DOCUMENTNUMBER

MITRE TECHNICAL REPORT

The Developer's Guide to Cursor on Target

August 2005

Mike Butler

MITRE
Center for Air Force Command & Control
Bedford, Massachusetts

Ancient History

Cursor on Target (CoT) was conceived in the afternoon on June 18, 2002. I was working on a problem that Jim Hill had handed me four days earlier on a scrap of notebook paper. I originally called it STP, short for "Space-Time Pieces." That name reflected what was common among all SA and C2 problems: they're all "what," "where," "when" problems. Rich Byrne, true to character, took a risk to fund the idea. On June 28th, he gave me $20K and 30 days to come up with a prototype; the project was to be called "Meta-Data Integration." What followed was a sleepless month of work, a second round of funding, and a very convincing demo.

Almost immediately, MITRE decided the effort needed a better name, and a speech from General Jumper provided the inspiration. General Jumper said "the sum of all wisdom is a cursor over the target," and MITRE decided "Cursor on Target" was a palatable sound bite. Though CoT was never about cursors and only sometimes about targets, it's still a good name.

Introduction

This is a developer's introduction to Cursor-on-Target, or, preferably, CoT. It describes CoT's objective, implementation, limitations, and future direction. If you are a developer, this document will help you understand what CoT is, why it works, and what's right and wrong with it. It does not discuss implementation details; you'll be better off pulling working samples from our website, http://cot.mitre.org, for that. You can contact the CoT team at cot@mitre.org for with questions.

Document Structure

To save you some time, when you see this mark:

----cut----

you can skip to the next section unless you're particularly interested in the topic at hand. I generally go into more detail on a topic than you need, but read it if you want.

----cut---

Non-developers will find useful information among the growing list of general publications referenced on the CoT website. These include case studies, briefings, etc. Those documents are aimed at a broad audience and aren't encumbered with lots of technical detail.

What is Cursor-on-Target?

In a nutshell, CoT is just a simplified messaging format that's rapidly gaining acceptance throughout the military as an "enterprise integration" solution. Today, there are more than a hundred systems that speak CoT either natively or through a third-party adaptor (called a plug-in). CoT's catching on quickly because it's simple to understand, simple to implement, simple to extend, and completely open.

CoT is simple, very simple. CoT's base class has only 12 required attributes and its complete description fits on a single printed page. Typically, a good developer can implement a basic CoT capability in about 12 hours of effort (less with an adequate supply of Mountain Dew).

----cut---

CoT's simplicity is in marked contrast to all the other military messaging standards that I know of. The Link-16 spec, MS-6016C, for instance, is seven thousand (7,000) pages long. It's so complex that in 20 years, no one has ever fully implemented all of Link-16. VMF is just as bad, maybe worse since there are six VMF message families and no two are interoperable. USMTF appears simpler because it's not binary, but it's a bear to implement because there's so much variability; in fact, it changes so fast that it's published on a 600MB CD every other year.

CoT is also very stable. Although it's in its infancy, CoT's base class hasn't changed at all since its release more than two years ago. In fact, we never expect the base class to change. Sure, we'll extend it by deriving more classes, but that won't change what's already out there.

At present, CoT is only implemented as XML, but there is nothing in CoT that marries it to XML. In fact, as I'll describe below, XML isn't well suited for implementing things like CoT. However, XML is the best thing we have...so far.

The first few sections of this paper begin somewhat aggressively by challenging a couple of tenets widely accepted within the DoD. The intent is not to be confrontational, but rather to set the stage for understanding what makes CoT different.

The "Community of Interest" Myth

Within the DoD today, it's fashionable to talk about corralling users into convenient "communities of interest." Such communities, it's commonly believed, will develop and adopt their own information exchange formats. This makes sense as long as the community boundaries are static and well delineated...but in this case they're not. The community approach does absolutely nothing to solve the inter-community data flow problem. Ironically, it's the inter-community problem that's really killing us.

----cut---

In the Air Force C2 Summit of April of 2002, General Jumper observed that the enterprise integration problem was due largely to the isolation of users into disjoint groups with dissimilar languages. Jumper referred to the existence of "tribes with their own tribal hieroglyphics." If you wanted to understand a tribe's hieroglyphics, you needed a liaison from that tribe.

To answer this, the "community of interest" idea was born. And this is different how? Well, we domesticated "tribes" into "communities," but the isolation remains. Some tribal lines will have to be redrawn, but "tribe" and "community" both reflect the same concept, a division of civilization into disjointed groups.

Fashionable though it may be, and despite their use of XML, "communities of interest" aren't going to help solve the integration problem. To do that, we need to break down the community lines. CoT attempts to do just that.

