrates clearly and unambiguously that we now have the ability to seamlessly integrate sensing technology, communications technology, and recent advances in computing technology for use in large scale integrated battlefield operations.

However, for future battlefield operations to be successful, there is a key need for scalable intelligent reasoning. Specifically, *SensIT* currently lacks:

1. **Predictive reasoning.** It is possible to do better than taking actions after vehicles have been detected at various locations. Based on past detections of vehicles, a system should be able to predict where these vehicles will be at various points in the future.

2. **Spatial reasoning.** Placement of sensors and the dynamic movement of (mobile) sensors can be done either by ad-hoc mechanisms, as it is today, or by more principled methods that dynamically evaluate a set of needs and decide where to place the sensors so that those needs can be best met.

3. **Spatio-temporal reasoning.** This extends the previous item. When it is known where and when certain needs are likely to occur, more informed decisions about where to place sensors could be made. The ability to do this statically and the ability to dynamically move sensors around with a changing battlefield situation is critical.

4. **Diagnostic reasoning.** When certain events are noted on the battlefield (e.g. the sensors say one thing, but the effects seen on the ground do not match what the sensors are saying), then we need to be able to ask "why" questions. Why are we off? Why are the sensors telling us that there are no significant vehicle detections in region r, but forces are still being ambushed? The need for this kind of diagnostic reasoning is critical.

5. **Logical reasoning.** In addition to the above, we need standard logical reasoning - this tells us, for example, that if a large convoy of passenger vehicles are spotted along a highway, traveling unimpeded, then there is a high chance that influential members of a foreign government are fleeing.

These are just a few types of reasoning that need to be performed in order to better interpret and act on data obtained by tasking sensors. We believe this is critical for sensor technology to be maximally successful in war fighting efforts.

Section 2: Representation and Automated Analysis of Narrated Events

2.1 Introduction

The understanding of recounted events (stories, histories, etc.) so that one can determine the best course of action is a common and critical part of decision making in many fields, including anti-terror intelligence assessment, military operational command, software engineering [1,2], and diagnostic problem-solving [10]. In these and many other areas, executive-level decisions must be made in real time by a person in the context of a "story" describing past/ongoing events. Decision making is hampered by the limited knowledge that is quickly available to the decision maker in a timely manner. One hypothesis of the proposed work is that an appropriate “story” can provide the decision maker with a “big picture” containing the needed information. Here and in the following, we use the term *story* to mean not just the description of a sequence of events that is unfolding over time and space (the “plot”), but also the associated relationships among the actors/objects in the story, specifications about which aspects of an object are fixed and which are mutable, functions describing the nature of the mutation in various contexts, factors such as motive, beliefs, etc. It is this complicated network of relationships that makes the problem hard.

A possible way to enhance human understanding and decision making is to provide automated interpretation of unfolding events through a computer system that can represent, interpret ("understand"), and communicate alternative response strategies based on a knowledge base of past related experiences. Such a system could be extremely valuable in suggesting additional information to obtain, predicting potential future events, and outlining alternative courses of action for consideration. Existing technology in artificial intelligence (AI), software engineering, and human-computer interactions is inadequate to implement a “story understanding” system that could function effectively as outlined above.

However, we believe that current technology has reached the point where research on some key issues could make automated "story interpretation" feasible. The overall goal of this project was to identify which aspects of story representation and analysis can be implemented using existing technology, and which represent critical research issues. The study also assessed the comparative advantages (or disadvantages) of using a story format for analysis and outcome exploration (this format has already been shown to be exceptionally useful for training). Key research questions focused on the potential of various technologies to: represent evolving sequences of events and associated conditions (contextual variables) in a machine-interpretable format along with course-of-action (COA) options; rapidly retrieve past related situations and relevant general knowledge; apply inference methods to encompass the interpretation of stories, to identify missing but needed information, to generate potential actions, and to justify its recommendations upon demand.

The specific aims that guided the work that we undertook were:

1. Assess the availability of source stories in multiple domains. Determine the representation and analysis aspects that are common to these diverse stories so that the software and methods produced will ultimately be reasonably general. Specific application domains that were considered include home-security intelligence, tactical military command, software engineering, and diagnostic problem-solving.

2. Evaluate technologies for formally representing the events and associated conditions/relationships forming a story. This involved considering past work on scripts, functional representation languages, AI methods for temporal and spatial information representation, and related work, assessing their adequacy for the selected application(s) and identifying needed enhancements. At a minimum the representation needs to capture the actors/agents involved, actions occurring, the temporal relations between events, and relevant decision points in stories, plus it must enable the representation of generic plots/ scripts that can be related to specific cases.

3. Assess the feasibility of implementing a story representation system based on the methods developed in (2). We prototyped a system to encode a few stories in the

selected application domains, and specified an interface allowing the rapid entry of a new story. The interface involved direct encoding of the background knowledge and terminology in a class of applications in a formal representation that underwent parsing and error detection. The indexing of new story information to the knowledge base and/or case base of related past stories was also studied.

4. Represent relevant cause-effect knowledge within the sequential event formalism developed as above, as well as generic course-of-action options and, to the extent possible, relevant spatial information, building a prototype knowledge base.

5. Explore basic inference abilities that are needed to generate useful output from the evolving prototype system. This included abductive reasoning methods based on cause-effect inferences [11] that generate interpretations of events. We also assess the ability of deductive inference methods to generate and possibly rank options for actions to take, and examined the integration of knowledge-based reasoning with retrieval of informative past cases.

6. Evaluate the functioning of the prototype story analysis system outlined above on one or more demonstration examples of limited scope to illustrate the viability of the methods developed. This is intended to be a demonstration system, not a robust application release.

In summary, our purpose was to identify key long-term research issues in knowledge representation, software engineering, common sense reasoning, generation of "what if" scenarios, adaptability and learning, qualitative physics, etc. that need to be addressed. We wanted to assess the ability of existing software technology to make sense out of large volumes of data, and to determine what open research issues need to be addressed to develop useful systems for automated analysis of narrated events and decision support in the context of complex ongoing situations. Ultimately, one wants to develop tools for analyzing the meaning of a situation—where might the current situation lead and what should we do to accomplish our goals? It should do this in the same way that "episodic memory" is thought to function, presenting the most reasonable alternatives to the user so that only a very few need to be examined. Research (Gary

Kline) has shown that few people use the type of cost-benefit analysis that is taught in management school when making decisions. Instead, they rely on their past experience, and stories about others' past experiences, to bring a course of action (COA) to mind immediately. They then assess the viability of this solution in the current context.

2.2 Methods

This research takes a story to be a sequence of episodes that are related to a goal, intention, moral or lesson. The story organizes multiple events and information into "storylines" that integrate the past, present and future with reference to goals or intent. The story provides a way of capturing experience and using this experience to predict and plan for the future.

Therefore, we believe an approach using stories can help identify the best COA based on a repository of all past experiences (expressed as episodes and episodes chained together to form stories, along with "compiled" knowledge relevant to a problem). This approach should be able to target information collection based on previous and expected episodes/threads. For prior episodes, the information collected can help confirm or deny what is believed to have happened. For future episodes, information collection can focus on that information needed to improve or estimate about whether the episode will actually occur and what its consequences are likely to be. This will decrease the time needed to respond to crisis events, such as a potential terrorist threat. An episode repository provides a corporate knowledge base of past experience to educate new decision makers. We thus did online and library searches for relevant sources of readily available past stories/cases in areas such as software engineering and urban warfare, plus reviewed available software packages.

