**Real-time Futures Graph Tracking**

**Visualization and Analysis Tool**

by

Joseph M. Fahey

S.B. EECS, M.I.T., 2010

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 2011



Author Joseph M. Fahey

Department of Electrical Engineering and Computer Science

April 18, 2011

Certified by_____

Soraya Stevens

VI-A Company Thesis Supervisor

Certified by_____

Leslie Pack Kaelbling

M.I.T. Thesis Supervisor

Accepted by_____

Dr. Christopher J. Terman

Chairman, Masters of Engineering Thesis Committee

## Abstract

A hybrid envisionment is a novel representation of a simulated state graph, specifying all possible states and transitions of the system, characterized by both qualitative and quantitative state variables. The Deep Green project creates a hybrid envisionment, called a *futures graph*, to depict all possible occurrences and outcomes of a combat engagement between friendly and enemy units on a battlefield. During combat, AI state estimation techniques are utilized to efficiently track the state of the battle in a futures graph, giving the commander an up-to-date analysis of what is taking place on the battlefield and how it the battle could turn out. Because state estimation of highly complex hybrid envisionments is a relatively unexplored and novel process, it is important to ensure that it is handled efficiently and accurately enough for usage on the field. This paper explores an approach for discerning the behavior in state estimation through the use of an analysis suite. By accompanying Deep Green state estimation with the analysis suite developed, estimation techniques could be benchmarked and analyzed over various implementations through both numerical and graphical metrics. The metrics generated greatly helped to improve the estimation algorithm over the course of its development.

Thesis Supervisor: Leslie Pack Kaelbling
Title: Professor of Computer Science and Electrical Engineering

# Contents

# Introduction

One of the most important features of the Deep Green project is a hybrid simulator, which aims to predict the outcome of a combat engagement between friendly and enemy units. Utilizing both qualitative process theory and stochastic simulation, the hybrid simulator constructs an envisionment called a *futures graph*, to depict all possible states and transitions that could occur on the battlefield [1]. Each state represents a unique situation with quantitative and qualitative data associated with various state variables, such as number of units, casualties, ammo levels and so forth. States are linked in the futures graph by transitions, denoting a possible change from one state to another that could occur on the battle field.

The futures graph is further read by a futures graph *tracker*, which processes information in real-time during the battle. The tracker is tasked with the challenge of estimating the current state of the battle conditioned on a stream of observations recorded from the field. The goal of this project is to design an analysis tool to analyze the tracker's performance in real-time and compare performance metrics across various implementations.

By providing both real-time and post-hoc analysis, the analysis tool becomes useful for tracker algorithm developers as well as the commander and staff on the battlefield. In the same manner that football teams review their performances on the field through film sessions, developers are able to attain an external perspective for debugging and critiquing their work. In addition, real-time analysis helps both the developers and commander and staff to follow tracking behavior more closely. Through experimenting with different software testing frameworks, visualization and tracker evaluation techniques, a futures graph tracking analysis tool has successfully improved

tracking capabilities throughout the development cycle.

# Background

The Deep Green project, awarded to BAE Systems by DARPA in 2008 [2], attempted to formalize many of the best existing approaches to qualitative simulation in its SimPath feature, designed to critique alternate battle plans by simulating multiple outcomes of adversarial plans. Courses of action (COAs) are sketched graphically by commanders using state of the art software and symbolism to represent friendly and enemy plans in consideration. The commander is able to sketch heavily detailed COAs to represent friendly and enemy behavior using a large library of combat tasks, spatial and temporal symbolism. Using an implementation of qualitative process theory [3] extended for its use with COA tasks, SimPath is able to produce a futures graph to represent an envisionment of enemy and friendly COAs head-to-head on the battlefield. In addition, SimPath uses a stochastic simulator to estimate the likelihoods of each state transition in the futures graph.

## 2.1 Futures Graphs

Futures graphs allow the commander to explore the space of possible outcomes using the likelihood and utility of different combat situations that appear in the futures graph. Once the futures graphs are analyzed by the commander, he chooses a particular COA to execute, presumably one that appears most likely to succeed against all enemy COAs considered. By simulating an envisionment detailing how various friendly and enemy COAs will play out when matched up, the commander can reduce uncertainty and more effectively critique alternate plans before choosing one to execute. While the details involved in qualitative process theory and stochastic simulation for

SimPath are outside the scope of this document, it is important to understand why futures graphs are created as well as the general processes involved.

