ARMY RESEARCH LABORATORY

Automated Policy-based Asset Cuing Using SQLite

by Jesse Kovach and Robert Winkler

ARL-TR-5779 **November 2011**

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Adelphi, MD 20783-1197

ARL-TR-5779 **November 2011**

Automated Policy-based Asset Cuing Using SQLite

Jesse Kovach and Robert Winkler
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
November 2011	Final	January to December 2010

4. TITLE AND SUBTITLE
Automated Policy-based Asset Cuing Using SQLite

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Jesse Kovach and Robert Winkler

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: RDRL-CII-B
2800 Powder Mill Road
Adelphi MD 20783-1197

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-5779

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

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Unattended assets are being employed in ever-increasing numbers to support current military operations. These assets produce large amounts of data, which is generally handled by presenting it to an operator or analyst for examination. If follow-on actions are required, these actions must be manually initiated by the operator or analyst. Automated asset cuing systems that recommend or initiate follow-on actions by applying policies or rules to incoming sensor data can help reduce the workload on the system operator. This report presents an asset cuing system built around the SQLite database engine that uses the Cursor on Target (CoT) message format to receive sensor data and Structured Query Language (SQL) to express cuing policies. Cuing policies are stored as triggers in the database schema and invoke user-defined functions written in C# to control sensors, robots, unmanned aerial vehicles, and other assets directly from SQL.

15. SUBJECT TERMS
Cross cuing, cuing, policy, UGS, SQL, SQLite

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Jesse Kovach
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UU	46	19b. TELEPHONE NUMBER (*Include area code*) (301) 394-3988

Contents

List of Figures

Acknowledgments

The authors wish to thank Steven Choy and William Gollsneider for developing portions of the software used for this effort.

1. Introduction

Unattended assets, such as unattended ground sensors (UGS), unmanned aerial vehicles (UAVs), and unmanned ground vehicles (UGVs), are being employed in ever-increasing numbers to support modern military operations. The proliferation of these assets has led to a corresponding increase in the amount of images, video, target detections, and other data generated by them. The traditional approach to handling this information is to present it to an operator or analyst, who decides what, if any, action to take in response to the information. If follow-on actions are desired, such as tasking a UAV to record video at the site of a UGS detection, those actions must be initiated manually and often require coordination among several different operators. As the amount of information generated by unattended assets increases, the workload on the operators and analysts increases as well. Automated systems that process incoming unattended asset data and recommend or initiate appropriate actions based on the data can help to reduce the workload on the system operators.

Some degree of automated triggering and cuing capability exists in currently fielded unattended assets. For example, many UGS systems use a simple activity detection sensor to trigger a camera to take pictures. More sophisticated UGS systems can correlate reports from multiple trigger sensors to determine additional information, such as direction of travel. However, automated processing capabilities in these current systems generally operate only within one sensor site (sensors at site A cannot cue sensors at site B) and do not support interoperability between assets from different vendors, limiting the number of useful tasks that can be performed. To realize the full potential of unattended assets as a force multiplier, an automated cuing system capable of using different types of assets manufactured by different vendors and located at different sites is necessary.

In support of the Sensor and Information Fusion for Improved Hostile Fire Situational Awareness Army Technology Objective (ATO), also known as the Hostile Fire Defeat (HFD) ATO, the Battlefield Information Processing Branch of the U.S. Army Research Laboratory's (ARL) Information Sciences Division developed an automated policy-based asset cuing system. This report documents the development of that system.

2. Requirements

2.1 Example Use Cases

There are a variety of situations where an automated cuing system would be useful. While designing the system described in this report, we considered three types of usage scenarios.

The first use case is to cue a pan/tilt surveillance camera to point to the location of detection from a "tripwire"- type UGS that is not co-located with the surveillance camera. Depending on the location of the sensor producing the detection, different surveillance cameras may be cued. The mapping of sensors to remote cameras can be done either by manually associating a sensor with a camera (sensors A and B cue camera X, sensors C and D cue camera Y) or based on the geographical location of the target detection (cue the closest camera to the target.) This can be modeled as a simple "if A occurs, then do B" policy.

The second use case is to correlate detections from multiple independent tripwires located along a roadway or other path to determine the direction of travel of a target, and keep a record of how many targets were detected traveling in each direction. A more complex version of this use case involves placing a number of independent sensors at the entrances to an intersection and determining which way targets turned at the intersection. This is an example of an "if B occurs within some time period after A, then do C" policy.

The third use case is to process target detection data from a sensor that produces multiple moving target tracks (such as a radar) and cue one or more remote surveillance cameras to track one of the detected targets. This involves selecting a single target to follow from the multiple target tracks, and then sending successive commands to the camera(s) to keep the target in the field of view.

All three of these use cases can be extended to incorporate UAVs, UGVs, or other types of unattended or unmanned assets, in addition to ground-based surveillance cameras.

2.2 Requirements

To guide the development of a cuing system capable of supporting the use cases described in the previous section, we established the following requirements:

- The system must be capable of ingesting sensor detections and asset position and attitude information from multiple sources.
- The system must convert the incoming detection and asset position data to a common intermediate format if it is not already in a common format.
- The system must be capable of generating commands based on detection and asset position events. This implies the need for a policy language, rules engine, or other mechanism to represent the conditions that will lead to the generation of an action.
- The system must be able to generate commands based on single events, combinations of events, or sequences of events over time.
- The system must be able to maintain internal state and use or update this state while processing incoming events.

3. Cuing System Design

3.1 Selection of SQL as the Policy Language

At the start of the cuing system development effort, we decided to avoid defining "yet another policy language" by seeking to leverage an existing policy language or rules engine. We had some prior experience with the KAoS policy system developed by the Institute for Human and Machine Cognition (A. Uszok et al., 2003; A. Uszok et al., 2008; Johnson et al., 2008; Bunch, Bradshaw, and Young 2008), but decided that the complexity of the use cases driving the design of the cuing system did not warrant the level of effort required to incorporate KAoS. Web Ontology Language (OWL) (W3C 2009) and Semantic Web Rule Language (SWRL) (W3C 2004) were also considered and rejected for similar reasons. We also evaluated a simple hard-coded rules engine that we developed for a previous sensor reporting system, but this engine was designed for a specific task and was not flexible enough to support a generalized cuing system.

While evaluating these policy engines, we came to the realization that it may be possible to implement some, if not all, of the cuing conditions for the use cases we were interested in by representing asset event and position data as rows in a relational database using Structured Query Language (SQL) queries to represent the cuing rules. SQL is relationally complete and, thus, has at least the same expressive power as first-order logic. Since we were already comfortable with databases and SQL, and this approach allowed us to leverage existing database engines and tools for rapid development, we ultimately decided to pursue SQL as the policy language for the sensor cuing system.

3.2 Other Systems using SQL as a Policy Language

We then conducted a literature search (mainly as a sanity check) to see if the idea of using SQL as a policy language had been previously considered by others.

(Cook, 2009) introduces a simple domain-specific policy specification language to express authorization policies and then converts the policies into SQL as the policy implementation language. (Didriksen, 1997) took a similar approach for rule-based database access control (although he did not explicitly identify them as "policies" *per se*). (Gencay, Kuchlin, and Schafer, 2007) introduce a simple domain-specific Extensible Markup Language (XML) policy specification language for Storage Area Network configuration, which uses SQL for specifying the policy conditions.

(Weth, et al. 2009) selected SQL as the policy language for their Policy-Based Helping Experiments for practical reasons similar to ours. "First, evaluation is 'for free', i.e., we can use an off-the-shelf database engine as the core of our experiment platform, and, given such a system, we do not have to design and implement any 'policy-evaluation engine' or any

repository for past service decisions ourselves. … Second, SQL is widely known, and learning this language is possible in many ways."

(Barker and Rosenthal, 2002) use stratified logic as a policy specification language for a wide variety of role-based access control policies and demonstrate how these specifications may be automatically translated into SQL. (Calo and Lobo, 2006) point out that most Business Rule Systems are built on an RDBMS framework and use SQL as the policy language. For non-temporal constraint policies, the integrity constraints available with an RDBMS provide an efficient implementation. They also demonstrate that for other types of non-temporal policies the use of the relational calculus provides a reasonable specification language (and consequently, SQL provides a reasonable implementation language). For temporal policies, they suggest introducing a fixed set of temporal connective predicates for use within a relational calculus specification.

As a result of our literature search, we concluded that exploring the use of SQL as a policy language for an automated sensor cuing system is not unreasonable.