One unusual aspect of this research is that, among other things, we tried to develop models of activity "from scratch" – without the existence of an a priori model. While it may be impossible to develop detailed physical (or "quasi-physical" based on qualitative physics) models that explain cause and effect factors within an event (or episode), we hoped to be able to develop models that predict the expected successor to a

given episode based only on a “story library” containing historical storylines (threads of episodes) and an ontology describing the terminology used in the domain of interest.

2.3 Results and Discussion

Initially, during the first year of this project, we identified and evaluated the suitability of a number of existing software tools: 1) For the development and analysis of semantic networks -- most notably the Semantic Network Processing System (SNePS) from NYU, Syracuse -- for representing the detailed structure of activities within an episode; 2) For the development of ontologies describing a specific domain (e.g., Urban Warfare) – most notably Cyc, and; 3) For identifying similar (and dissimilar) episodes so we could build up a story based on the predecessor and successor episodes (and outcomes) of historical events (a repository of episodes) – most notably Case Based Reasoning tools from the University of Edinburgh.

The basic premises arising from this evaluation are that we can:

- Characterize the current episode in a way that allows it to be compared with analogous episodes (stored in a library).
- Identify analogous episodes that are most similar to the current episode with respect to factors important to the specific analysis being conducted (questions being asked).
- Determine similarities and differences among the episodes.
- Use these findings to both improve our picture of the current situation and to improve decision making with respect to future courses of action.

With respect to the first objective (evaluating existing software tools and experimentally producing representations to determine their capabilities and ease of use), we discovered that the use of any one required the development and maintenance of an elaborate model (or linked models) – e.g., of terrain, typography, equipment capabilities, logistics. Also, SNePS and Cyc had considerable overlap – while SNePS is built around

Semantic Nets and Cyc is built around rules, they can represent the same features – indeed, Cyc uses graph analysis algorithms to do much of its reasoning.

We soon realized that (contrary to research claims) there were no practical tools to transform the text describing an episode into a useful semantic net. We shifted our approach to prepare for (using manual analyses) using Cyc to build an ontology of terms unique to a domain and for recognizing declarative sentences that might serve as rules to relate these domain specific terms. We conducted these experiments initially using simple fictional stories and subsequently using Urban Warfare episodes. As part of this effort, we defined those components of an episode that we needed to capture. These are:

An EPISODE provides the basic element of the story's narrative or plot. It describes some (usually 1) action. It should:

- Define the spatial and temporal boundaries of the action. As episodes are composed into stories, these boundaries provide one check on whether the links "make sense" – teleportation and time jumps may occur in fiction, but generally not in the type of stories we're analyzing (e.g., software development projects or urban resistance movements) ;

- • Have start (pre-) and stop (post-) conditions. The start conditions specify the conditions necessary to start the action; the stop condition defines the end of the action. The stop condition (and the start condition for the next episode(s)) will usually be a decision (e.g., attack the enemy), an action (e.g., a car runs into a tree), or the lack of an expected decision or action (e.g., a meeting ends with no decision or consensus). It may also have a goal, in which case its (immediate) success is determined by the distance between the goal and actual end condition. Note that the success of a story as a whole thread of episodes can only be judged at the end of the thread.

- Have predecessor and successor episodes (except for the first and last episodes which have only pre- and post-conditions). The end condition of the

preceding episode must match (to some degree) the start condition of the current episode – providing one check on the links making sense.

- • Contain (conceptual or physical) objects that act/interact to perform the action being accomplished by the episode. This structure might be described, for example, by some type of semantic network structure.

- • Specify structures of objects that are invariant within the episode (e.g., lines of authority among objects) as well as other invariant properties that are not associated with or determined by specific objects (e.g., in most circumstances, the weather)

An OBJECT is an element that is contained in and linked to other objects in an episode. Since it is working, possibly with other objects, to accomplish an action, it has behavior associated with it, although it may only fill time or space in a specific episode. It is characterized by:

- • "Invariants" that remain fixed across all the episodes it participates in (e.g., name, sex, size (for inanimate objects);

- • "Characteristics" that are invariant within an episode but can change, according to specified rules, at episode boundaries. Elements that may change within an episode but that are observed only at episode boundaries are characteristics.

- • "Variables" which can change, and whose change is observed, within an episode.

- • "Behavior", both autonomic and through interaction with other objects.

A BELIEF is a strongly held position or idea attached to an object. For non-intelligent objects, these are the same as the rules by which they operate (their behavior). For intelligent objects, these are beliefs that are often elicited as rationale for a decision. Thus, in Fred Moody's *I Sing the Body Electric*, the story of a Microsoft software

development project, the value of being first rather than best is continually emphasized by management.

HISTORY is the rationale underlying a belief. “In our experience, this has (or has not) worked in the past.”

There are several critical research issues involved in this approach. The overarching question is whether it is possible to build an “implicit” model of activities in a domain based (almost entirely) of plausible (and often historical) sequences of episodes in that domain. The fact that episode type A precedes episode type B models a potential relationship between A and B. Understanding the circumstances under which this relationship does or does not hold can provide an understanding of factors critical to the relationship. Changes in the existence (or strength, as measured by the probability of occurrence) of the relationship may indicate changes in the context (or more global situation) that need to be factored into decision making. The ability of a system to learn with experience will also be examined.

We also came to appreciate more the central role that explanation generation plays in story understanding. Because of this, we determined that the overall objectives of this project could best be met by approaching the development of a story interpretation system by working at two levels in parallel: one level continues to focus on evaluating existing software as originally planned, while a second level extends our planned investigation to assess the use of explanation-based (abductive) inference.

Our work on abductive inference did not adopt specific software packages nor large scale pre-existing knowledge/ontology sources such as Cyc. Instead, we investigated whether knowledge represented in a natural descriptive fashion could be used to automatically generate plausible explanations for unexpected events. Our hypothesis was that contemporary technology for cause-effect reasoning has matured to the point that it can provide an effective and efficient inference method in automated reasoning systems based on descriptive knowledge. More specifically, we believe that advances over the last several years in the use of abductive inference methods and

probabilistic reasoning with Bayesian networks make the automated processing of descriptive knowledge both a viable and timely approach to explore.

Our initial work related to explanation generation and abductive reasoning included the following. First, we completed most of the implementation for a knowledge acquisition tool based on encoding of descriptive knowledge. The output of the knowledge acquisition system is an application-specific automated reasoning system for end users that includes a knowledge base, an inference mechanism that can process causal and other types of information in the knowledge base in useful ways, and an automatically-generated user interface. As an initial problem on which to evaluate this system, we used the problem of interpreting and responding to the unexpected discovery of chemical contaminants in the Chesapeake Bay. A prototype system for abductive reasoning using descriptive knowledge about chemical contaminants was implemented and was found to be capable of interpreting the meaning and some basic implications of relevant multi-sentence text.

During the second year of this project, we extended the above work by understanding four tasks. These were: 1. find urban warfare stories/episodes; 2. use these episodes to build an ontology (using Cyc) suitable for identifying similar (or different) episodes; 3. experiment with ways of measuring the "distance" between terms; 4. implement a demonstration program combining natural language processing, rule-based deduction, and indexed case retrieval in the urban warfare domain; and 5. in collaboration with the Fraunhoffer Institute for Experimental Software Engineering Research, find descriptions of "episodes" in software development projects, as documented in Software Inspections, that were suitable to serve as stories with outcome predictions.