The XML Mystique

Another widely held belief within the DoD is that XML has mystical powers. As it's briefed, one gets the impression that merely invoking the name of the XML deity is sufficient to excise interoperability demons. XML's not a panacea, and CoT uses (and abuses) it because there's nothing better...yet.

----cut---

In truth, XML does give developers a tremendous leg-up on a convenient, simple, and common way to represent machine readable information. And XML's acceptance has ushered in a cadre of tools to generate, analyze, edit, parse, and view information. These are all absolutely wonderful perks, a tremendous windfall.

[Note, though, if you substitute the letters "ASCII" for "XML" in the previous paragraph, there's an exact a historic parallel. What XML has done for today's programmers is akin to what ASCII did for their fathers decades ago. XML, like ASCII, offered a huge leap forward, but it's not the total solution to the interoperability problem.]

XML is good, but it has issues. In fact, for some applications, XML is quite a poor choice. For example, XML is a horrid technology for implementing object hierarchies: there's no information protection, no multiple inheritance, no polymorphism, no virtual base classes, no methods at all. If one really needs to pass around legitimate OOP objects, XML doesn't cut it. (Sure, folks are working on such things, but they're not here yet.)

And then there's the law of unintended consequence: A decade ago, there were only a couple dozen message formats in use in the DoD (TADIL-J, USMTF, VMF, OTH-G, ESD, etc). But XML has made implementing a data model so trivial that there are now literally tens of thousands of incompatible XML schemas in use within the DoD today. (Visit the DISA XML registry for a very sobering look at the problem.)

Finding the Perfect Word

An oddly crippling aspect of XML is the flexibility of choosing attribute and entity names. CoT errs on the side of pragmatism here. We try to choose good attribute and entity names, but we often miss the mark. But, the parsers don't really care about the names, and we try to adopt their attitude. We certainly don't want to have a hundred systems change all their code because we think *radius* would be a more descriptive attribute name than *ce*.

----cut---

"Back in the day" when variable names were single characters or, worse, hex addresses, there was little debate about the "right" name for a particular variable. It was what it was and that was it.

But now that data modelers can use arbitrary strings for variable (attributes and entity) names, many modelers get hung up finding the "perfect" name. Committees charged with coming up with data models often get mired down in heated debates over names of things and not what they mean. Such arguments, while occasionally interesting, are rarely very profitable. In truth, it matters very little what name a tag or attribute has, provided all users understand what it contains. (Of course, "good" tag names can make a coder's life much simpler by providing useful mnemonics, but "perfect" isn't necessary.)

Many data models charge down the road to "perfect" and completely miss the turnoff for "pragmatic." For better or worse, CoT took the turnoff, though some would argue we took it too early.

CoT tries to use "good" attribute and entity names, but we openly admit they're not "perfect." Furthermore, in the interest of pragmatism, CoT will not change an attribute's name once it's in common use. This has resulted in a number of "warts" with CoT's entity names. For example, why did we use *ce*, *le* and *event* instead of *radius*, *height*, and *cot* respectively? No good reason, but we did, and so we're stuck with it, and remember that your XML parser doesn't really care.

The CoT Approach

CoT is an object hierarchy for the DoD's tactical information. That's all. But that is what distinguishes CoT from other DoD message standards. CoT completely ignores community lines and abstracts the commonality from the underlying data. In true OOP fashion, common data items are "bubbled up" into base classes from which other, more detailed, classes can be "derived."

----cut---

CoT sprang from a realization of the frightfully obvious fact that information from completely unrelated communities often exhibits a very similar internal structure.

This is nothing new. In fact, it's obvious to anyone with object-oriented programming experience that finding commonality is the key to building a good object hierarchy. Common (shared) attributes and methods are placed in base classes from which more refined object classes are derived and more attributes added.

Furthermore, unlike other DoD standards, CoT is being continually developed by experimentation. What I mean by "continually developed by experimentation" is that CoT's object hierarchy is being extended by collecting and analyzing reams of data from DoD systems. In CoT, the classes are made to reflect the data, not vice versa.

One of the wonderful things about this approach is that it is parallelizable. One group can be looking for patterns in METOC data while another looks for patterns in CSAR reports. In fact,

most of our derived classes (subschemas in CoT parlance) come from domain experts who just happen to be using CoT. CoT's development path is looking much like the Internet's whole RFP process, and it works well. (We have a similar "draft," "proposed," "standard" track as well.)

Existing CoT applications won't break with the introduction of new classes; those applications just won't be able to take full advantage of the new classes. They still can deal with the new information polymorphically, though.