3.3 Selection of SQLite as the Database Engine

To implement the cuing system, as with any type of database application, we needed to select a database engine. We had previous experience with two database engines—Microsoft SQL Server and SQLite. SQL Server[1] is a widely used, mature, commercial database engine that exists in desktop, server, and embedded versions, and integrates tightly with the Visual Studio development environment and C# programming language that we prefer to use for application development. However, SQL Server has licensing restrictions and is only available on Microsoft platforms. We anticipate that the cuing system will eventually be ported to an embedded platform, and we prefer to use Linux for embedded applications.[2]

SQLite[3] is a lightweight, public domain, license-free database engine designed for embedded applications. SQLite is provided as part of the base system by most Linux distributions as well as popular consumer device platforms including Apple iOS and Google Android. While the core SQLite engine is written in C, System.Data.SQLite[4], a public domain third party wrapper, provides C# support as well as Visual Studio integration. A pure C# implementation of SQLite is also available.[5] We chose to use SQLite as the database engine to facilitate portability to embedded systems platforms.

---

[1] www.microsoft.com/sqlserver

[2] While this preference may initially seem to be at odds with our use of Microsoft's C# programming language, we have successfully used Mono (www.mono-project.com) to build and run C# programs on embedded Linux systems for previous projects.

[3] www.sqlite.org

[4] http://sqlite.phxsoftware.com

[5] http://code.google.com/p/csharp-sqlite

3.4 Selection of Cursor on Target (CoT) as the Intermediate Data Format

Different types of assets produce data using different, often vendor-specific, message formats and protocols. The cuing system must convert these formats to a common intermediate representation. Building on previous projects, we adopted the CoT format as this intermediate representation. CoT is an XML-based message format originally designed by MITRE for the U.S. Air Force. Over 100 systems currently support CoT in some form. CoT provides basic data types for exchanging position, imagery, and tasking information, and is also extensible so that new message types can be added without breaking backwards compatibility with existing systems. We are in the process of designing extensions to CoT for sending UGS detection messages and used a preliminary version of these extensions in the cuing system. For reference, the XML schema for these preliminary extensions can be found in appendix A.

3.5 Database Schema Design

The database design used for the cuing system uses two main tables: one for asset information and one for event information. The asset table is used to store information about sensors and other assets that the cuing system can control. The events table stores information about target detections. Each asset or event is represented as an individual row in the appropriate table. The columns in the tables represent a subset of the fields present in the CoT messages that we use for asset and event data.

Geographic positions are stored as latitude and longitude in WGS84 decimal degrees. Heading and azimuth are stored in degrees, with 0° representing true north. Elevation angles are stored in degrees, with 0° being horizontal. Altitude is stored in meters above the WGS84 ellipsoid. If a particular data item is not available for a particular asset or event, then the corresponding field is set as NULL in the database.

3.1.1 Assets Table

The table storing asset data is called **`sensors`**. The definition for this table is shown in figure 1. While the table name and the fields within it use the term “sensor” or “snsr” for historical reasons, this table is used to store data for all types of assets known to the cuing system. This table contains the following columns:

- `snsrID` – a human-readable unique identifier for the asset
- `snsrEnabled` – true if reports from this asset should be used to trigger cuing rules
- `cueEnabled` – true if cuing commands should be sent to this asset
- `snsrCurLat` – current latitude of the asset
- `snsrCurLon` – current longitude of the asset
- `snsrCurHdg` – current heading of the asset

- `snsrCurAlt` – current altitude of the asset
- `snsrLastRptTime` – the time that the last message was received from the sensor
- `isSensor` – true if the asset will generate sensor detection messages
- `isMobile` – true if the asset can be commanded to move to a different location
- `isTriggerCamera` – true if the asset can be commanded to take a picture
- `isPanTilt` – true if the asset has a pan/tilt device that can be commanded to look in different directions

```
CREATE TABLE sensors (
    snsrID text PRIMARY KEY NOT NULL,
    snsrEnabled bool NOT NULL,
    cueEnabled bool NOT NULL,
    snsrCurLat double,
    snsrCurLon double,
    snsrCurHdg real,
    snsrCurAlt real,
    snsrLastRptTime datetime,
    isSensor bool,
    isMobile bool,
    isTriggerCamera bool,
    isPanTilt bool
);
```

Figure 1. Data definition language (DDL) for the sensors table.

The rows in the table are populated based on received asset status messages and other position and attitude messages. All entries in this table represent friendly (blue) entities. The boolean flags, such as `snsrEnabled` and `isMobile`, are provided for use in trigger rules and are not otherwise checked or enforced by the cuing system (i.e., nothing will prevent a malformed rule from generating and sending pan commands for a disabled asset that has no pan/tilt capability.)

3.1.2 Events Table

The table storing event data is called **`events`**. The definition for this table is shown in figure 2. This table contains the following columns:

- `eventID` – a unique identifier for the event
- `eventTime` – the time when the event occurred
- `receivedTime` – the time when the event message was received by the cuing system
- `fusedNum` – the number of raw sensor reports that were used to produce this event
- `type` – the CoT type of the event, for example, “b-d-a” for an acoustic detection
- `how` – the CoT code identifying the method by which the data in the event message was generated, for example, “m-g” indicates a GPS measurement
- `snsrID` – the unique identifier of the asset that generated the event
- `snsrLat` – the latitude of the asset that generated the event
- `snsrLon` – the longitude of the asset that generated the event
- `snsrHdg` – the heading of the asset that generated the event
- `snsrAlt` – the altitude of the asset that generated the event
- `tgtLat` – the latitude of the detected target
- `tgtLon` – the longitude of the detected target
- `tgtAlt` – the altitude of the detected target
- `tgtHdg` – the heading of the detected target
- `tgtVel` – the velocity of the detected target in meters per second
- `lobAz` – the azimuth of the line of bearing (LOB) associated with this event
- `lobEl` – the elevation angle of the LOB associated with this event
- `lobRange` – the range of the LOB associated with this event
- `tagID` – a unique identifier for the target, if available

```
CREATE TABLE events (
    eventID text PRIMARY KEY NOT NULL,
    eventTime datetime NOT NULL,
    receivedTime datetime NOT NULL,
    fusedNum integer,
    type text,
    how text,
    snsrID text,
    snsrLat double,
    snsrLon double,
    snsrHdg real,
    snsrAlt real,
    tgtLat double,
    tgtLon double,
    tgtAlt real,
    tgtHdg real,
    tgtVel real,
    lobAz real,
    lobEl real,
    lobRange real,
    tagID text
);
```

Figure 2. DDL for the events table.

The cuing system inserts or updates rows in this table as detection messages are received from sensors and other assets. All entries in this table represent unknown (yellow) entities. For mobile sensors, the position stored in this table is the position of the sensor at the time of the event. The `snsrID` field serves as a de facto foreign key referencing the `sensors` table, although this constraint is not declared in the SQL or enforced by the database engine since we want the cuing system to be able to process messages from unknown sensors.

3.1.3 Cuing Rules

Cuing rules are represented and stored as SQL triggers within the database schema. Depending on the type of action being performed, rules may consist of INSERT or UPDATE triggers on either the sensors or events tables. The triggers can make use of SELECT statements to check the event that just occurred as well as other events in the database. Triggers send commands to assets by calling user-defined functions (UDFs) written in C#. The UDFs generate and send the appropriate command to the appropriate asset based on parameters provided by the caller. This is a somewhat unorthodox use of database UDFs, as the side effects of the function call are much more important than the return value. In fact, the return value of these UDFs is generally ignored for this application.

Triggers representing cuing rules may need to use temporary tables to store intermediate results or state information. As it is not legal to create tables within a SQLite trigger, the cuing system implementation must ensure that all necessary temporary tables exist before any rows are inserted that may cause triggers to fire.

A detailed discussion of the UDFs available to trigger rules can be found in section 4, and some example trigger rules are shown in section 5.

4. Implementation

The cuing system was implemented as a new application within an existing mission command system framework that we have developed over the years in support of multiple programs here at ARL. This framework provides basic communications and message delivery services, discovery services, asset data distribution and management services, asset control services, and a number of end-user applications. The framework was designed for operation in dynamic, unreliable network environments, with clients using a decentralized discovery mechanism to locate servers. The framework includes support for acquiring data from a number of experimental and operational UGS systems, along with interfaces to multiple pan/tilt imagers and UGVs. The cuing system makes use of these preexisting components to interact with live and simulated assets.

4.1 Supporting Software

This section contains a brief description of the components in our existing mission command framework that are used by the cuing system. A more detailed description of an earlier version of this mission command framework can be found in (Gregory, et al. 2007).

4.1.1 Communication and Data Management

Discovery and communication functionality is provided by the ARLNetwork libraries and agent registry server. The ARLNetwork library, built on top of the IHMC Mockets library (Tortonesi et al., 2006; Suri et al., 2009; Benvegnu et al., 2009; Benvegnu et al., 2010), provides message-oriented point-to-point communications services. Unreliable/reliable and inorder/unordered transport types can be selected on a per-message basis, and messages using different transport types are sent over the same logical connection. The agent registry server, based on the IHMC Group Manager (Suri et al., 2006; Suri et al., 2008; Suri et al., 2010), provides distributed discovery functionality using a yellow pages model. Service consumers broadcast a query for a particular type of service, identified by a simple string, and any producers of that service will respond to the request.