On the first task, we collected information resources for different urban warfare battles. We have extracted data for multiple episodes spanning several distinct battles and have developed and refined a knowledge base for describing urban warfare episodes.

Work on the second task produced several insights. First, the terms themselves (e.g., the fact that one side used Kalashnikov assault rifles that was reported in the source documentation) are less important than the ontological implications that can be drawn

from them (e.g., that they were using weapons commonly found in weapons bazaars, that they were using weapons that do not require a sophisticated infrastructure for maintenance). Note that these inferences are based on "common sense" knowledge about the terms in the general domain of urban warfare, not on data from any specific episode/story. Second, while Cyc provides the ability to create separate "micro theories" to resolve potentially conflicting uses of a term or concept, these created problems in assessing term similarity, since two terms could be linked through several micro theories, creating drastically different similarity measures. Based on the above, we experimented with using OWL-DL instead of CYC to hold the ontology in addition to expanding the number of terms employed as described in the first task. We also conducted some initial experiments in the context of our third task on distance measures, and are currently analyzing the data.

On the third task, we studied case based reasoning as a means to evaluate the similarity of episodes based on term (or inferred term) similarities. A naïve approach would be to apply a distance measure from a term in Episode A to all terms in Episode B and compute a distance based, for example, on the average of all these distances or of all non-zero distances, since we are likely to be dealing with an extremely sparse similarity matrix. An investigation of available technologies uncovered tools such as AIAI Case-Based Reasoning Shell developed at the University of Edinburgh that identifies the sets of characteristics, each expressed as a vector, that, used together, best differentiate among cases (events or episodes) in this work.

On the fourth task, we implemented a prototype decision system that integrates knowledge based reasoning and case retrieval in support of story interpretation. Currently, the knowledge base has thirty-five basic attributes that may be extracted from a story. The knowledge base contains deductive rules which, when applied, give us values for an additional six attributes. Abductive reasoning rules enable us to make up to 10 different assessments about a story. The knowledge base provides suggested courses of action for addressing each one of these assessments. Examples of inferred attributes of story episodes include: enemy technological level and motivation, and quality of intelligence information.

Appendix A, *Integrating Knowledge-Based and Case-Based Reasoning*, provides a more detailed description of the urban warfare cases, the prototype knowledge base, the reasoning methods, and case retrieval in the urban warfare story interpretation system.

As a fifth task, we investigated the applicability of the "stories" concept to decision-making in software engineering. The goal of providing decision-support in this area is especially appealing due to the large array of practices, methods, procedures, etc., that exist for software development, all of which are claimed to be beneficial, and which can significantly impact the success (in terms of budget and schedule) of a development project. Large-scale studies, such as those conducted by the Standish Group, consistently show that achieving such project successes is a problematic undertaking. To undertake this study we chose as a test bed the practice of "software inspections," one of the most mature and well-studied practices available. Our investigation identified a lack of suitable data for describing the episodes needed to build such decision-support models. Ideally, we would have the results from a series of inspections of a single product, with the results of each inspection describing the outcome of a preceding episode (e.g., requirements specification, architecture development). Data in the area of inspections, however, tended to fall into two large categories:

1. Organizational-level data, which describe aggregate results across many applications of inspection and many projects, by way of claiming high-level benefits. For example, a paper might discuss how inspections were applied on all projects in the organization and resulted in measurably lower rates of defects in the systems produced. The problem with this type of data is that there is no way to make a linkage between an individual inspection application and a specific outcome; it provides only a general statement of overall effect.

2. Snapshot data, which describes the results from a specific inspection in a specific context. For example, a paper might describe a specific software module that was inspected, the amount of time required, and the number of defects removed as a result. The problem with this type of data is that there is no way to make a linkage from those inspection results downstream to future experiences with that module. For example, there

is no way of telling if in the future the inspected module was less error-prone, required less testing, etc.

Types of data that might usefully be captured longitudinally include, for example, percent of previous errors corrected, number of errors representing misunderstanding of specifications, project size (measured as number of discrete functional requirements, design modules, lines of code, etc.), and effort spent on defect correction versus system construction. Such data may be used to improve the software development process through comparison with historical episodes in an organization's library.

Appendix B, *Software Inspections as Choice Points*, provides a more detailed summary of our assessment of existing software inspection data, and of how story interpretation could contribute to decision support for software development projects.

2.4 Conclusions

1. The existing general purpose software tools that we evaluated for story interpretation appear to offer only very limited utility.

2. A significant barrier to development of story interpretation in the application areas that we considered is the limited availability of online story repositories.

3. Evaluation of a prototype story interpretation system developed as part of this work suggests that general purpose tools are feasible for automated story interpretation when based upon combined knowledge-based and case-based processing.

3.0 References

[1] Basili V, Caldiera G and Rombach H. The Experience Factory, *Encyclopedia of Software Engineering*, Second Edition, Vol. 1, John Wiley and Sons, 2002, 511-518.

[2] Basili V, Lindvall M, and Shull F. A Light-Weight Process for Capturing and Evolving Defect Reduction Experience, In *Proc. 8th IEEE Intl Conf. Engineering of Complex Computer Syst*, IEEE, 2002, 129-132.

[3] J. Dix, S. Kraus, V. S. Subrahmanian. Agents dealing with time and uncertainty, Proc. AAMAS 2002: 912-919.

[4] J. Dix, F. Ozcan and V.S. Subrahmanian. Optimizing heavily loaded agents, draft manuscript.

[5] T. Eiter, V.S. Subrahmanian and G. Pick. (1998) Heterogeneous Active Agents, I: Semantics, Artificial Intelligence Journal, Vol. 108, Nr. 1-2, pps 179–255.

[6] J. Gehrke, F. Ozcan and V.S. Subrahmanian. (2002) Tasking Networks of Sensors, draft manuscript.

[7] T. Hammel, T.J. Rogers, B. Yetso. Fusing Live Sensor Data into Situational Multimedia Views, Proc. 2003 Intl. Workshop on Multimedia Information Systems, Ischia, Italy, May 2003.

[8] F. Ozcan and V.S. Subrahmanian. Partitioning Activities for Agents, Proc. 2001 Intl. Joint Conference on Artificial Intelligence, Seattle, WA, Aug. 2001, pages 1218-1228.

[9] Fatma Ozcan, V. S. Subrahmanian, Leana Golubchik. Optimal Agent Selection. KI/GAI 2001: 2-17. Invited talk.

[10]Peng Y and Reggia J: *Abductive Inference Methods for Diagnostic Problem-Solving*, Springer-Verlag, 1990.

[11] Reggia J: Abduction, *Encyclopedia of Artificial Intelligence*, S Shapiro (Editor in Chief), Wiley-Interscience, 1992, 2-3.

[12] V.S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan. R. Ross. Heterogeneous Agent Systems, MIT Press.