And, perhaps surprising to some, CoT's object classes never seem to obey community lines. For example, TST, ISR, and CAS communities all have the concept of an "engagement;" CoT makes an "engagement" object that serves them all. The side effect of this is that other systems who also understand "engagements" can be instantly interoperable with other "engagement" producers or consumers. Hence ADOCS could instruct a Navy TACTOM to "engage" without ever being in the TACTOM "tribe."

Implementing CoT

CoT's actual implementation is somewhat crippled by pragmatic issues. Specifically, to minimize implementation effort and foster acceptance, CoT needed to capitalize on the "hot" information technologies. XML was the best-of-breed. But, despite its benefits, XML isn't adequate for implementing object hierarchies; it's a document markup language, not an object transfer language. As such, XML provides no information protection, no multiple inheritances, no polymorphism, no virtual functions. In short, it's not the right tool for the job, but it's a tool that everyone can use easily. As you'll see, CoT abuses XML; to CoT, XML is like a Crescent adjustable hammer. XML purists hate that about CoT, but even when you don't have a hammer, some problems are still nails.

CoT's information model isn't dependent on XML, but it has been influenced by it. CoT would be cleaner if it didn't use XML at all, but anything else wouldn't be ubiquitous, and XML brought so many tangible benefits—parsers, editors, viewers—that the good outweighs the bad.

To overcome XML's shortcomings, CoT took quite a few liberties in representing "objects" and "inheritance." These are very unfortunate, and occasionally ugly, but largely necessary. The CoT object model (as captured in the UML diagram CoT.vsd) shows CoT's object model and describes some of the shortcuts we've had to take to represent that in XML. That document offers a view of what CoT strives to implement, while the XSDs show how it was built. Sincere apologies to any purists, whether XML purists or OOP purists, both will certainly be revolted.

Like all technologies, XML has its place, but it's not going to be XML that solves the information interoperability problem. We need something much more powerful and much less fashionable: common sense and programming discipline.

Why Isn't There One "Unified" CoT Schema?

CoT's objective is to provide an object hierarchy. As such, we want each new CoT "class" to reference its superclasses, but have no knowledge of, and subclasses derived from, it. In short, we need the *IS A* relationship to implement an object hierarchy.

Unfortunately, many XML tools want an XML document whose root explicitly names all the legal components of the document. This is precisely the *HAS A* relationship. We want an object hierarchy where the *root* of the class tree has no knowledge of the derived (more detailed) classes, but XML tools want a *root* that references all possible elements. Essentially, XML supports the *HAS A* relationship that allows aggregation, but it's lacking the *IS A* relationship needed for abstraction. If we were to make a "unified" CoT root schema (as is the XML way), we wouldn't have an object hierarchy any more.

----cut---

We're painfully aware that CoT's schema definitions "aren't like the others." It's not an oversight or accident; we're really trying to solve a problem that XML wasn't designed to solve. We want an object hierarchy, not a composite schema. We know that our approach renders some of your favorite XML tools inert, and we're very sorry for that.

It is very, very tempting to take a snapshot of the CoT object hierarchy and build an aggregate schema that will make your XML tools happier. Unfortunately, doing so "locks down" CoT's object hierarchy and actually breaks the whole concept behind CoT. So, please be aware that if you "convert" the CoT schemas into something your tools like, it won't be an object hierarchy any more.

We very strongly urge you not to do that, and if you're absolutely compelled to do it anyway, please do not propagate your work; it's going to break more than it fixes. Now, if you're a Java developer and you are familiar with JAXB (Java Architecture for XML Binding), you will feel this pain more acutely than everyone else will. JAXB provides you with some cute hooks that allow you to treat XML documents like JAVA objects. JAXB is a wonderful kit for manipulating XML documents, but it doesn't overcome XML's inherent limitations; JAXB does not make XML an OOP language.

If you're a JAXB user, you'll probably curse CoT for the liberties we've taken with XML. But remember CoT's objective: implement an extensible object hierarchy. JAXB can't do that, and if we bend the CoT schema to make JAXB happy, we subvert our own goal.

Choices Breed Complexity

Most standards claim they're "flexible" because they give a multitude of ways to represent everything. That's an implementation nightmare; you have to be prepared for absolutely anything and code up things that will never be used.

CoT doesn't do this. In CoT, there's precisely one way to represent a piece of information. Angles (including *lat/lon*) are always decimal degrees. MKS (meters, kilograms, seconds) is used everywhere. In general, we bend over backwards to make it simple to code. Sometimes we have to sacrifice coding simplicity for correctness (see HAE below).

----cut---

USMTF is notorious for offering choice. Location can be *lat/lon*, MGRS, UTM, range/bearing, all in a wide variety of formats, flavors, and colors. That looks wonderful on a viewgraph, but it is flat out awful to have to code it. As such, no one ever codes all the options. And we end up with interoperability problems when two groups use the same message but different value codings.