Distribution of asset messages is handled by the Tactical Object Services (TOS) and Sensed Object Services (SOS) servers. These servers receive CoT messages from producers and send them to all connected clients, while also maintaining a local cache so that newly connected clients can receive messages that were previously sent. TOS handles asset position data and fused sensor reports. SOS handles raw sensor reports. Although the type of information sent to the TOS and SOS servers is different, the low-level functionality provided by the two servers is

identical and both services are provided by a common server process. The distinction is for historical reasons, as SOS and TOS were separate applications that used different XML message formats prior to our adoption of CoT.

4.1.2 Asset Mission Command

The cuing system interfaces with two types of assets: assets that produce data relatively infrequently and are capable of limited remote control, such as UGS, and assets that produce live streaming data and can be remotely controlled in real time, such as UGVs and surveillance cameras. We use different sets of software components to interface with these two different classes of asset.

The interface to UGS and similar assets is provided by the SensorDataProcessor. The SensorDataProcessor receives raw sensor messages from a variety of UGS assets, parses the proprietary message formats, and produces CoT messages containing sensor status, detection, and imagery data. Sensor status data is sent to TOS. Detection data is sent to SOS. Imagery data is sent to a media server, which is beyond the scope of this document as it is not used by the cuing system. The SensorDataProcessor also provides support for sending configuration, status request, and control commands to remote UGS assets.

Interaction with devices like UGVs and surveillance cameras is handled through a suite of software originally developed for ARL's small robotics program. The main component of this software suite is the NewRobotAgent, which provides the low-level interface to a number of different hardware devices and sends and receives commands and data over the network using a common binary format. The NewRobotAgent also contains support for controlling pan/tilt cameras. Additional servers collect GPS position information and send CoT position reports to TOS. Some mobile assets also run autonomous navigation software. Control of these assets takes place via point-to-point connections between software running on the asset and the controlling application.

4.1.3 Miscellaneous Components

There are two additional services that provide miscellaneous functionality used by the cuing system. The first is the TripwireTracker, which converts raw LOB reports from sensors to "fused" position reports using simple algorithms. TripwireTracker serves as a placeholder/proof-of-concept application and is not intended or presented as an advanced fusion algorithm. The second service is CueCamServer, which keeps track of pan/tilt camera locations and converts requests to point a camera towards a particular geographic location to lower-level commands to pan the camera to a particular azimuth and elevation.

4.2 Cuing System

The cuing system consists of two applications: AutocueServer and AutocueConfig. Both are written in C# using Microsoft Visual Studio and use SQLite 3 with the System.Data.SQLite

wrappers as a database engine. AutocueServer is the main application that evaluates cuing rules, interfacing with the previously described components to receive data and send cuing commands. AutocueConfig is a graphical user interface (GUI) tool used to configure cuing rules and view sensor data. Both applications use a common library, AutocueDB, to interact with the SQLite database.

4.2.1 AutocueDB Library

AutocueDB is a library that is used by AutocueServer and AutocueConfig to perform all database operations. AutocueDB contains functions for inserting and updating information in the database and for creating, updating, and deleting the database triggers that are used to store cuing rules. The library uses ADO.NET TableAdapter classes generated by the Visual Studio dataset design tools for inserting and updating rows, but uses the SQLite APIs for manipulating triggers as the generated classes do not expose this functionality. As all database interaction logic is contained within AutocueDB, the underlying database engine can be changed by modifying only AutocueDB while leaving the rest of the system as is.

4.2.2 AutocueServer

AutocueServer performs three main functions: ingesting detections and position reports from SOS and TOS; processing the cuing rules; and sending commands to cameras, robots, and other assets. The data ingest functionality is handled by SOS and TOS client classes that receive CoT messages from the servers, extract the relevant fields, and call AutocueDB to insert rows for new detections or update rows for existing detections or reports. The TOS client examines the type field in the CoT message to determine whether the report is for a friendly unit. Friendly unit reports are placed into the **`sensors`** table, while all other reports are placed into the **`events`** table. All reports received from the SOS client are placed into the events table. As a result, the events table contains both raw sensor data and fused detections. These two classes of data can be identified using the contents of the **`fusedNum`** and **`type`** fields.

The actual processing of cuing rules is handled entirely by SQLite. Rules are represented as triggers stored within the SQLite database schema, and are automatically executed by the database engine as messages are received and rows inserted and updated in the database. The trigger rules call UDFs to send commands to assets and perform calculations on geographic coordinates.

Sending commands to assets is less straightforward. The UDFs used to control assets take an asset identifier as a parameter, along with other arguments specific to the command being called. Mapping this identifier to an actual asset within our framework presents a challenge. As previously mentioned, real-time control within our framework is accomplished using point-to-point connections between the controlling application and the controlled asset. Discovery of these assets, as with any other type of service in the framework, occurs in a distributed manner—there is no central facility that can provide an authoritative list of assets. This design avoids

single points of failure, but places a higher burden on individual control applications as they must maintain their own list of assets and update this list as assets enter and leave the network. For pan/tilt cameras, maintenance of this asset list is handled by the CueCamServer, and the pan/tilt control UDFs simply send camera pointing requests to this server. For UGVs, the asset list is maintained by client classes within AutocueServer, and the UGV-control UDFs interact with these classes to send commands directly to assets. The code that maintains the asset lists also updates the `sensors` table to ensure it remains synchronized with the actual list of available assets.

AutocueServer contains a number of additional functions. A background thread updates a row in the events table once a second, which can be used as a trigger for UDFs that implement time-based or periodic behaviors. AutocueServer also contains a command line interface that can be used to run queries on the database for testing and debugging purposes. Finally, AutocueServer provides UDFs that print output to the console and generate Windows message boxes for use when debugging rules.

Detailed reference information for the UDFs provided by the cuing system can be found in appendix B.

4.2.3 AutocueConfig

The AutocueConfig application provides a simple GUI for viewing the current contents of the database and for creating and editing cuing rules and temporary table definitions. AutocueConfig accesses the database files directly and can run either by itself or concurrently with AutocueServer. The SQLite libraries manage concurrent access to the on-disk database file.

Figures 3 and 4 show the sensor and event list views in AutocueConfig. These contain standard .NET DataGridView controls that allow the user to view and edit information about assets and events stored in the database and to clear the contents of the asset and event tables.

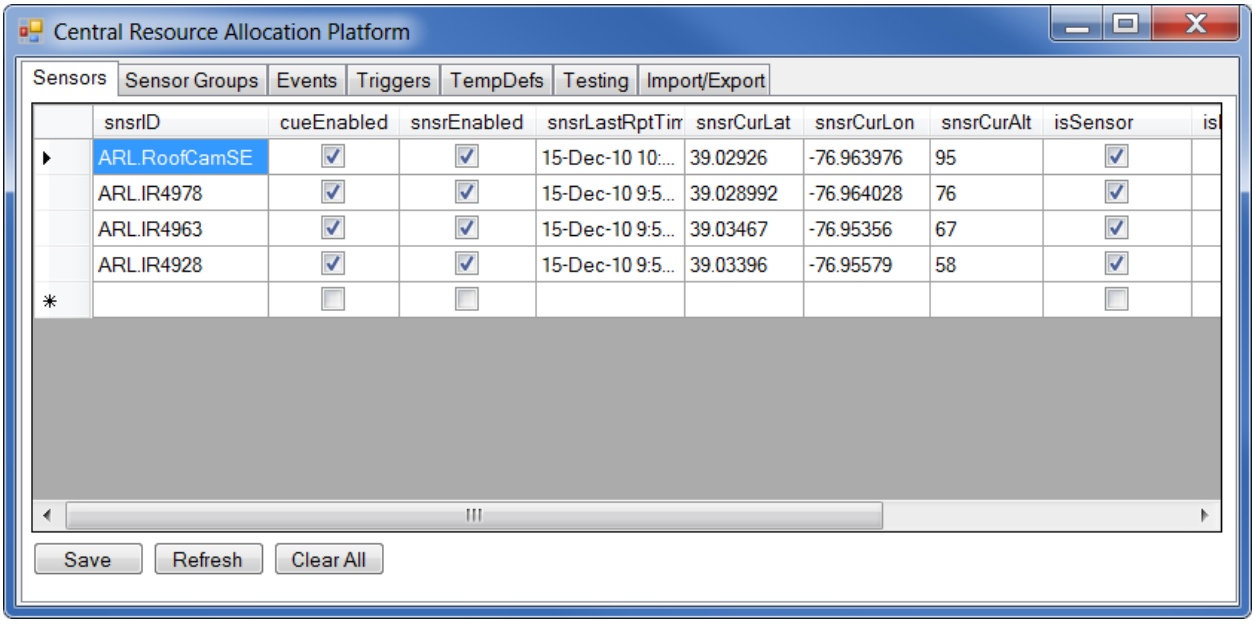

Figure 3. Sensor list viewer in AutocueConfig.