Appendix A
Integrating Knowledge-Based and Case-Based Reasoning

Timur Chabuk[1], Mark Seifter[3], John Salasin[1,4] and James Reggia[1,2]

Computer Science Dept.[1] & Inst. Advanced Computer Studies[2], University of Maryland
Electrical Engineering and Computer Science Dept.[3], MIT
Information Processing Technology Office[4], DARPA

Abstract: There has been substantial recent interest in integrating knowledge based reasoning (KBR) and case-based reasoning (CBR) within a single system due to the potential synergisms that could result. Here we describe our recent work investigating the feasibility of a combined KBR-CBR application-independent system for interpreting multi-episode stories/narratives, illustrating it with an application in the domain of interpreting urban warfare stories. A genetic algorithm is used to derive weights for selection of the most relevant past cases. In this setting, we examine the relative value of using input features of a problem for case selection versus using features inferred via KBR, versus both. We find that using both types of features is best (compared to human selection), but that input features are most helpful and inferred features are of marginal value. This finding is surprising to us, but it supports the idea that KBR and CBR provide complimentary rather than redundant information, and hence that their combination in a single system is likely to be useful.

INTRODUCTION

In many application fields, expert-level problem solving naturally involves reasoning from a combination of both general knowledge and individual past cases. Well-known examples of this occur in legal reasoning, medical diagnosis and management, military tactical planning, software engineering, and related areas. From the viewpoint of those developing AI systems intended as decision aids, the need for reasoning from both general knowledge and individual cases has led to substantial recent efforts to find ways to integrate these two approaches within a single framework (reviewed in Marling et al, 2002), and this continues to be an active research area today.

In this context, we are investigating the feasibility of creating an application-independent approach to interpreting multi-episode "stories" that combines a variety of AI reasoning methods (rule-based reasoning/deduction, cause-effect reasoning/abduction, Bayesian inference, constraint-satisfaction problem solving, etc.) with the retrieval of past related cases. Our goal is to implement a system that, given an application-specific knowledge base plus a database of past cases represented in terms of the same features, is able to generate inferences about new situations by concurrently using both knowledge-based reasoning and examination of past cases. This is an ambitious goal that involves addressing a number of challenging issues related to understanding narration [Herman, 2003]. A central idea in this is that specific human-readable knowledge descriptions, written in a simple but formal knowledge representation format, contain sufficient

information about an application area's ontology and terminology to enable a natural language "story interpretation system" to be generated automatically.

In this paper, we focus on one aspect of such a general system, the issue of whether the inferences made by a knowledge-based reasoning process can help guide the identification of the most relevant past cases during problem solving. Effective retrieval of related cases is widely recognized to be very important to successful applications of case-based systems [Pal & Shiu, 2004] and it is one way in which synergistic integration can occur. More specifically, we focus here on the issue of the relative value of input features of a problem versus inferred features in guiding the retrieval of the most relevant past cases. By *input feature*, we mean an evident/observable aspect of a specific problem that serves as input to a decision aid (e.g., for a medical diagnosis system, a patient's age or a symptom), while *inferred feature* refers to an inference made by the system (e.g., diagnosis, recommended treatment, or prognosis). While our approach is intended to be general in nature, for concreteness and because our most recent attention has focused on this topic, in the following we present our work in the context of a specific application, the understanding of multi-episode stories involving urban warfare. There are many sources of such stories available in natural language format (e.g., [Antal & Gericke, 2003; Grau, 1996; Keegan, 1994]), and it is unlikely that any person can memorize the large volume of information and lessons they contain. Thus, if relevant episodes could be quickly identified by an automated system, they would provide a rich source of potentially useful information for military commanders who must make real-time tactical decisions.

METHODS

Our work on effective case retrieval is being done within the context of a broader study, as follows. An application-specific story interpretation system built within our framework works as illustrated in Figure 1. A user supplies a narrated description of an *episode*, a set of events that have occurred in the application domain. A natural language (NL) parser translates the episode into a set of input features, and from these the inference method(s), such as rule-based deduction, derive various conclusions (inferred features) using domain specific knowledge. These conclusions, plus the most relevant known past related cases (or episodes), are retrieved.

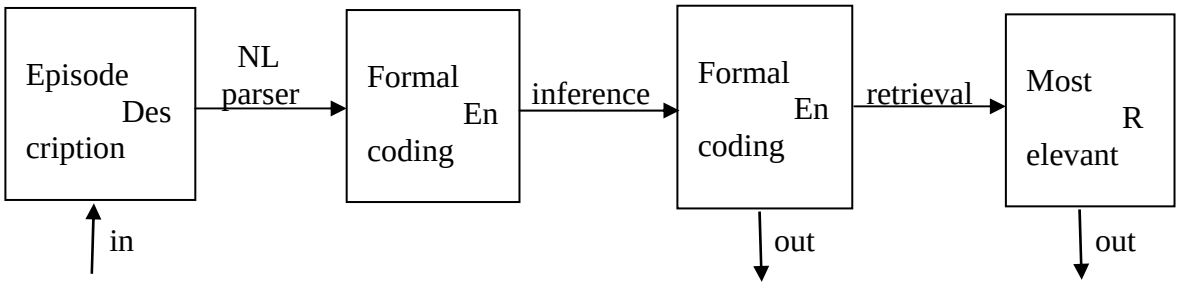

Figure 1: User's view of an application-specific "story understanding system".

An application system like that described above (Figure 1) is built by a domain-independent *constructor* as illustrated in Figure 2. The constructor takes two inputs, an application-specific source file ("knowledge base"), written in a simple knowledge

representation language, and a database of past multi-episode stories. The constructor uses this information to generate the NL parser, inference mechanisms, and case retrieval software needed in the application-specific story interpretation system (Figure 1), and encodes the cases in terms of the source file's attributes for later retrieval. At present, the constructor is implemented, but more work is needed on the urban warfare source file in order for our keyword parser to be able to parse all of the cases. As a result, the encoding of some cases for the experiments described below has been done manually.

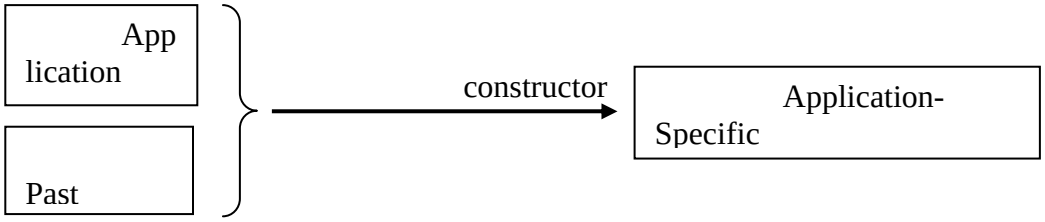

Figure 2: Story interpretation systems like that shown in Figure 1 are constructed automatically from an application-specific source file and a database of past episodes.