Many have complained that CoT does not allow both HAE and MSL. The problem is that every such alternative means coding work for every single system that accepts it. If your system uses MSL and you want to talk CoT, the burden is placed on you to use HAE, not on everyone else to support both.

This is probably the single biggest reason that folks can get "on line" quickly with CoT in hours. It's designed to be simple to implement.

Evil Enumerations

Many fields within CoT have a strange encoding resembling this:

"a-h-G-E-V-A-T-t". Some people argue that we should use enumerations for those fields. What they overlook is that you don't get polymorphism in an enumeration.

CoT is an object hierarchy. In an object oriented language, CoT would use inheritance to express hierarchy. For example, in an OOP world, a T-72 tank might have been represented as: "atoms::hostile::ground::equipment::vehicle::armored::tank::t72", and an instance of the t72 object would provide lots of interesting data through inherited methods and be accessible by polymorphic methods.

But, this is XML. We don't get object hierarchies or derivation, we have to make due. So, CoT records the path through an imaginary object hierarchy with a short-hand notation. In the interest of brevity and ease of coding, CoT abbreviates the path to short hyphen-separated strings. Hence, "a-h-G-E-V-A-T-t" means "atoms::hostile::ground::equipment::vehicle::armored::tank::t72", it's just shorter.

----cut---

But why not use enumerations for types?

Enumerations have one simple but fatal flaw: they can't give you partial information, so the sender and receiver must have identical enumeration tables to exchange information. If the sender uses an enumeration that the receiver doesn't know, the receiver cannot do anything with the

message. This introduces a huge interoperability problem when senders and receivers must upgrade; either everyone upgrades simultaneously, or suffer an interoperability problem.

The object hierarchy avoids this problem. If either sender or receiver has an older (partial) object hierarchy, it can still understand that the object is a "tank" even if it doesn't understand precisely what a t72 is. To another receiver which only understands the first three branches of the "atoms subtree," the type string "a-h-G-E-V-A-T" is still "a-h-G-<something>" and can be understood to be a hostile ground unit. CoT uses these odd constructs everywhere. It's the best we could do in the absence of an OOP.

Why are those '-'s in there? To allow higher cardinality in the derivation tree. For example, a-ABC-XYZ is understood to be the object a::ABC::XYZ. If we didn't have dashes, that would be a::A::B::C::X::Y::Z.

CoT's type hierarchy allows you to quickly filter out things you don't care about. For example, if you're an SA system interested in "atoms," and you get a message that doesn't begin with 'a-', you're done, move on. If your application does deconfliction and you're looking for friendly ground units, "a-f-G" gets you there.

CoT's Type Tree

CoT's type field is critically important to CoT since it's how we express objects. When you code CoT, you'll have to make branch decisions based on CoT's type. We recommend you do this by using the predicates in CoTtypes.xml. Those will insulate you from the details of the type tree, and we will maintain and distribute that tree (more on this in the Coding Hints section). The latest type tree is kept on the web site.

----cut---

CoT's type tree has very few branches at the root note. They are

a - atoms	(anything you drop on your foot)
b - bits	(a chunk of information, e.g., image or chat)
t - tasking	(any kind of request which elicits a response)
r - reply	(what you get in response to a request)
c - capability	(an "offer" of a service - e.g., shooting mortars)
r - reservation	(a "lock" on a chunk of space and time)

The Atoms tree is the most populated, it is based on MS2525B. (This is a change from V1.0 which used our own organization.) Also in broad use are the *tasking*, *reply*, and *bits* branches.

Avoiding Implicit Information

A weakness common in many current message standards is their reliance on implicit information. Sometimes it's very subtle. For example, all systems report coordinates, but relatively few ascribe an error bounds to those numbers. Coordinates are nearly useless without knowing how good they are. CoT makes these error bounds explicit. We express coordinates like a machinist would express measurements, e.g., 1 +/- 0.001. CoT tries to eliminate implicit information as much as possible.

----cut---

CoT makes information explicit as often as practical. We take heat for this. Most systems don't know their error tolerances and are offended when we demand them. If they can't put an upper bound on their errors we use 9999999 meters since we are confident that their coordinates are within +/-10,000,000 meters, at least if we're near the Earth.

Yes, calculating an error budget for a system can be painful, but it's necessary if we're going to be able to deal with information in a machine-to-machine manner.

What, When, Where, But No Whom?

Yes, there's no "whom." Many, many, many folks have argued that CoT is flawed because it doesn't capture "whom." This was not an oversight. Knowing "whom" is almost always used to bring some implicit information into play.