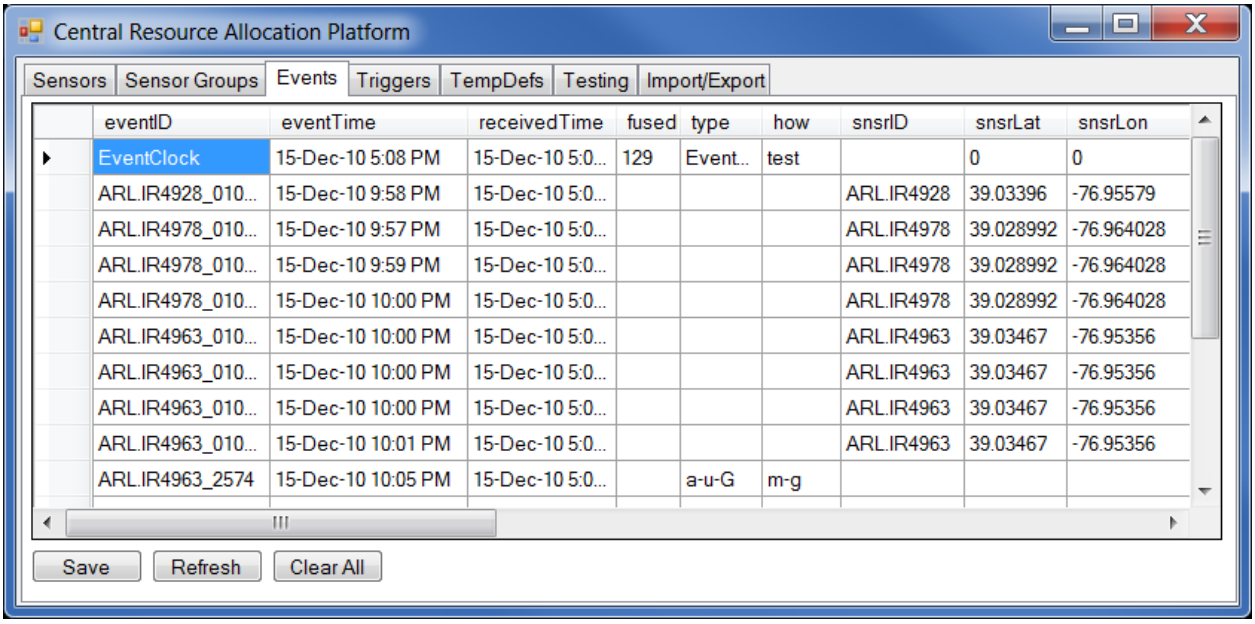

Figure 4. Event list viewer in AutocueConfig.

Figure 5 shows the rules editor in AutocueConfig. This screen allows the user to enter, edit, and delete cuing rules, which are represented as triggers stored in the SQLite database. Essentially this screen presents a GUI frontend for editing SQLite triggers. The user can create and edit triggers for inserts or updates on either the events or sensors tables.

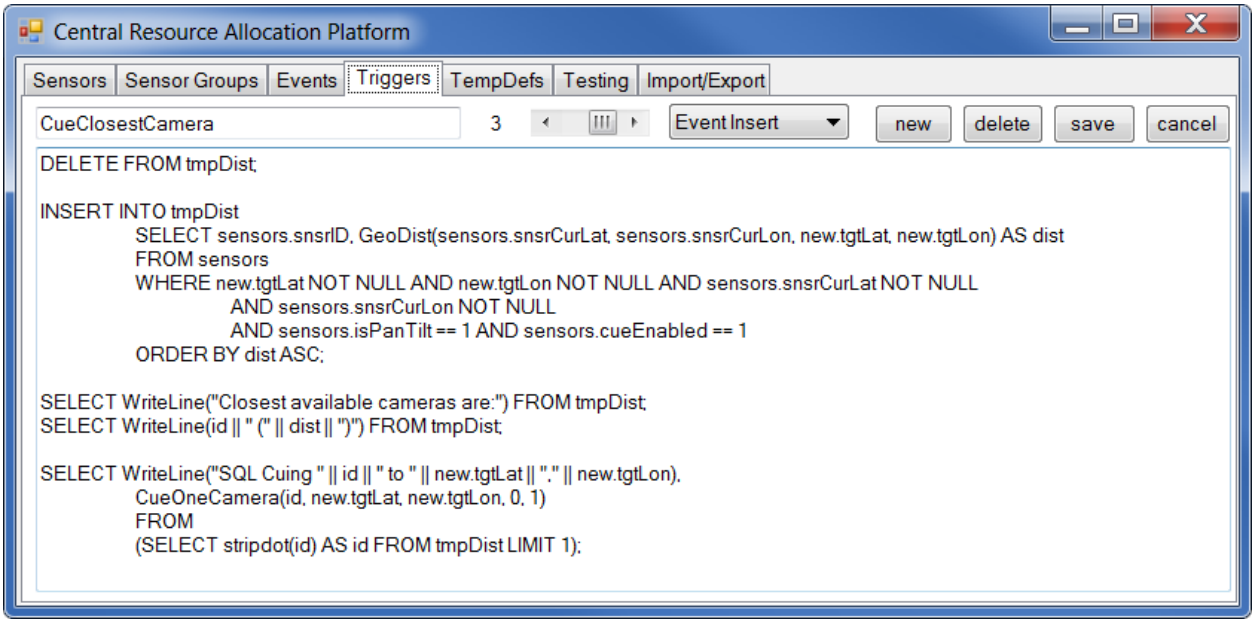

Figure 5. Rules editor in AutocueConfig.

As seen in figure 5, cuing rules are entered directly as SQL. This allows the user to make full use of the SQL language to define complex rules and behaviors, but requires that the user have general knowledge of SQL as well as specific knowledge of the cuing system database design. To make the system useful to users who are not also database developers, it will be necessary to develop or incorporate a graphical rule builder tool.

Figure 6 shows the temporary table definition editor in AutocueConfig. Queries entered in this screen are stored as text in the database, and are read and executed by AutocueServer upon startup. This feature can be used to create or initialize any tables that are used by triggers to store state or intermediate results since it is not legal to create or delete tables within a SQLite trigger.

AutocueConfig contains features for importing and exporting policy sets as XML documents. This allows policies to be backed up, archived, and loaded into other instances of the cuing system. Additionally, AutocueConfig contains an option to manually generate random sensor events for test purposes.

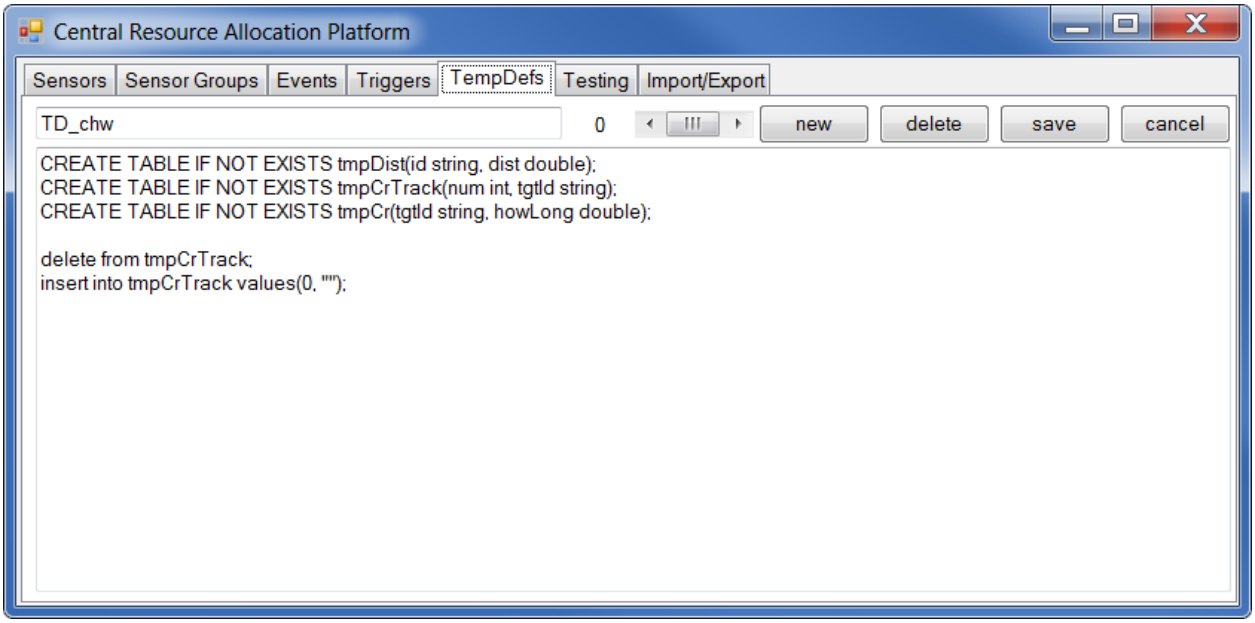

Figure 6. Temporary table definitions editor in AutocueConfig.