For example, for the urban warfare domain, the current source file encodes a set of input and inferred attributes, along with their possible values, plus knowledge in the form of production rules and simple descriptions. The attributes form a hierarchy that is implicitly defined by the encoded knowledge. Attributes are defined as being single or multiple-valued. There are forty-three input attributes and thirteen inferred attributes in the existing urban warfare source file. Most knowledge is in the form of production rules, but for two of the inferred attributes simple pattern matching and scoring is used. In some cases, the inferred features (attribute value assignments) represent abstractions of the input features. Here is an example of an input attribute named "arms" in the source file:

arms [mlt]:
small arms [synonyms: pistols, hand guns, rifles, guns],
light machine guns [synonyms: submachine guns, automatic rifles],
anti-tank weapons [synonyms: rocket propelled grenades, RPG], …

This declaration conveys that "arms" is an attribute/property of a story that, for a specific problem, can simultaneously take on multiple ("mlt") possible values such as "small arms" and "light machine guns". Ontological information is implicitly present in the synonym declarations, so the NL parser (Figure 1) can recognize that the presence of "RPG's", for example, means that (arms = anti-tank weapons) is an appropriate interpretation. An inferred attribute, such as

assessments [mlt]:
enemy likely has outside support,
civilians at great risk, …

is defined in a similar fashion, but has its value determined by the inference mechanism rather than the NL parser, e.g., via rule-based deduction. Rules are defined in terms of attribute values. For example,

IF (enemy organization = militia) AND (enemy technological level = high),
THEN (assessments = enemy likely has outside support)

is a rule in the current knowledge base. This rule indicates that, if an irregular military group ("militia") has high tech weapons, the system should make the inference that the group probably has outside support.

In the current implementation of our system, an abductive keyword natural language parser (Figure 1, on the left) is used to extract the values of the input attributes from stories written in natural language text. As the constructor processes an application-specific source file, each word encountered is indexed to the attribute/value it helps to name. The resultant NL parser processes a subsequent story word by word, and as each word is processed, it evokes the set of all possible senses (assertions about the value of known attributes) of that word. Words can be very ambiguous. For the urban warfare domain, the system evokes five different senses for the word "fire", such as the phenomena of friendly fire or the tactic of setting buildings on fire. The presence of a word in a story is explained in a context-sensitive fashion by making an assertion about an attribute's value. Disambiguation of the word's meaning is done using a parsimonious covering process, a type of abductive reasoning [Josephson, 1994; Reggia, 1992] that results in a set of assertions representing the interpretation of the story. For example, the presence of the word "fire" may be explained by the assertion (our tactics = set fire to building) or several other assertions.

For the urban warfare scenario used in our experiment below, a case-base was established containing 30 episodes from ten different stories. The stories spanned the time period from just before World War II through the mid 1990's, and were taken primarily from Russia's Chechen Wars [Oliker, 2001], the Wikipedia online encyclopedia [Anon, 2003], and City Fights [Antal, 2003].

An assembled system like that for the urban warfare scenario identifies the past most relevant episodes by measuring the similarity of every episode in the case-base using a linearly weighted distance metric. This *similarity function* takes two episodes and outputs a numeric score indicating their degree of similarity in terms of their attribute values. The contribution of an attribute to this similarity score depends on whether the attribute is single or multiple valued, and whether it is nominal or ordinal. Multiple valued attributes are treated as being a collection of single valued attributes, where each value can be either present or absent. In the similarity function, a positive real-valued weight between 0 and 10 is associated with each single-valued attribute, and with each possible value of multiple-valued attribute. The similarity of two episodes on each attribute are multiplied by their respective weights, and then summed to produce the final overall similarity score. Since the optimal weights for case retrieval are not known a priori by our application-independent system (Figure 2) for a specific application, our approach is to include with each episode in the case database the identity of the other single most similar

case that is present. This best match identity is specified by a person at the time of case-base creation and represents the "gold standard" for the evaluation described below.

As others have done [Dubitzky, 2001], we used a genetic algorithm (GA) to automatically evolve the set of weights to be used by the similarity measure in the resultant application-specific system. In the urban warfare example used here, the GA population was 600 haploid chromosomes, each a vector of real-valued weights to be used by the similarity measure. Tournament selection of reproducing parents (with elitism) was used, with probability of double-point crossover 0.35 and of mutation 0.60. If selected for mutation, each real number in the chromosome had a 10% chance of being replaced with a random number between 0.0 and 10.0. The fitness of a chromosome's set of weights was based on how well they correctly identified the a priori human-identified most similar other case in the case base for each and every existing case. The genetic algorithm was run for 300 generations and the most fit set of weights in the final population was used in the application-specific similarity measure.

While the inferred features are intended in and of themselves to be useful to a human operator of the system (leftmost "out" arrow in Figure 1), we consider here whether or not they are also useful when trying to automatically assess similarity of episodes for case-retrieval. To address this issue, we evaluated the ability of the story interpretation system to identify the best match in the case base for input episodes when using all attributes, input attributes only, and inferred attributes only. This allowed us to examine one aspect of the impact of integrating knowledge-based reasoning with case retrieval. A standard leave-one-out strategy was used to evaluate how well our methods for evolving attribute weights with specific data generalize. The episodes from one story were removed from the training data and an optimal set of weights evolved using the rest of the episodes in the case-base as training data. That set of weights was used to find the most similar episodes to the episodes that had been removed, and for each of the removed episodes we recorded how highly the system ranked its actual most similar episode (according to the "gold standard"). Attribute weights for the similarity measure were independently evolved using all of the attributes, only the input attributes, and only the output attributes. These three experiments were repeated 10 times each, each time excluding the episodes from a different story from the training-data. We also performed the three experiments while leaving out no story episodes in order to establish a baseline of optimal performance.

RESULTS

The set of weights obtained by the GA for the urban warfare case base similarity function indicated that some attributes are much more important for assessing similarity than others. The 235 weights found were fairly evenly distributed over the full 0.0 – 10.0 range of possible weight values. Some of the most highly weighted features in this specific application, each having a weight above 9.8, were (our tactics = blitzing attack), (enemy abstract tactics = sewer battle), (arms = missiles), and (results = failed to expel invading force). Some of the lowest weighted features, each having a weight below 0.04, were (our tactics = aerial bombardment), (assessments = possible hasty attack), and

(civilian actions = civilians serve as guides). Input features had an average weight of 4.43, while inferred attributes had an average weight of 4.17.

Our domain-independent approach to story interpretation appears, in limited testing to date, to work reasonably well. For example, the following excerpt from a made-up urban warfare story that is not in the case base,

We were a conventional army controlling a city in the middle of a war zone. The enemy militia was trying to take the city, and we had to defend it. We had virtually no re-supply lines and were poorly supplied. We had some small arms, some heavy machine guns, and some IEDs. In anticipation of the enemy advance we set up some fortified positions with heavy machine guns from which we could strafe the streets with fire. The public was not entirely supportive of the battle, so we wanted to try very hard to avoid civilian casualties. In the morning, three enemy tank columns entered the narrow streets of the city. We had to expel this invading force. From the rooftops, we began dropping grenades on to the enemy tanks. We detonated explosives that we had planted in buildings, exploding the buildings on to the approaching forces. The enemy attack choppers were ineffective in the battle as we kept them at bay with our SAMs….

is readily processed by the interpretation system. The abductive keyword parser processes this text and extracts the correct values for all of the input attributes, such as (arms = stingers), (arms = homemade explosives), (public opinion = public skeptical), and (tolerance for civilian casualties = low). Inferred attribute values include (inferred quality of intelligence = adequate), (quality of combined arms usage = poor), (assessments = enemy likely has outside support), and (suggested course of action = cut off enemy's outside support).

The similarity measure using the best set of weights derived by the GA identifies two episodes from a 1948 battle in the mandate of Palestine as being the most similar to the example episode above. Examining these two retrieved past cases, their similarity to the episode given above is readily apparent. In both cases an invading force of tanks is repelled by attacking them from above with explosives and by exploding buildings onto the tanks as they pass by. Examination of these stories by a human operator has the potential of leading to a number of new and useful inferences. For instance, in the second episode from the Palestine story, the enemy learns to deploy infantry support along with tanks, and this could be a useful point for a military commander. While our system currently does not have the ability to make inferences from retrieved cases, methods used in past case-based reasoning systems could be effective in this regard.