----cut---

It almost always turns out that folks who "absolutely need" to know "whom" want it so that they can use some other bits of implicit information they know from experience. For example, "I need it because if it's a SOF unit I know he'll have a laser designator."

In reality, wouldn't it be better to be told that he really brought a laser designator? And that it was working properly? And he was somewhere he could use it? All that information takes only one *bit*: "i_am_laser_capable=1". That *bit* is a heck of a lot more useful and reliable than experience, and it's much easier to make sure it's accurate; a built-in-test procedure can set that *bit*.

Lat, Lon and HAE?

We use *lat/lon* because it's ubiquitous and relatively simple. Other formats (UTM, MGSR) are much harder to code to. CoT uses HAE because it's much, much simpler, more accurate than MSL or AGL, and it's also what devices like GPS use (well, not really). MSL seems simpler at first, but it's fraught with interoperability problems that HAE conveniently avoids.

----cut---

Internally GPS uses an even simpler Cartesian coordinate system called ECEF. That would make life much, much easier since distances and speeds, spheres, would be much easier to calculate. The problem with ECEF is that nothing makes it available. It's used inside the GPS receiver, but they (in)conveniently convert to *lat/lon* or MGRS when they cough out their coordinates on the NMEA interface. Same is true of most every system out there.

CoT would like to use ECEF, but doing so would place a burden on absolutely everyone. Not a good trade-off. Pragmatism prevailed here and CoT took on *lat/lon*. Like XML, it's not perfect, but it's the right choice. Now, about the HAE/MSL issue:

MSL is empirically derived (as is AGL from DTED). MSL is based on an estimate of "sea level" which is affected by the density of the Earth's crust. "Sea level" is actually an irregular, undulating surface that's higher where the crust is densest. The shape of this surface is modeled by a big 2-D table of measurement, an Earth Gravity Model (EGM). The model is refined periodically (EGM84 and EGM96 are both in current use). Models are available in multiple resolutions (10x10 degree grids, 15x15 min grids, etc.), and there are multiple interpolation algorithms in use.

In short, MSL is an interoperability nightmare. You have to have the same model, gridding, and interpolation to exchange information, or errors will result. All this is avoided if we use HAE. HAE is a mathematical model with just a few parameters for ellipse axes. It's simple, accurate, and stable.

Where–Why A Cylinder?

CoT's base class models "where" as a vertical cylinder having of specific radius and half-height and centered on a specific point in space. For legacy reasons the cylinder's radius is called *ce*, and its half-height is *le*. The cylinder is sized so that it completely encloses the entire object with a 1-sigma certainty. A bigger object gets a bigger cylinder; and an object whose position is less accurately known gets a bigger cylinder. Basically the cylinder sums (linear sum) the positional uncertainty and the objects size.

----cut---

So, for example, a 1-meter radius Mountain Dew tank sitting in a precisely known location will have a CoT *ce* of 1. If that same tank's location wasn't known precisely, it would be in a larger cylinder. A half-meter positional uncertainty would make a *ce* of 1.5, 1.0 for the tank and 0.5 for the uncertainty.

Why does CoT sum size and error? For simplicity; it's extremely simple to implement. If it's ever really necessary for an application to separate the size from the position error, CoT will provide that additional level of detail in a subschema (a derived class). However, despite the academic's objections to this, we've never once seen a system that needs both.

But why does CoT use a cylinder? Simplicity. To calculate whether two cylinders overlap, you check if their heights overlap, then you need only compute the Cartesian distance between their centers and see if it's less than the sum of their radii. One line of code. Any other shape required serious machinations.

(Note that in the *lat/lon* coordinate system, a true "cylinder" is hard to calculate. By convention, we use the latitude scaling at the center of the cylinder when doing this calculation. This is one place where ECEF would be a true blessing. See HAE below).

My Object Isn't Cylindrical

Wrap it in a cylinder big enough to contain all points with a 1-sigma confidence interval. If you have a need to express more detail than a cylinder, use CoT's <shape> subschema. That'll give you ellipses, polylines, and DXF.

----cut---

Targeteers swear but are mistaken that their impact areas are ellipses. Target impact zones are really the intersection of an inclined cone of uncertainty with a nearly flat plane of the ground. The resulting shape is eccentric, it's not an ellipse.

CoT's shape schema allows you to have multiple "levels" of shape modeling. The lowest fidelity is the simplest; it's the cylinder of the base class. Within the shape class there are ellipses, polylines, and full DXF. The targeteers can use their ellipses as "level 1" models and, in a decade or so when they realize their impact areas aren't symmetric, they can add a "level 2" DXF model. It's a poor man's polymorphism.