4.2.4 Issues Encountered During Development

We encountered two major issues during the cuing system development effort, one related to performance and the other due to query optimizations performed by SQLite. The performance issue was relatively straightforward. Whenever the cuing system receives a CoT message, it generates an insert or update query on the database. The original implementation did not explicitly use transactions when executing these queries, which results in SQLite running each query as a separate transaction. A transaction commit in SQLite is a very expensive operation that involves multiple file creations, deletes, and disk cache flushes to ensure the database file is always in a consistent state. In our experience, SQLite can process individual queries within a transaction very quickly but can only commit a few transactions per second. This caused message traffic to “back up” in network receive buffers when the incoming message rate was too high.

To work around this issue, we modified the cuing system to use a background thread to process incoming messages. These messages are inserted into a queue instead of being sent directly to the database. The background thread runs in a loop, first removing all messages from the queue as a group and then posting this group of messages to the database as a single transaction. Essentially, all messages received during the commit of a previous transaction are grouped and processed as a single transaction. This keeps incoming message latency down to, at most, the amount of time required to commit two transactions.

The issue we encountered with SQLite query optimizations is more fundamental. As previosuly mentioned, the cuing system uses UDFs in an unorthodox manner. A typical UDF in a database performs a calculation on the supplied values and returns a result which is used elsewhere in the

query or returned to the database application. In the cuing system, calling some UDFs will send commands to external systems. These "side effects" of calling the UDF are much more important than the actual return value of the UDF, which is almost always ignored in cuing rules. The SQLite query planner does not account for UDFs having side effects and will often generate optimized queries that result in more UDF calls than one would expect. We encountered this problem when using nested queries in cuing rules, as the query planner will flatten these queries in order to make better use of indexes. This issue cannot be "fixed" without modifiying the core SQLite code. However, it can be circumvented by splitting up the nested query into multiple queries that store intermediate results in a table or by writing queries that the query planner cannot flatten. More details regarding this issue, along with an illustrative example, can be found in appendix C.

5. Examples & Field Test Results

The cuing system was tested as an element of the HFD ATO Capstone Experiment at Aberdeen Proving Ground (APG) in September 2010. This experiment brought together sensors, images, and other assets from multiple vendors. Many of these vendors made their asset data available as CoT messages. In addition, ARL provided cameras, UGVs, and other controllable assets that could be cued by the cuing system. This provided an ideal environment for testing the cuing system with live, real-world data and assets. More information regarding the HFD ATO can be found in (Scanlon and Ludwig, 2010).

We demonstrated several cuing behaviors at the HFD experiment. A description of each behavior, as well as the actual cuing rules used to implement each behavior, follows.

5.1 Cuing Pan/Tilt Imagers from UGS Tripwire Detections

The first behavior demonstrated was to cue ARL-provided pan/tilt imagers (Sony SNC-RZ30N IP surveillance cameras) based on detection events reported from ARL-provided tripwire sensors. When a tripwire sensor detects motion, it sends a message to SensorDataProcessor, which generates a CoT event indicating activity at the location of the sensor. The cuing system finds the closest camera to the location of the event and then cues that camera to look towards this location.

This rule is written as three queries (shown in figure 8) and makes use of a temporary table (defined as shown in figure 7) to store intermediate results. The first query clears the temporary table. The second query computes the distance from each pan/tilt camera to the location of the new event using the GeoDist UDF and stores the results in the temporary table in sorted order with the closest camera first. The third query cues the closest camera to look towards the detection as a side effect of the CueOneCamera UDF. These three queries can be combined, but this can cause problems with the SQLite query planner's optimizations as described earlier. It is

possible to write a combined query that does not cause problems with optimizations, but we did not discover this until after the experiment.

```
CREATE TEMPORARY TABLE IF NOT EXISTS tmpDist(id string,
      dist double);
```

Figure 7. Temporary table definition for tripwire cuing rule.

```
DELETE FROM tmpDist;

INSERT INTO tmpDist
      SELECT sensors.snsrID, GeoDist(sensors.snsrCurLat,
            sensors.snsrCurLon, new.tgtLat, new.tgtLon) AS dist
      FROM sensors
      WHERE new.tgtLat NOT NULL AND new.tgtLon NOT NULL
            AND sensors.snsrCurLat NOT NULL
            AND sensors.snsrCurLon NOT NULL
            AND sensors.isPanTilt == 1 AND sensors.cueEnabled == 1
      ORDER BY dist ASC;

SELECT CueOneCamera(id, new.tgtLat, new.tgtLon, 0, 1)
      FROM
      (SELECT stripdot(id) AS id FROM tmpDist LIMIT 1);
```

Figure 8. Tripwire cuing rule.

5.2 Cuing Pan/Tilt Imagers from Acoustic Detection Events

The second behavior demonstrated was to cue an ARL-provided pan/tilt imager based on gunshot detections from acoustic sensors provided by multiple vendors. When an acoustic sensor detects gunfire, it sends a report that is ultimately converted to a CoT event containing the location of the sensor along with the azimuth and elevation for a LOB indicating the direction to the shooter. These LOB messages are received by the TripwireTracker application, which computes an estimate of the shooter location (based on the averaged intersection points of the LOBs from each sensor) and publishes this estimate as a new CoT event. Some sensor systems additionally compute and publish their own estimated shooter locations. The cuing system receives these fused events and cues cameras to look in the direction of the shooter. At the HFD experiment, TripwireTracker successfully processed CoT-formatted LOB messages from the Unattended Transient Acoustic MASINT Sensor (UTAMS) system provided by the ARL Sensors and Electron Devices Directorate (SEDD), the PDCue system provided by AAI Corporation, and various experimental acoustic sensors provided by the ARL Computational and Information Sciences Directorate (CISD). The cuing system successfully processed estimated shooter locations from UTAMS as well as TripwireTracker.

This rule was written as two relatively straightforward queries (shown in figure 9). The first query cues imager RSN101 based on detections received from the UTAMS system (as a side

effect of the CueOneCamera UDF), which have IDs beginning with “ARL.UTAMS”. The second query cues imager RSN103 based on detections received from TripwireTracker, which have IDs beginning with “ARL.CROSSPOINT”. We also tested a rule similar to the one shown in figure 8 that cued only the closest camera based on detections from either source.

```
SELECT CueOneCamera(id, new.tgtLat, new.tgtLon, 0, 1)
      FROM (SELECT "RSN101" AS id)
      WHERE new.eventID LIKE "ARL.UTAMS%"
      AND new.tgtLat NOT NULL AND new.tgtLon NOT NULL;

SELECT CueOneCamera(id, new.tgtLat, new.tgtLon, 0, 1)
      FROM (SELECT "RSN103" AS id)
      WHERE new.eventID LIKE "ARL.CROSS%"
      AND new.tgtLat NOT NULL AND new.tgtLon NOT NULL;
```

Figure 9. Acoustic sensor cuing rule.

5.3 Cuing Pan/Tilt Imagers from the Compact Radar

The third demonstrated behavior was to cue a pan/tilt imager to track a target detected by the ARL Compact Radar sensor. The Compact Radar detects moving targets and tracks them over time. This ultimately generates a CoT event for each target, which is updated as the target moves. The cuing system receives these events, selects a single target to follow, and commands a pan/tilt imager to follow the selected target. This behavior uses a very simple algorithm to select the target to track. The system will select the first target that it sees and will continue to track that target until it has not been seen for 10 s. The system will then select the next target it sees.

This capability was conceived and implemented in the field and required changes to the AutocueServer code. The original implementation of AutocueServer assumed that each sensor would only send discrete events and, therefore, only supported triggers on database INSERTs. This assumption is valid for gunshot detectors and tripwires, but not for the Compact Radar as it tracks targets over time and updates existing CoT events. To accommodate this, we modified AutocueServer and AutocueConfig to allow triggers to be set on database UPDATEs as well as INSERTs. These modifications were not difficult.