To assess in more general terms the effectiveness of the GA in deriving appropriate weights for application-specific case retrieval, we generated these weights using just the input features, just the inferred features, and both (10 trials with each), using the 30 cases in the urban warfare case base. The results of the similarity function's

performance are given in Table 2. When the GA evolved weights using all of the stories in the case base, and then the rank assigned to the a priori human-specified best matching case was determined for each case in the case base using the similarity function, the mean rank was 1.37 if all features were used, 1.43 if just input features were used, and 2.87 if just inferred/output features were used. While using all features thus did marginally best, the contribution of the inferred features was almost negligible.

Table 2: Performance of similarity function using different sets of attribute.

	ALL STORIES			LEAVE ONE OUT		
ATTRIBUTES USED IN TRAINING	ALL	INPUT	OUTPUT	ALL	INPUT	OUTPUT
AVG. RANK GIVEN TO MOST SIMILAR	1.37	1.43	2.87	3.33	3.37	6.20
STANDARD DEVIATION OF RANK	0.72	0.73	2.26	2.51	2.76	6.58

To test the ability of this approach to generalize to new cases, we repeated the above study, but now using a leave-one-out strategy. In this situation, the mean rank was 3.33 if all features were used, 3.37 if just input features were used, and 6.20 if just inferred/output features were used. Though not as accurate as when all episodes are used, it suggests that one could use a strategy of retrieving three or four cases in general. The difference in performance between when all features were used and when just input features were used is not statistically significant. However, the difference between using just output features and either all features or just input features is statistically significant, at a higher than 95% confidence level.

DISCUSSION

In this project, we explored integrating reasoning from both general knowledge and from individual cases within a single framework. To this end, we investigated the feasibility of creating an application-independent approach to interpreting multi-episode "stories" that combine knowledge based reasoning methods with the retrieval of past most similar cases. In this paper we focused on whether the inferences made by a knowledge-based reasoning process can contribute to identifying the most relevant past cases during problem solving. Specifically, we explored the value of input features of a problem versus the value of inferred features in the retrieval of most relevant past cases.

For the urban warfare domain, we found that the input features of an episode are more important than the inferred features when attempting to assess the similarity of episodes. Our similarity measure performed significantly better when using only input attributes to assess similarity than when using only inferred attributes. This suggests that there is some information or relationships among the input attributes that our current knowledge, mostly in the form of production rules, simply does not capture. If this is correct, it supports the hypothesis that integrating knowledge based and case based reasoning is synergistic, because it suggests that the information in specific cases may be different from that inferred using general principles in an application domain. This was illustrated by the specific example case/story above where general rules deduced, for

example, that the enemy force probably had outside support and that a reasonable course of action would be to try to disrupt this support, while both retrieved cases included the sensible point that infantry should accompany tank incursions in an urban setting to help prevent attacks from above. In a limited sense, this can be viewed as a kind of "ensemble reasoning". Of course, it is entirely possible that a different set of inferred attributes might be more informative. The inferred attributes that we used were based on an a priori conception of useful inference without consideration of their utility for case retrieval. Our results are also limited to the specific domain of urban warfare, and it is unclear whether they will generalize to other areas like medical or legal reasoning.

The GA-derived weights used in the similarity function did not generalize extremely well. As expected, the ability to identify the a priori human-selected most similar case declined when a leave-one-out strategy was used to evaluate case retrievals. However, the results were still quite reasonable if one is willing to allow a system to retrieve a few apparently best cases rather than just the single best case. Using both input and output features to assess similarity of cases continued to result in the highest performance, but with inferred features being of negligible value.

An important direction for future work is the integration of additional reasoning methods with case-based reasoning. Some central questions are whether alternative reasoning methods or different inferred attributes can help improve case retrieval, how the results described here will generalize to domains other than urban warfare, and whether knowledge-driven inferences can contribute to more powerful case-based reasoning in general through better case adaptation/modification and storage. A more objective way to evaluate the performance of these tasks is also desirable. In particular, cost effective methods are needed for replacing the "gold standard" with more objective and precise ways of measuring relevance between cases. One possible approach would be the development of a simulated environment where agents could use case-based reasoning to solve problems. In such a system, the true relevance of past stories to a current problem could be determined by measuring the performance of the agent in addressing a new problem when recalling different past cases.

REFERENCES

[1] Antal J & Gericke B. *City Fights*, Ballantine, 2003.

[2] Anon. http://www.wikipedia.org/wiki/Battle_of_Mogadishu,Wikipedia (25 Jan 2006).

[3] Dubitzky W & Azuaje F. A genetic algorithm and growing cell structure approach to learning case retrieval structures, in *Soft Computing in Cased Based Reasoning*, S. Pal et al, eds), Springer-Verlag, 2001, 115-146

[4] Grau L. *The Bear Went Over the Mountain*, National Defense University Press, 1996.

[5] Herman D (ed.) *Narrative Theory and Cognitive Sciences*, CSLI, 2003.

[6] Josephson J & Josephson S (eds.) *Abductive Inference*, Cambridge Univ. Press, 1994

[7] Keegan J. *A History of Warfare*, Vintage, 1994.

[8] Marling C, Sqalli M, Rissland E, Munoz-Avila H, & Aha D. Case-Based Reasoning Integrations, *AI Magazine*, 23, 2002, 69-86.

[9] Oliker, Olga. *Russia's Chechen Wars 1994-2000*, RAND, 2001.

[10] Pal S & Shiu S. *Foundations of Soft Case-Based Reasoning*, Wiley-Interscience, 2004.

[11] Reggia, J. Abduction, *Encyclopedia of AI*, John Wiley, 1992, 2-3.

Appendix B
Software Inspections as Choice Points: Necessary Data to Provide Decision Support in Software Development

Forrest Shull
Fraunhofer Center – Maryland
fshull@fc-md.umd.edu

Of the many ways of modeling software development, perhaps one of the most interesting is as a series of *choices*. There are certainly a large number of different decisions to be made, in order to achieve the goal of delivering agreed-upon functionality on-time and within budget: Before the project begins there are questions about which lifecycle model to use, how to staff the project, etc. During the project, decisions are more focused on whether the project is on track and how best to expend scarce resources to achieve the desired goals: For example, if the quality of products being produced is less than optimal, the project may choose to put more effort into quality assurance, detecting defects that can then be corrected before the project goes any further. Conversely, if quality is under control, the project can allocate those resources to construction activities, perhaps giving the team the chance to implement functionality that was otherwise considered optional for the current release. The primary difficulty is in accurately assessing whether the quality of the system is on-target or not.

As a mechanism for facilitating decisions about whether to allocate resources into or away from quality assurance techniques, software inspections can provide information that is more accurate than that gained from other means. Some authors [Gilb99] have suggested that the information gained during an inspection about the current state of the system under development is as valuable as the actual defects detected and corrected. Inspections also have the advantage that they can be applied during any phase and to any type of work product or support artifact.

Currently, decision-making based on inspection results is necessarily rather subjective, based on the decision-maker's expertise and past experiences. We have been exploring whether a more objective, empirically-based basis could be provided, in which decisions based on inspection data could be based on reasoning about what occurred on previous projects, to make informed decisions about what is likely true about the current project.