The concept of a futures graph as an envisionment is relatively novel in use for the Deep Green project. The idea of an envisionment was originally proposed by Johan de Kleer in 1977 to represent the use of qualitative reasoning and causality for constructing a behavioral description [4]. In order to describe the behavior of an encounter between friendly and enemy units, a futures graph details possible occurrences in the form of *situations*. As briefly mentioned before, a situation is a state of combat and contains qualitative and quantitative values for its variables. For instance, a situation may contain a variable describing a friendly unit's ammo level as RED, meaning it is dangerously low at the time. All situations contain a unique set of values for its variables, so that no two situations may be characterized in the same way. To outline the behavior of a combat scenario over time, situations are linked together by transitions. By doing this, all possible occurrences on the battlefield may be described through a series of linked situations.

Another key component in simulating battle outcomes is the notion of concurrency. Because units often times interact with the enemy independently, situations do not describe the state of the entire combat scenario at one point in time. A COA described by the commander, for instance, may involve two units defending different areas of the battlefield. In the event that an enemy attacks only one of the areas, situations depicting the outcome of their engagement should be independent of any activity involving the other defending unit. Because of this, a futures graph describing a complete battle scenario is immediately broken down into *engagements*, where each engagement illustrates an interaction between a subset of the units described in the COAs. An engagement is simply portrayed by a direct acyclic graph (DAG) of situations, none of which may occur simultaneously. Through the use of engagements, situations are allowed to occur concurrently or exclusively depending on

the units involved.

Deep Green attempts to help the commander make two types of decisions during plan execution. The first is a decision point, which the commander creates during the COA sketching phase, prior to plan execution. While the commander is sketching the different possible friendly COAs that he anticipates will be necessary to analyze, he is allowed to create decision points in the plan, which define a branch in the futures graph. When analyzing a futures graph with a decision point during execution, the commander is able to consider the different outcomes from the decision point branches, and decide which option he would like to pursue. Decision points take various forms and include events such as whether or not a unit should attack the enemy at a particular location.

The second type of decision that a commander might make is whether or not to re-plan and re-sketch his COAs. While decision points are factored into the futures graphs and detail specific event outcomes for each option, re-planning and re-sketching is a much more drastic change and requires the commander to essentially start over at the battle's current state, beginning with the planning process. After re-sketching the COAs he would like to use, future graphs are produced, a COA is picked for execution and the process continues.

## 2.2 Observation Data

During plan execution, the commander continues to explore futures graphs involving the executed COA as new information is constantly being gathered about the current combat scenario. New information is gathered through messages called *battle updates,* created by units in the battlefield and received through a messaging client in the Deep Green architecture.

The battle updates are processed through a file known as a OneSAF document. The OneSAF document [5], or log, is constantly updated throughout the battle,

appending new messages, in Publish and Subscribe Services (PASS) format, to the bottom of the document. The PASS message format is based on an XML schema for organizing information carried in the message and is a standard used by Army C2 systems. Typically, each friendly vehicle in a combat scenario will report an update message once every few minutes to be added to the OneSAF document. At the end of most battle scenarios, a OneSAF document may contain many thousands of different PASS messages.

### 2.3 Tracking

One of the most important components in the Deep Green project is the *tracker*. Using incoming information from battle updates, the tracker attempts to track the current combat status in the futures graphs available. The tracker is tasked with determining which of the commander's sketched COAs the enemy is most likely executing, based on observations carried in the battle updates. The tracker communicates its estimate of the most probable enemy COA with the commander periodically throughout the battle. By conditioning the futures graph likelihoods on battle updates, the commander is allowed to dynamically assess futures becoming very unlikely and focus on the futures that are becoming more likely to gather a better understanding of the possible battle outcomes.

# Tracking Methods

While a future graphs maybe a relatively new concept, the idea of probabilistic reasoning over time, and more specifically, 'state' estimation as it is commonly described in theory, has implications in many areas of practical artificial intelligence applications [6]. The term 'state' is used loosely because the definition of a state may vary depending on the implementation of the reasoning model (where Deep Green uses

a situation to describe the state). The goal of estimation in such systems is entirely focused on estimating the likelihoods that each state represents the system, conditioned on past and present observation data. Thus, the techniques of estimation in modern artificial intelligence may be applicable for solving the Deep Green tracking problem.