Confidence

A common criticism of CoT is that CoT doesn't provide ID confidence on the CoT type. This omission was intentional, not an oversight. The problem is that providing a single "confidence" interval would be misleading. Every bit of information has a confidence interval associated with it. In thinking this through, it was pretty clear that this belonged in a derived "confidence" class. In practice, we've yet to need it.

----cut---

What do I mean that every bit of information needs a confidence interval? For example, there's a red oak outside my office. In CoT this may be (completely fictitious type) "a-n-p-d-t-o-r" which would be short hand for an object class:

atoms::neutral::plant::deciduous::tree::oak::red

What's my confidence in this?

I'm 100% certain it's atoms.

I'm 98% certain it's neutral (Steven King makes me unsure.)

I'm 100% certain it's a plant

I'm 98% certain it's a tree

I'm 90% certain it's an oak

I'm 70% certain it's a red oak

So, what one confidence weight should I apply to the ID? Hmmm...

[Aside: It seems that weighting confidence makes things very amenable to maximum likelihood decoding kinds of processing. This gets really, really interesting since it could be a key enabler for some new fusion processing. This will all go into a subschema when needed.]

Three Time Fields?

CoT's base class has three times associated with it: *time, start, stale*. They indicate when the information was generated, when it will become valid and when it ceases to be valid.

----cut---

To understand why three times are needed, imagine reserving a conference room. You make the call at 8:00, to reserve it between 9:00 and 10:00. You generated the request at 8:00, but the reservation begins at 9:00 and ends at 10:00. All three times are needed. The first, the *time* you called, is needed to sort requests to determine the most recent information. The second and third indicate the interval of time we're discussing.

Note that *time* needn't be earlier than *start*. I can describe a reservation I had this morning. I generate the information at 11:00, but it describes a time in the past.

By convention, CoT allows the *stale* time to predate the *start* time. This reversal is how CoT indicates "drop track." If the information was stale before it started, it's considered an overt indication that the sender wishes to cancel the data. (Note that many systems, including our own, often just make *stale* less than *time* to indicate drop. This isn't strictly correct, since it may still describe an interval in the past, but the effect on SA systems that only care about "right now" is the same. To them, information in the past gets dropped. A historical note: the only change between the version 1.0 CoT schema and the version 2.0 schema was the addition of the third time field, *start*. Prior to that there were only two times.

Taskings (Open-Loop vs. Closed-Loop Messaging)

CoT differentiates messages that require responses from those that do not. In CoT parlance, any message that will elicit a direct response is a *tasking*, and all *taskings* fall under one branch of CoT's object hierarchy. *Taskings* elicit responses from the receiving system. The receiver sends

back acknowledgments, status updates, and completion information. Full details are in CoT's request documentation which is available on the schema portion of the web site.

----cut---

Many message standards are organized based on message purpose or intended recipient, e.g., a single standard may have one group of messages for "Situational Awareness (SA)" and another related to "Command and Control (C2)." CoT doesn't organize messages according to their purpose or audience, but rather by what they have in common. As such, there are no explicit SA or C2 types in CoT since those are usage-based concepts.

All messages that require a response are, in CoT parlance, considered *taskings*. Don't get too hung up on the word *tasking*, it's just the class name [see "Finding the Perfect Word"]. All *taskings* have CoT types beginning with "t-".

Tasking objects carry additional information (sub schemas) about (1) where responses should be sent, (2) who is being tasked, (3) who authorized the tasking, etc. In an OOP world, these would be data members of the *tasking* class when it was derived from "event," but in the XML world, these extra attributes are carried in a subschema (unfortunately named <request>).

In an OOP world, a *tasking* object would have methods like

tasking→Reply(...);

tasking→Addressee();

tasking→Authorization();

In the XML world, you'll have to implement these by pulling information out of the <request> subschema.

Other documents on the CoT web site provide a rather lengthy description of CoT's request-response model. Please refer to that for more detailed information.

What's the Message Router?

What an IP router does at layer 3 for IP packets, CoT's message router does at layer 7 for CoT messages. And, just like a layer 3 router, you don't need the message router to connect up CoT applications.

----cut---

The message router is just a plain old CoT application. It's given a bunch of "business rules" that determine how messages flow through the enterprise, and it applies those rules to incoming messages. Pull the router and documentation from the website if you're really interested. It brings a tremendous amount of flexibility to the way systems are interconnected via CoT.

Like layer 3 routers, there can be more than one CoT router in play in a network. Concepts analogous to unicast, multicast, firewalling, spoofing, packet sniffing, NAT, are all available in the message router.

Coding Hints

1. The CoT debugger is your very best friend
2. Use the CoTtypes.xml files where possible.

----cut---

I maintain the debugger and message router myself. They will always have the latest CoT concepts built in. The debugger changes frequently. Get a new one as often as you can. If you need a feature, ask: it may already be in there.