Software Inspections

Software inspections are technical reviews whose objective is to increase the quality and reduce the cost of software development by detecting and correcting errors during the process. A formal process is used to provide rigor in this task, and checklists or other work aides (such as scenarios) are typically used to keep inspectors focused on important aspects of the document under inspection. Since inspections depend on human analysis and reasoning, they can be applied to just about any document created

during the software lifecycle. Ideally, they would be applied as early as possible to remove errors before they amplify into larger and more costly problems downstream.

Software inspections take as input a work product (or portion of a work product) to be inspected, and require personnel with sufficient expertise and time to work as inspectors.

As output, software inspections yield a list of defects to be corrected by the author. They also produce supplementary metrics, usually consisting of type information associated with each defect, the number of participants who were involved, and the total effort required to perform the inspection.

Using the inspection outputs to support decisions involves two types of judgment calls. The first is, how effectively the inspection was conducted – that is, how likely it is that the list of defects found during the inspection represent the majority of the defects actually extant in the document. The second judgment call involves reasoning about what those defects actually mean for the system.

Assessing Inspection Effectiveness

Many empirical studies have documented the effectiveness of inspections for finding defects in a cost-effective way across the organization, e.g. [Fagan86, Russell91]. A useful rule of thumb, based on data from across many organizations, is that a given inspection will find between 60% and 80% of the defects currently in the document [Shull02].

Although there is still investigation into some of the finer points (such as the type of inspection training that produces the best results [Land05]), the broad brushstrokes of what makes an effective inspection are well understood. Some example guidelines include:

- Inspectors should have sufficient technical expertise to analyze the work product under review;
- The team of inspectors should be neither too small (which results in insufficient debate and refinement of the list of issues to be fixed), nor too large (which results in unmanageable discussion and might dissuade some participants from making a contribution);
- A formal process should be followed to avoid "cutting corners," e.g. generating issues during the inspection that are never fixed;
- Inspectors should have sufficient time to devote to preparation;
- Management should not be part of the inspection, as this can shift the focus from improving the work product to saving face;
- One inspector should be tasked with recording issues generated by the inspection team, so that fixes can be verified;
- A trained moderator should facilitate the discussions and ensure that participants prepare properly for their roles.

Assessing the actual practice of an inspection against such guidelines is an important component of knowing whether the results represent an accurate assessment of

the quality of the work product inspected. If important guidelines were not followed (for example, if inspectors without appropriate knowledge were used or they were not given enough time for preparation) it is possible that the defects found by the inspection greatly under-count the number of defects existing in the work product.

Although the above issues have not been analyzed separately for their contribution to inspection success, there have been no data from any environment which indicate that different guidelines apply in different types of organizations or for different application domains.

An important issue that is *not* considered here is whether the inspection results include a large number of false positives – that is, issues that were reported by the inspection team but that do not really represent problems in the system – in addition to useful information about existing defects. Since inspections are a team-driven activity, in that they rely on a team of inspectors with different expertise to reach consensus that an issue is really a defect before it is reported, it is also assumed that a well-performed inspection will minimize the number of false positives, and one that deviates from the guidelines will similarly produce untrustable results by over-estimating the number of quality problems in the system.

Reasoning about the Current State of System Quality

Having done an inspection and found out something about the number of defects in a specific work product, what kind of choices could then be made based on those results? Regarding the specific work product that was inspected, the team lead can choose one of the following options:

1. If the number of defects was within acceptable bounds: Fix the defects found and continue without further QA on this product.
2. If the number of defects was high enough to cause concern: Fix the defects found and then do additional QA on this product.
3. If the number of defects was excessively high: Throw out much of the development work that was done and re-develop from scratch. (Although extreme, this is based on the hypothesis that very defect-prone components tend to be defect-prone throughout their lifespan – it is difficult to patch up components that had unacceptably low quality to begin with.)

Furthermore, if the defect density is found to be unacceptably high and this product is considered to be fairly representative of the quality of the larger system, the inspection output might also convince us that:

4. The percentage of effort allocated to quality assurance on the system as a whole should be increased.

To support this type of decision-making, we investigated the use of "stories." A story in this context is a way of representing knowledge about a situation as a series of discrete episodes describing events that happened in chronological order. Episodes are separated by choice points. In other words, a story may contain episode *x* which describes the situation at some moment at time, when a choice is made; this leads to episode *x+1*, at which point a new choice is made, and so on. In theory, if I have a new situation which is roughly analogous to a series of episodes in an existing story, ending with episode *y*, then I have some confidence for believing that making the same choices

is likely to recreate results in my current situation analogous to those recorded in episodes starting with *y+1*.

Using this scheme to support software development decisions, we treat software development as a sequence of episodes, where performing an inspection is the latest episode in the sequence. The inspection episode has to be characterized according to the defects that were detected, along with the size of the work product inspected. From these measures we could understand the defect density of the inspected portion of the system, which tells us something about the quality of system at this moment in time. With this information, we could examine any known stories about inspection applications to look for similar situations, and reason about the types of actions that would be appropriate for systems with this level of quality at this point in the software development lifecycle.

To accurately make this comparison, however, the *types* of defects detected by the inspection also have to be characterized. As an extreme example for why this is necessary, suppose that a code module *A* is inspected and based on the defects detected it appears that the defect density is one defect per 1K lines of code. Let us assume that there exists a story about inspections applied to code module *B*, from a similar system, where *B* has a similar size and complexity to *A* and resulted in a similar measure of defect density. Will later experiences on *B* provide any indication about what could be expected in later development of *A*? The answer may be yes but only if information about the *types* of defects detected is also comparable. If the defects in *B* were mainly clerical or otherwise easily fixed, while defects in *A* concern important system functionality that was incorrectly specified in the earliest phases of the project, the downstream experiences on both projects may be quite dissimilar indeed.

This necessity of comparing defect types from one project to another forms the major stumbling-block to this work. Although software inspection may well be the software development practice for which the most data exist, no standard classification system for defects is used across a large number of the available datasets.

Data Sources Examined on this Project

From various projects, FC-MD has access to a wide variety of inspection data from numerous contexts. Searching for data that would support the planned analyses, we investigated:

- NASA software development bug/change trackers:

Many NASA projects use tracking systems to store and analyze the known issues on a given project during the course of development. Each record in the database corresponds to a single issue that must be changed or repaired in the given system. Thus, the database taken as a whole contains a record of all unplanned changes that arose over the lifetime of the system development. For a large project, there will likely be hundreds of such records.

Although the exact information recorded for a defect/change record varies from system to system, at a bare minimum projects generally record: A description of the issue to be addressed, the date the issue was discovered and tracked, how the issue was found, the severity associated with the issue, how the issue was fixed, and the date the implemented fix was reviewed and approved.

Such change tracking databases seemed to be a rich source of analysis for the planned analysis. Although the databases will contain information about quality issues found by mechanisms other than inspections, the inspection results themselves are entered into the bug tracker so that they could be followed and closed by project personnel. However, these databases suffered from two major problems for our work: First, quality issues were generally recorded starting only with the implementation phase of the project. Although requirements and design issues could still be found and recorded at this point, the database was missing any record of the majority of quality issues that were discovered during those activities. Secondly, the databases in general were missing any meaningful categorization of the defects. Not only was no single classification scheme used to organize the issues being tracked, but the information stored concerning what the issue was and how it was fixed were at a very detailed level that was meaningful only to project personnel. Therefore, it was not possible to provide any comparison between new projects and old projects on the basis of the recorded issues.