This rule is written as two queries (shown in figure 11) and uses a temporary table (defined as shown in figure 10) to maintain state information (in this case, the ID of the target currently being tracked) between calls to the trigger. The first query filters out events that did not come from the Compact Radar and selects the target to track. The nested query determines whether the target currently being tracked, which is stored in the tmpCrTrack table, has been seen in the last 10 s. If this target has not been seen, the target being tracked will be changed to the target in the event that is currently being processed. The second query cues the camera to the new location of the tracked target if the current event is for that target, ignoring all other events.

```
CREATE TEMPORARY TABLE tmpCrTrack(num int, tgtId string);

INSERT INTO tmpCrTrack VALUES(0, "");
```

Figure 10. Temporary table definition for radar cuing rule.

```
UPDATE tmpCrTrack SET tgtId = new.eventID WHERE
      (SELECT count(*) FROM events
      WHERE events.eventTime > new.eventTime - (10/86400.0)
      AND events.eventID =
            (SELECT tgtId FROM tmpCrTrack WHERE num = 0)) = 0
      AND new.eventID LIKE "ARL.CR4966%"
      AND new.eventID NOT LIKE "ARL.CR4966_0001%"
      AND tmpCrTrack.num = 0;

SELECT CueOneCamera("RSN102", new.tgtLat, new.tgtLon, 0, 1)
      FROM tmpCrTrack WHERE num = 0
      AND tgtId = new.eventID;
```

Figure 11. Radar cuing rule.

A more complex rule could use an alternate method to pick the target. For example, a rule could track the fastest moving target or the closest target to a location.

6. Conclusions

We have shown that SQL is a viable policy language for unattended asset cuing applications through a successful practical demonstration at the HFD capstone experiment. In addition, we demonstrated the flexibility that SQL provides in these applications by quickly defining and adding new types of cuing capabilities during the experiment. We also demonstrated the viability and utility of multivendor cross-cuing by successfully cuing assets based on data from both ARL-developed and vendor-provided sensors.

7. Future Work

The current implementation of the cuing system is intended as a proof-of-concept demonstration. As a result, it has several architectural issues and limitations. We are currently working on enhancing the system to address some of these issues.

7.1 Distributed Operation

The current cuing system uses a centralized architecture where a single server receives all asset reports, processes all cuing rules, and sends out commands to all assets. This creates a single

point of failure. Additionally, we have found that centralized architectures do not perform well in mobile ad-hoc networks commonly used in tactical environments (Suri, Benvegnu, et al. 2009). To address this issue, we plan to redesign the cuing system to support distributed operation.

One possible distributed implementation would run a separate cuing server on each asset and use a broadcast or multicast mechanism to send asset detection, position, and attitude messages to the cuing servers. Each instance of the cuing server would have its own copy of the cuing rules and be responsible for cuing only its local asset. When a multicast message is received, the cuing server on each asset will use the database engine to evaluate the cuing rules, but will ignore any commands generated for non-local assets. Assets will essentially be cuing themselves. If one asset drops off the network, other assets will continue to function. With this design, each instance of the cuing server will maintain its own state information (sensors and events tables). This information may not be correct or consistent with other server instances running on other assets. This will require a different approach to writing rules when compared to the current centralized system, especially with rules that make decisions based on the state of multiple assets.

7.2 Usability Improvements

The current cuing system requires that rules be entered as SQL statements. Therefore, a working knowledge of SQL is required in order to construct cuing rules. Many potential end-users of the cuing system will not have this knowledge. An alternate mechanism for defining cuing rules, such as a graphical rule builder, would help improve the usability of the system. However, creating an interface that is simple enough for an untrained user to understand but powerful enough to create useful and complex cuing rules will be difficult. Various existing query-by-example and programming-by-flowchart tools attempt to perform these types of tasks, but in practice are still not usable without experience or training. A possible compromise may be to provide rule templates for the most common use cases that could be set up and configured using a graphical interface, while still providing advanced users the ability to enter custom rules as SQL.

7.3 Data Fusion in SQL

Asset cuing is only one component of a full-scale sensor exploitation system. It may be possible to use a SQL-based approach for other sensor exploitation tasks, such as data fusion. As an initial exploration in this area, we have performed some LOB intersection calculations on test data by using SQL queries that call cuing system UDFs to perform coordinate math. Further investigation is required to evaluate the performance of this approach compared to traditional approaches and to determine whether it is feasible to perform other types of sensor fusion computations in a similar manner.

7.4 Other Database Systems

SQLite is intended as an embedded database engine and does not provide support for stored procedures or a stored procedure language. Stored procedure languages (such as Microsoft's T-

SQL) provide looping, branching, and other control structures that may allow for the development of more complex cuing rules than can be represented using the simple trigger syntax supported by SQLite. This would be especially useful when trying to do data fusion within the database. This capability can be provided by extending the cuing system to use a full-featured database engine like Firebird or Microsoft SQL Server. However, doing so would also increase the footprint and decrease the portability of the system.

INTENTIONALLY LEFT BLANK.

Appendix A. XML Schema for CoT Sensor Detection Extension

The CoT message format, as defined by MITRE, does not provide fields for some of the data items generated by the UGS systems we use. To accommodate this data, we defined experimental extensions to CoT for UGS detections. These extensions were rapidly designed to support specific experiments and demonstrations and do not accommodate all types of UGS messages. We are currently redesigning these extensions to be more universal in nature. The final version of these extensions will be published in a future report.

The XML schema for a preliminary version of these CoT extensions, used at the Sensor and Information Fusion for Improved Hostile Fire Situational Awareness ATO Capstone Experiment and Empire Challenge 2010 demonstration, is included here for reference.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="detection">
    <xs:complexType>
      <!-- ID for the sensor. Must be unique -->
      <xs:attribute name="sensorID" type="xs:string" use="required"/>

      <!-- Azimuth in degrees. 0 = true north (NOT magnetic north.)
          90 = due east. -->
      <xs:attribute name="azimuth" type ="xs:decimal" use="optional"/>

      <!-- Azimuth error +/- in degrees-->
      <xs:attribute name="azimuthErr" type="xs:decimal" use="optional" />

      <!-- Elevation in degrees. 0 = horizontal. 90 = straight up. -->
      <xs:attribute name="elevation" type="xs:decimal" use="optional" />

      <!-- Elevation error +/- in degrees -->
      <xs:attribute name="elevationErr" type="xs:decimal" use="optional" />

      <!-- Range TO TARGET in meters, if available. -->
      <xs:attribute name="range" type ="xs:decimal" use="optional" />

      <!-- Range error +/- in meters -->
      <xs:attribute name="rangeErr" type="xs:decimal" use="optional" />

      <!-- Maximum effective detection range of the sensor. Used to put
                an upper bound for length on the LOBs when doing
           intersection calculations (i.e. how far out will
           we look for intersection points) -->
      <xs:attribute name="rangeMax" type="xs:decimal" use="optional" />

      <!-- Timestamp of the detection -->
      <xs:attribute name="detectiontime" type="xs:dateTime" use="optional" />
```

```
      <!-- Detection identifier from the sensor if available.
                  (used if sensor generates multiple messages for the same
           event -->
      <xs:attribute name="DetectionID" type="xs:string" use="optional" />

      <!-- RFID tag from the target, if available (or equivalent value
           for other sensors) -->
      <xs:attribute name="TAGID" type="xs:string" use="optional" />

      <!-- Quality factor. Higher is better -->
      <xs:attribute name="quality" type="xs:decimal" use="optional" />