If the debugger gives you a thumbs-up on your CoT interactions, many of your integration issues will be eliminated. We strongly recommend that you mail a releasable sample of your CoT XML (which the debugger can capture) to cot@mitre.org. We'll look it over and work with you to fix anything that's not right.

Open Squirt Close

A CoT object (<event>) describes only one object. There is no aggregation element defined, and there is no streaming model (where you can continue to get multiple CoT messages over a single TCP connection). This was done on purpose. If we allowed streaming, we'd have to require everyone to implement a framing mechanism. That goes against the CoT "keep it simple stupid" philosophy. Instead, for a reliable transfer, we open a TCP connection, "squirt" one CoT object, and close the connection.

We don't want to define an aggregate object as that makes message-level routing more complex. Message routing is an important part of how CoT does the whole enterprise integration thing, and we want that to be easy.

----cut---

If we were to pass aggregate objects, things like message routing would be nightmarishly complex. Furthermore, we'd have to tear things apart, package them back up for each CoT consumer. I firmly believe that the "one-at-a-time" model is definitely the right model.

The one downside of the one-at-a-time model is that it's not efficient for TCP transfers. If you have a lot of data, you end up with a whole bunch of lingering sockets. We handle this by moving SA data via UDP, reserving TCP for things like targets, maydays, images. (Note, also, that CoT is in no way dependent on being on IP network. We routinely run CoT messages over SATCOM links that are not IP.)

Information Linking (Aggregation)

CoT's <event> base class describes the *what*, *where*, *when* of a single object. This makes processing CoT messages remarkably simple, but the one-item limitation is inadequate for describing complex collections of objects (e.g., the multiple DMPI's of a single target, or multiple obstacles in a single drop zone.) To support these complex concepts, CoT uses unidirectional links. A CoT object may have zero or more links to other CoT objects (and links to non-CoT objects are in the works.) Links allow you to represent any acyclic directed graph (that is, a tree.)

----cut---

It's very tempting to define subschemas that describe the aggregate objects, but doing so negates the advantages CoT provided (simplicity, polymorphism). An alternative approach is providing a means to "stitch together" a number of simple objects.

CoT provides this "stitching" mechanism by a scheme called "linking." A CoT object can be "linked" to other objects (think "pointer"). Thus, a CoT "drop zone" object can contain any number of separate "obstacle" objects, each of which is a CoT object in its own right.

By making each component a CoT object in its own right, an application that doesn't understand the complex notion of "drop zone" can still process the individual "obstacle" objects.

Acyclic direct graphs seem like a very limited data structure choice, unless, of course, you're a lisp programmer. In practice, though, links provide a tremendous capability with very little complexity. CoT uses links to implement routes (routes in CoT are just a list of waypoints), link aircraft to their flight leads, link weapons to targets, SPI's to sensors, etc.

Each link has a relation type that describes the relationship of the two objects. Since the link is directed, the relation is usually clear.

Warts

CoT has warts, but mercifully they're small and few. We try to excise the big ugly ones early; so far we've been lucky. Ultimately, we expect that CoT will be recast in a new technology (XML++?) and we'll give it a face lift when that happens. Until then, we're going to live with the warts rather than inflict pain on everyone who's already implemented it.

Here's what a look in the mirror reveals:

- Affiliation (friend, hostile, ...) is in the type. That's not the ideal place, but it is the best place. It's only applicable to the atoms branch,
- MS2525B has some redundancies. For example, there are multiple representations for helicopter. This is a wart we adopted. We have pruned some branches of that tree and intend to prune more. The sole "official" CoT types tree is in CoTtypes.xml which ships with the CoT debugger.

- Freetext shouldn't be in the tasking tree. This is egregious enough that we're going to ask some folks to do some code rip-up. Removing freetext from that branch will simplify the implementation of the reply mechanism considerably.
- I added an entity called <uid> to hold the name of an object as it was known on other systems. This has confused countless people since there's a 'uid' attribute in the base schema, and a <uid> entity as one of the subschemas. This was a screw-up; I should have called it <alias>. But now everyone implements it, so we'll live with it as <uid>
- There are lots of unfortunate attribute name choices, *ce* instead of 'cylinder', *le* instead of 'height,' etc. They're unfortunate, but we're going to live with them; it's not worth asking coders to rip and risk bugs simply because we don't like a word's spelling.

CoT Predicates

CoT frequently uses funky, hyphenated strings to indicate class hierarchies.