- Inspection experiments:

Personnel from the Fraunhofer Center – Maryland and University of Maryland have, in combination and separately, run a number of experiments comparing the effectiveness and efficiency of varying inspection approaches for detecting defects, in all phases of software development. The repository of experimental results contains another rich source of inspection data. Because each experiment applied inspection to a work product with a known number of defects, it is possible to measure accurately what percentage of the issues extant in the document were found by each application of inspection. Because data is recorded on each subject's background, it is also possible to judge whether effectiveness varies according to attributes of the person applying it, for example, his or her amount of previous experience in inspection teams or with software development in general. Furthermore, because the experiments were designed to be comparable with other published work, a common taxonomy of defects was used across the studies.

The weakness of this dataset for our purposes here is that the inspections are applied in an academic environment – that is, the work products under inspection were not actually being used in a real development project. For this reason it is impossible to follow the products forward in time to understand the effect that the correction of the known defects had on the overall quality of the system: Whether the defects detected would have saved significant development effort or the defects missed had the potential to greatly increase the amount of rework effort needed later in the project. Thus the usefulness of this dataset for the desired decision-support was minimal; it could be concluded from this data that appropriately-conducted inspection would detect a majority of the extant defects but not how this contributed to the overall budget and quality of the project.

- Inspection databases from large government and commercial organizations:

Organizations that invest in inspection programs often maintain a cross-project database of inspection results. Such a database typically contains one record for each inspection, recording information such as the project and module on which the inspection was applied, the size of the work product inspected, the number of participants involved

and the total amount of effort spent on the inspection, and the number of defects detected and fixed as a result of the inspection. Such databases are used by the organization for baselining and monitoring inspection results to spot problems. For example, when a team has just conducted an inspection the output can be compared to baselines recorded for similar projects to understand whether this process has required much more effort than has traditionally been the case, or whether team leads are trying to inspect too much material in too little time. By making certain assumptions about the amount of rework saved for different types of detected defects, it is also possible to use the data to analyze the expected return to the organization for the effort expended on the inspections [Kelly92, Madachy95].

Again, such databases are useful for showing the contribution of inspections in general and demonstrate their usefulness as a quality assurance technique. However, they also do not provide information about the same module over time so that the quality of a module after an inspection and rework could be assessed. While theoretically they could contain information on multiple inspections of the same module, in practice this did not occur in the datasets to which we had access. Thus, it was not possible to judge the contribution of any inspection to downstream quality aspects of the same part of the system.

- Software Engineering Laboratory data:

The Software Engineering Laboratory (SEL) at NASA's Goddard Space Flight Center had a long history of data collection in the area of software development, which contributed to recognized success in the area of software process improvement [Basili95, Basili02]. While the database is no longer maintained, archives exist of the data and numerous publications have been written summarizing various subsets of its contents.

During investigation of the database it was clear that forms for inspection data collection existed[1] and had been used by developers. However, in interviews with SEL data analysts it was discovered that the inspection data collected was from projects using the Cleanroom methodology, a distinct software development methodology based on formal models and correctness verification that is not in widespread use. Efforts to reconstruct the database contents were thus discontinued, since the predictive power of this data for more traditional software development environments was judged to be very slight.

Recommendations for Necessary Data

As this was a small, exploratory study, it was perhaps not surprising that the data archives already available did not happen to match the needs of the new, proposed analysis. However, the resources summarized above constitute a large body of data for any practice in software engineering, compiled by researchers at UMd and FC-MD over years of work. It is not likely that there are great reserves of other data sets available on inspections, or that more data would exist for other software development practices.

The above list suggests that longitudinal data from industrial or government development projects is needed for the planned analysis – although inspection results in an academic environment can be measured and categorized most accurately, it is difficult to get information about decisions being made on the basis of

[1] See http://sel.gsfc.nasa.gov/website/exp-factory/datacol-process/sample-forms/IDCF_np.html

those results based on experience. If data is to be gathered for the planned analysis, it would need to be instrumented as part of an ongoing, “live” development project.

Since the purpose of the planned decision support is to reason about software quality – and whether if, judging by early indicators such as inspection results, it appears to be at a sufficient level that would enable end-product goals to be met – we would need some more objective indicator of quality in the particular portion of the work product being inspected. Test data is the most likely choice for this, as this is the final quality assurance check before the software is released. Although test does not catch all of the remaining defects, it is usually considered to catch a majority of those defects and project managers will often place extra resources into test if early test results show more defects remaining than was expected. The final quality of a given portion of a software system could be approximated by the number of defects found during test, the amount of rework effort that has to be done to fix defects found during test, and the amount of time that portion of the software system spends in test. High values for any of those three indicators could be taken as evidence that the quality of that portion of the system was inadequate and that better decisions could have been made about allocating early QA activities.

Thus to actually conduct the planned analysis would require negotiating with a development project to collect data on both in-process inspection results from various phases (which would include defects that would presumably have to be categorized by the research team) along with test data and test effort. Unfortunately, as this data collection would span the entire life of the project, it would be a substantial undertaking. However, no existing dataset with all of the required information is known.

References

[Basili95] Basili V., M. Zelkowitz, F. McGarry, J. Page, S. Waligora, and R. Pajerski, SEL's software process-improvement program, IEEE Software 12, 6 (1995) 83-87.

[Basili02] Basili V., F. McGarry, R. Pajerski, M. Zelkowitz, Lessons learned from 25 years of process improvement: The rise and fall of the NASA Software Engineering Laboratory, IEEE Computer Society and ACM International Conf. on Soft. Eng., Orlando FL, May 2002.

[Fagan86] Fagan, M. Advances in Software Inspections. IEEE Transactions on Software Engineering, vol. SE-12, no. 7, July 1986.

[Gilb99] Gilb, T. “Software Inspections are not for quality, but for engineering economics.” Unpublished draft article. http://www.gilb.com/Download/IEEESWAr.pdf

[Kelly92] Kelly, J., Sherif, J., Hops, J. An Analysis of Defect Densities Found During Software Inspections. Journal of Systems and Software, 1992; 17: 111-117.

[Land05] Land, L., Tan, B., and Bin, L. Investigating Training Effects on Software Reviews: A Controlled Experiment. *Proceedings of ISESE05*, Noosa Heads, Australia, November 2005.

[Madachy95] Madachy, R. Measuring Inspections at Litton, Software Quality Assurance, Ridgetop Publishing, Silvertown, OR, Vol. 3, No. 3, 1996 and Proceedings of the Sixth International Conference on Applications of Software Measurement, Orlando, FL, Software Quality Engineering, October 1995

[Russell91] Russell, G. Experience with Inspection in Ultralarge-Scale Developments. IEEE Software, 8(1): 25-31. 1991.

[Shull02] Shull F., Basili V. R., Boehm B., Brown A. W., Costa P., Lindvall M., Port D., Rus I., Tesoriero R., and Zelkowitz M. V., "What We Have Learned About Fighting Defects", In Proceedings of 8th International Software Metrics Symposium, Ottawa, Canada, IEEE, June 2002, pp. 249-258.