    </xs:complexType>
  </xs:element>
</xs:schema>
```

Appendix B. User-defined Function Reference

Unless otherwise specified, all latitude and longitude values are in decimal degrees, all distance values are in meters, all bearing values are in degrees with 0 representing true north and 90 representing true east, and all altitude values are in meters above the WGS84 ellipsoid.

Coordinate Math

double GeoDist(double lat1, double lon1, double lat2, double lon2)

This function returns the distance between two points on the Earth's surface.

- lat1: the latitude of the origin point
- lon1: the longitude of the origin point
- lat2: the latitude of the destination point
- lon2: the longitude of the destination point

double GeoBearing(double lat1, double lon1, double lat2, double lon2)

This function returns the bearing from one point to another point on the Earth's surface.

- lat1: the latitude of the origin point

- lon1: the longitude of the origin point

- lat2: the latitude of the destination point

- lon2: the longitude of the destination point

string LineIntersect(double lat1, double lon1, double az1, double lat2, double lon2, double az2)

This function returns the intersection point of two LOBs. Each line is specified as an origin point and a compass direction. The return value is a string containing the latitude of the intersection point, a space, and the longitude of the intersection point. If the lines do not intersect, the return value is the empty string. The latitude and longitude can be extracted from this string using IntersectGetLat() and IntersectGetLon().

- lat1: the latitude of the origin point of the first LOB

- lon1: the longitude of the origin point of the first LOB

- az1: the bearing of the first LOB

- lat2: the latitude of the origin point of the second LOB

- lon2: the longitude of the origin point of the second LOB

- az2: the bearing of the second LOB

double IntersectGetLat(string str)

This function returns the latitude component of a result string from LineIntersect.

- str: a result string from LineIntersect()

double IntersectGetLon(string str)

This function returns the longitude component of a result string from LineIntersect.

- str: a result string from LineIntersect()

Robot Cuing

bool MoveRobot(string robotID, double lat1, double lon1, [double lat2, double lon2, …])

This function sends a waypoint list to a robot. If the robot supports autonomous navigation, it will attempt to navigate to each point in the waypoint list. This function returns true if the waypoint list was successfully sent to the robot and false otherwise. This function takes a variable number of arguments. The robot ID and first waypoint must always be specified.

- robotID: the ID of the robot to move. This is generally the platform name of the robot in the ARL C2 framework.

- lat1: the latitude of the first waypoint.

- lon1: the longitude of the first waypoint.

- lat2: optional latitude of the second waypoint.

- lon2: optional longitude of the second waypoint.

- …: optional additional waypoints.

Camera Cuing

bool CueOneCamera(string cameraID, double lat, double lon, float alt, byte ignoreCueState)

This function commands a camera to look at a point on or above the Earth's surface. This function returns true if the command was successfully sent to the camera and false otherwise.

- cameraID: the ID of the camera to move.

- lat: the latitude of the target point.

- lon: the longitude of the target point.

- alt: the altitude of the target point.

- ignoreCueState: if true, the camera will be moved regardless of whether camera cuing for this camera has been disabled in CueCamServer.

bool PanTiltOneCamera(string cameraID, double az, double el, bool isWorld)

This function commands a camera to look in a given direction. This function returns true if the command was successfully sent to the camera and false otherwise.

- cameraID: the ID of the camera to move.

- az: the commanded pan angle (azimuth.)

- el: the commanded tilt angle (elevation.)

- isWorld: if this is true, the pan and tilt angles are handled as global coordinates (zero azimuth is true north, and zero elevation is horizontal, with positive elevation angles looking up.) If false, the pan and tilt angles are sent directly to the pantilt device as local coordinates.

Video Capture

bool GrabVideo(string platform)

This function commands all video servers on a platform to capture an image and send it to the media server.

- platform: the platform to use.

bool GrabVideo(string platform, string server)

This function commands a single video server to capture an image and send it to the media server.

- platform: the platform that is hosting the video server.

- server: the service name of the video server that will grab the image.

bool GrabVideoDelay(int msecDelay, string platform)

This function will command all video servers on a platform to capture an image after a specified delay has passed and send the image to the media server. This can be used to wait for a pan/tilt command to complete before capturing an image.

- msecDelay: the amount of time to wait before grabbing the image. The function will return immediately, regardless of the value of the setting, and the command will be sent in the background.

- platform: the platform to use.

bool GrabVideoDelay(int msecDelay, string platform, string server)

This function commands a single video server to capture an image after a specified delay has passed and send the image to the media server for display to end users.

- msecDelay: the amount of time to wait before grabbing the image. The function will return immediately regardless of the value of the setting. The command will be sent in the background.

- platform: the platform that is hosting the video server.

- server: the service name of the video server that will grab the image.

Miscellaneous

string StripDot(string arg)

This function will look for a dot in the supplied string and will return the portion of the string that follows the dot. This is mainly intended to convert CoT names, such as "ARL.robot1", to their corresponding ARL asset name, such as "robot1".

- arg: the string to process.

bool CreateTosIcon(string uid, string type, string how, double lat, double lon, double alt, double lifetimeSecs)

This function will create a CoT event in TOS with the specified parameters. This function will return true if the event was successfully created and false otherwise.

- uid: the CoT UID for the new event.

- type: the CoT type string for the new event (see the CoT type tree.)

- how: the CoT how string for the new event (see the CoT type tree.)

- lat: the latitude for the new event.

- lon: the longitude for the new event.

- alt: the altitude for the new event.

- lifetimeSecs: the lifetime of the new event, in seconds. The CoT stale time for the new event will be set to the current time plus this.

Debugging

string WriteLine(string arg)

This function prints a message to the console. The message will be printed only if debug messages have been turned on in the autocue server. This function returns the argument that was passed in.

- arg: the message to print to the console.

string ForceWriteLine(string arg)

This function prints a message to the console. The message will be printed regarldess of whether debug messages have been turned on. This function returns the argument that was passed in.

- arg: the message to print to the console.

string MsgBox(string arg)

This function pops up a Windows message box containing a specified message. The message box will run in the thread that called the UDF, which will likely cause the cuing server to stop processing further data until the user closes the message box. This function returns the argument that was passed in.

- arg: the message to put in the message box.

string MsgBoxThreaded(string arg)

This function pops up a Windows message box containing a specified message. The message box will be created in a new thread. The cuing server will continue to process data while the message box is on the screen. Repeated calls to this function will pop up additional message boxes regardless of whether the original message box has been closed. This function returns the argument that was passed in.

- arg: the message to put in the message box.

INTENTIONALLY LEFT BLANK.

Appendix C. SQLite Order by Optimization Examples

Typical UDFs in database applications perform a calculation on the parameters supplied to the UDF and return a value. The cuing system makes use of UDFs that generate and send commands to external systems as side effects. The query optimizer in SQLite does not account for UDFs having these types of side effects, and will often generate queries that call a UDF more times than one would expect. Within the cuing system, this optimization can cause unexpected results as illustrated below.

These examples are all run using the debugging interface in AutocueServer and make use of the WriteLine() UDF available within AutocueServer, which prints a message to the console along with a timestamp. WriteLine also returns the value passed to it.

First, we define a test table, foo, with two columns a and b. a contains integers, b contains text strings.

```
sql> CREATE TABLE foo (a INTEGER, b TEXT);
```

Now we insert some data into the test table and then verify that the data has been inserted. For illustrative purposes, we will make sure the values are not already in sorted order.

```
sql> INSERT INTO foo VALUES(5, "five");
sql> INSERT INTO foo VALUES(3, "three");
sql> INSERT INTO foo VALUES(1, "one");
sql> INSERT INTO foo VALUES(8, "four plus four");

sql> SELECT * FROM foo;
a, b
5, five
3, three
1, one
8, four plus four
```

SQLite allows you to use most standard SQL commands, so we can get values in sorted order using an ORDER BY clause and limit the number of returned values with a LIMIT clause. (In the cuing system, a similar query can be used to find the closest camera to a location of interest.)

```
sql> SELECT * FROM foo ORDER BY a;
a, b
1, one
3, three
5, five
8, four plus four

sql> SELECT * FROM foo ORDER BY a LIMIT 1;
a, b
1, one
```

If we want to take the result of this query and pass it to a UDF, the obvious way to do so is to incorporate it as a subquery within an outer query that calls the UDF. However, if we use this approach the optimizer will alter the query, resulting in unexpected behavior.

```
sql> SELECT a, WriteLine("UDF called for " || a) FROM
     (SELECT * FROM foo ORDER BY a LIMIT 1);
[11:30:10] UDF called for 5
[11:30:10] UDF called for 3
[11:30:10] UDF called for 1
[11:30:10] UDF called for 8
a, WriteLine("UDF called for " || a)
1, UDF called for 1
```

Notice that while the return value of the outer query is correct, the UDF is actually called four times, once for each row in the database in unsorted order. The ORDER BY and LIMIT clauses are not being processed with the nested query. What is happening here is that the SQLite query planner is flattening the query in order to avoid using a temporary table, as temporary tables do not have indexes and can exhibit poor performance. (SQLite n.d.) Basically, it is executing the query as

```
SELECT a, WriteLine("UDF called for " || a) FROM foo ORDER BY a
LIMIT 1;
```

(This can be verified by running EXPLAIN on the modified query and then running EXPLAIN on the original query. The sequence of operations in the query plans is the same.) The altered query returns the same result as the original query. However, in the cuing system, we are not interested in the result of the query—we are interested in the actions that are taken when the

UDF is called. If the UDF is called too many times, we do not get the expected results. If we want to find the closest camera to a location and then send a command to only that camera, we will actually end up sending the command to all the cameras.

A query must meet a certain set of conditions in order to be flattened by the SQLite query planner. One of those conditions is that the inner and outer queries cannot both have ORDER BY clauses. If we add an ORDER BY clause to the outer query, then the query is not flattened and the UDF is called the expected number of times.