As with other "magic constants," it's unwise to hard-code these class strings into your code. For that reason, there is a CoT-provided XML file-CoTtypes.xml-that contains a set of predicates. The file has entries in the file look like this:

```
<is what='friend'       match="^a-f-"       />
```

The "what" attribute is a predicate name, and the "match" attribute provides a Perl style regexp that matches any event of the specified type. With the addition of a predicate test method, is(), we code all our tests as:

```
if(event->is("friend")) { ... }
```

This is the recommended way to make runtime type decisions with CoT. The file will be updated as the class hierarchy is extended.

----cut---

Why is this necessary? The issue is that XML isn't an OOP language, so it provides no methods. If we had polymorphic methods, we wouldn't be dependent on this runtime type checking and branching in the application. These predicates provide a kind of poor man's run time typing mechanism. Yes, that's ugly, but without OOP support in XML, it's unavoidable.

Data Transport

CoT messages are transport agnostic; they can be moved by SOAP, FTP, TCP, sneaker-net, etc. The two most popular means for moving CoT messages are via raw TCP and UDP exchanges, a single CoT message per UDP packet, or a single CoT message per TCP session. We call the one message per TCP session "open-squirt-close." We don't advocate a persistent TCP connection for CoT message transfer because doing so requires a framing protocol (to delineate message boundaries) which TCP doesn't provide.

----cut---

While CoT is really an object hierarchy, in practice, CoT is only trying to solve the inter-COI data exchange problem, not tell developers how to architect their software. Therefore, CoT has relevance only when data being transferred between systems.

CoT's UID

The <event> element of CoT's base schema has an attribute "uid." There's been a little confusion about what this and how it should be used.

CoT is really an object hierarchy. We expect that an entity in the battlefield (e.g., a particular aircraft, vehicle, target, or image) will be represented by a CoT "object." CoT's UID is an intended to uniquely identify the object being described. The UID is an opaque string and does not necessarily convey any meaningful information. Its sole purpose is to ensure global uniqueness. (You may be familiar with the UUID concept; think of CoT's UID as being similar to UUID.)

For example, there would be a one-to-one correspondence between a CoT UID and a specific aircraft. As that aircraft moved through space, its CoT position reports—a new one every few seconds—would all have the same UID. This is how one knows that the messages all apply to the same object (that particular aircraft).

The thing that the UID is making "unique" here is the object (the aircraft), not the individual message. The UID is not unique to the message, rather it uniquely identifies the object the message is describing.

If there are multiple messages with the same UID, how does one know which message is most current? The attribute *time* indicates when the message was generated. The CoT message with the latest timestamp is presumed to be the most up-to-date information. (CoT presumes that all systems are reasonably well synchronized in time.) See the discussion on time for more details.

How Can CoT's UID Be Guaranteed Unique?

Recall that the CoT glues together other systems. Most systems use some internal scheme for differentiating between objects they're processing. Link-16 uses track numbers, FBCB2 uses URNs, ADOCS uses target numbers, etc.

CoT's UIDs are generated by combining the reporting system's designation (e.g., track number) with some globally unique prefix (e.g. "Link-16"). Thus, the CoT UID for track number 01520 in Link-16 might be uid="Link16.01520". This ensures that if the particular object is unique within the Link-16 network, it will be globally unique in the CoT world.

But remember that the CoT UIDs are opaque types. Do not attempt to dissect them to determine things like what track number the object had on Link-16.

The prefixes are administered much like domain names on the Internet. That is, there is a "numbering authority" that ensures that there are no prefix clashes. You can claim your own prefix by coordinating with the folks who answer mail at cot@mitre.org.

What is This CoT Object Called on <system->?

If the CoT UIDs are opaque, how can you tell what a certain systems (e.g. Link-16) internal numbering was? There is a sub-schema that captures the name of this object on various systems. It is an unfortunate mistake (wart) that this entity is named <uid>. A much better name would have been <alias>, but changing now would cause great pain for little gain.

The <uid> entity (distinct from the uid attribute in the <event> element) is an attribute-value paring which shows system name and alias on that system. For example,

```
<event uid="f37qPbs9z" ...
  <detail>
    <uid tadilj="01520" adocs="XY3241" fbfb2="1134527" />
    ...
  </detail>
```

This means that the CoT object with the UID "f37qPbs9z" is known on tadilj systems as track number 01520, and is known to adocs as XY3241, etc. In retrospect, it was probably unwise to implement this as we did; it would have been better if we had defined a structure that like:

```
<!-- WRONG - THIS WILL NOT WORK -->
<alias system="tadilj" id="01520" />
<alias system="adocs" id= "XY3241" />
```

However, this is yet another case where the die has been cast and we'll live with the <uid> implementation as it is.

What Next?

Please visit http://cot.mitre.org for background information about CoT, implementation details, examples, and some useful tools. Address any questions you may have to cot@mitre.org.