```
sql> SELECT a, WriteLine("UDF called for " || a) FROM
     (SELECT * FROM foo ORDER BY a LIMIT 1)
     ORDER BY a;
[11:30:38] UDF called for 1
a, WriteLine("UDF called for " || a)
1, UDF called for 1
```

In addition to causing unintended behavior in applications that make use of UDF side effects, this optimization can cause performance issues in applications that call computationally intensive UDFs. In this case, when constructing a query, it will be necessary to evaluate the tradeoff between potential performance gains from the use of indexes on the flattened query against the performance loss from unnecessary calls to the computationally intensive UDF.

Bibliography

Barker, S.; Rosthenal, A. Flexible Security Policies in SQL. *Proceedings of the Fifteenth Annual Working Conference on Databse and Application Security*, Kluwer, 2002, 167–180.

Benvegnu, E.; Suri, N.; Hanna, J.; Combs, V.; Winkler, R.; Kovach, J. Improving Timeliness and Reliability of Data Delivery in Tactical Wireless Environments with Mockets Communication Library. *IEEE Military Communications Conference (MILCOM), IEEE*, 2009.

Benvegnu, E.; Suri, N.; Tortonesi, M.; Esterrich, T. III. Seamless Network Migration using the Mockets Communications Middleware. *IEEE Military Communications Conference (MILCOM)*, IEEE, 2010. 1604–1609.

Bunch, L.; Bradshaw, J. M.; Young, C. O. Policy-governed Information Exchange in a U.S. Army Operational Scenario. IEEE Workshop on Policy-governed Information Exchange in a U.S. Army Operational Scenario. IEEE, 2008, 243–244.

Calo, S.; Lobo, J. A Basis for Comparing Characteristics of Policy Systems. IEEE International Workshop on Policies for Distriubted Systems and Networks, 2006, 183–194.

Cook, W. Policy-based Authorization. Austin: University of Texas, 2009.

Didriksen, T. Rule Based Database Access Control - A Practical Approach. ACM Workshop on Role-based Access Control. ACM, 1997, 143–151.

Gencay, B.; Kuchlin, W.; Schafer, T. SANchk: An SQL-Based Validation System for SAN Confiuration. *IFIP/IEEE International Symposium on Integrated Network Management*, 2007, 333–342.

Gregory, T.; Kovach, J.; Winkler, R.; Winslow, C. *UGS, UGV, and MAV in the 2007 C4ISR OTM Experiment*; ARL-TR-4419; U.S. Army Research Laboratory: Adelphi, MD, 2008.

Johnson, M.; Bradshaw, J. M.; Jung, H.; Suri, N.; Carvalho, M. Policy Management Across Multiple Platforms and Application Domains. *IEEE Workshop on Policies for Distributed Systems and Networks*. *IEEE* 2008, 199–202.

Scanlon, M.; Ludwig, W. Sensor and Information Fusion for Improved Hostile Fire Situational Awareness. Unattended Ground, Sea, and Air Sensor Technologies and Applications XII. Orlando: SPIE, 2010.

SQLite. The SQLite Query Planner. SQLite. n.d. http://www.sqlite.org/optoverview.html.

Suri, N.; Benvegnu, E.; Toronesi, M.; Stefanelli, C.; Kovach, J.; Hanna, J. Communications Middleware for Tactical Environments: Observations, Experiences, and Lessons Learned. *IEEE Communications Magazine* **October 2009**, 56–63.

Suri, N., et al. An Adaptive and Efficient Peer-to-Peer Service-oriented Architecture for MANET Environments with Agile Computing. *IEEE Workshop on Autonomic Comptuing and Network Management. IEEE* **2008**, 364–371.

Suri, N., et al. Peer-to-peer Communications for Tactical Environments: Observations, Requirements, and Experiences. *IEEE Communications Magazine,* **October 2010**: 60–69.

Suri, N.; Rebeschini, M.; Breedy, M.; Carvalho, M.; Arguedas, M. Resource and Service Discovery in Wireless AD-HOC Networks with Agile Computing. *IEEE Military Communications Conference (MILCOM). IEEE*, 2006, 1–7.

Tortonesi, M.; Stefanelli, C.; Suri, N.; Arguedas, M.; Breedy, M. Mockets: A Novel Message-oriented Communications Middleware for the Wireless Internet. *International Conference on Wireless Information Networks and Systems (WINSYS)*, 2006.

Uszok, A., et al. KAoS Policy and Domain Services: Toward a Description-logic Approach to Policy Representation, Deconfliction, and Enforcement. *IEEE 4th International Workshop on Policies for Distributed Systems and Networks.* IEEE, 2003, 93–96.

Uszok, A., et al. New Developments in Ontology-based Policy Management: Increasing the Practicality and Comprehensiveness of KAoS. *IEEE Workshop on Policies for Distributed Systems and Networks.* IEEE, 2008, 145–152.

W3C. *OWL 2 Web Ontology Language* . October 27, 2009. http://www.w3.org/TR/owl2-overview/.

W3C. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML.* May 21, 2004. http://www.w3.org/Submission/SWRL/.

Weth, C.; Bohm, K.; Burghardt, T.; Hutter, C.; Yue, Jing Zhi. Indirect Reciprocity in Policy-based Helping Experiments. *IEEE European Conference on Web Services.* 2009, 171–180.

List of Symbols, Abbreviations, and Acronyms

ARL	U.S. Army Research Laboratory
APG	Aberdeen Proving Ground
ATO	Army Technology Objective
CISD	Computational and Information Sciences Directorate
COT	cursor on target
DDL	data definition language
GUI	graphical user interface
HFD	Hostile Fire Defeat
LOB	line of bearing
OWL	Web Ontolugy Loanguage
SEDD	Sensors and Electron Devices Directorate
SOS	Sensed Object Services
SQL	Structured Query Language
SWRL	Semantic Web Rule Language
TOS	Tactical Object Services
UAV	unmanned aerial vehicle
UDF	user-defined function
UGS	unattended ground sensor
UGV	unmanned ground vehicle
UTAMS	Unattended Transient Acoustic MASINT Sensor
XML	Extensible Markup Language

NO. OF COPIES	ORGANIZATION
1 ELECT	ADMNSTR DEFNS TECHL INFO CTR ATTN DTIC OCP 8725 JOHN J KINGMAN RD STE 0944 FT BELVOIR VA 22060-6218
1 CD	OFC OF THE SECY OF DEFNS ATTN ODDRE (R&AT) THE PENTAGON WASHINGTON DC 20301-3080
1	US ARMY RSRCH DEV AND ENGRG CMND ARMAMENT RSRCH DEV & ENGRG CTR ARMAMENT ENGRG & TECHNLGY CTR ATTN AMSRD AAR AEF T J MATTS BLDG 305 ABERDEEN PROVING GROUND MD 21005-5001
1	PM TIMS, PROFILER (MMS-P) AN/TMQ-52 ATTN B GRIFFIES BUILDING 563 FT MONMOUTH NJ 07703
1	US ARMY INFO SYS ENGRG CMND ATTN AMSEL IE TD A RIVERA FT HUACHUCA AZ 85613-5300
1	COMMANDER US ARMY RDECOM ATTN AMSRD AMR W C MCCORKLE 5400 FOWLER RD REDSTONE ARSENAL AL 35898-5000
1	US GOVERNMENT PRINT OFF DEPOSITORY RECEIVING SECTION ATTN MAIL STOP IDAD J TATE 732 NORTH CAPITOL ST NW WASHINGTON DC 20402
1	FLORIDA INSTIT FOR HUMAN & MACHINE COGNITION ATTN N SURI 40 S ALCANIZ STREET PENSACOLA FL 32502

NO. OF COPIES	ORGANIZATION
1	NAVAL AIR WARFARE CTR AIRCRAFT DIVISION ATTN A PONTZER 17673 WEBSTER FIELD RD ST INIGOES MD 20684
1	PROGRAM MANAGER ACOUSTIC SYSTEMS AAI CORPORATION ATTN T WITTE 124 INDUSTRY LN HUNT VALLEY MD 21030
1	THE MITRE CORPORATION ATTN M/S E030 P G GONSALVES 202 BURLINGTON RD BEDFORD MA 20879
1	VIRGINIA CONTRACTING ACTIVITY ATTN C CROSS 1030 SOUTH HIGHWAY A1A, BLDG 989 PATRICK AFB FL 32925
1	DIRECTOR US ARMY RSRCH LAB ATTN RDRL ROE V W D BACH PO BOX 12211 RESEARCH TRIANGLE PARK NC 27709
23	US ARMY RSRCH LAB ATTN IMNE ALC HRR MAIL & RECORDS MGMT ATTN RDRL CII A D BARAN ATTN RDRL CII A S H YOUNG ATTN RDRL CII B C WINSLOW ATTN RDRL CII B J KOVACH (4 COPIES) ATTN RDRL CII B L TOKARCIK ATTN RDRL CII B R WINKLER (4 COPIES) ATTN RDRL CII B W GOLLSNEIDER ATTN RDRL CIO LL TECHL LIB ATTN RDRL CIO MT TECHL PUB ATTN RDRL SEL B GOLLSNEIDER ATTN RDRL SES A G STOLOVY ATTN RDRL SES A J HOUSER ATTN RDRL SES A L KAPLAN ATTN RDRL SES A N SROUR

NO. OF
COPIES ORGANIZATION

ATTN RDRL SES P M SCANLON
ATTN RDRL SES S D WARD
ADELPHI MD 20783-1197

TOTAL: 36 (34 HCS, 1 CD, 1 ELECT)