In order to effectively analyze tracking behavior, it is necessary to explore the techniques involved in estimation. While it is not necessary to fully understand the algorithms involved, it is important to understand the general ideas in an attempt to abstract as much as possible from the tracking process. Under the hood of probabilistic reasoning in AI, there are many types of models, theories and algorithms tailored to the unique aspects of different estimation problems. At the heart of most probabilistic reasoning models in dynamic systems involving state estimation lies an implementation of a recursive Bayesian filtering approximation.

Recursive Bayesian filtering provides the underlying probabilistic approach that is adopted by the large category of estimation techniques known as dynamic Bayesian networks (DBNs), in which most reasoning models, such as Kalman filtering and particle filtering, fall under. Bayesian filtering is useful for estimating the probability density function of a state distribution, over time, given incoming noisy measurements in a dynamic system. The Bayesian filter requires a transition model and a senor model described by:

$$P\left(X_{t+1} \mid \mathbf{x}_t\right) \qquad P\left(e_{t+1} \mid \mathbf{X}_{t+1}\right)$$

transition model sensor model

where X represents the state, e represents the observation and t is a discrete timestep.

The Bayesian filtering model is broken up into 2 estimation steps, as well as one recursive step. The first step is called the 'a priori' estimate, or one step prediction estimate. It uses the transition model to estimate the state distribution one time-step forward given the current state distribution:

$$P(X_{t+1} | \mathbf{e_{1:t}}) = \int_{x_t} P(X_{t+1} | \mathbf{x_t}) P(x_t | \mathbf{e_{1:t}}) dx_t$$

The second step is called the 'posteriori' estimate, or 'filtered' estimate and uses the sensor model as well as an adjustment factor, alpha, to ensure probabilities will sum to 1.

$$P(X_{t+1} | \mathbf{e_{1:t+1}}) = \alpha P(e_{t+1} | \mathbf{X_{t+1}}) P(X_{t+1} | \mathbf{e_{1:t}}) dx_t$$

The filtered estimate uses the sensor model as a correction term to give a better state distribution estimate based on the noisy input e. Lastly, the recursive step simply states that the current state distribution is used as the input into the next time-step's 'a priori' estimate. The beauty of the Bayesian filter model lies in its recursive nature. It estimates the current state distribution given only the previous time-step's state distribution and a noisy observation.

Kalman filtering is a particular instance of Bayesian filtering with added assumptions to allow exact inference to be computed efficiently [7]. The Kalman filtering model makes the assumptions that the state variables being estimated are continuous in nature. The Kalman filtering model also assumes that the state distribution, transition and sensor models, as well as the observation noise are all capable of being modeled by a multivariate Gaussian distribution. These assumptions are made because of the convenient nature of Gaussian distributions to remain closed under linear transformations. This means that both the one step prediction and recursive state distributions will also be described by multivariate Gaussian distributions, and so forth. This facet of Kalman filtering provides a powerful model when approached with an adaptive sampling or state estimation problem.

The general Kalman filtering model is an efficient and capable solution for state estimation, but there are a number of drawbacks that make it unattractive. For one, the assumptions made by the Kalman filter make it particularly restrictive for many applications involving non-linear, non-Gaussian dynamic systems, such as in Deep

Green. When the number of state variables is large, computation of the the Kalman gain may also become intractable.

In order to overcome these non-linearities, many dynamic systems consider an augmented version of the Kalman filter, called the ensemble Kalman filter (or EnKF) [8]. Even though the EnKF uses full non-linear dynamics to propagate the a priori error statistics, the EnKF assumes that all probability distributions involved are Gaussian. Each variable represented in the state distribution is extended to be a vector representation containing N particles to describe the variable's non-linearity. This collection of state vectors is known as the 'ensemble'. The transition and sensor models work exactly the same, as does the rest of the system. The error covariance matrix involved, however, is approximated using the ensemble mean instead of using the traditional equations called for in general Kalman filtering. By approximating the matrix using the ensemble mean, calculation of the Kalman gain matrix becomes a much more computationally feasible process. From here, the state estimate may be calculated and represented by a new ensemble. Just as the general Kalman filter performs its recursive update, the new ensemble is pushed back into the prediction step and the process continues.

One of the most common implementations of estimation in a dynamic Bayesian network, which the Deep Green tracker most frequently attempts to utilize, is particle filtering. The particle filter uses a similar technique to the ensemble Kalman filter in which it begins its estimation by accumulating a group of N particles to represent the non-linear probability distribution function of a state variable. However, the update process the particle filter performs is different than the EnKF. For one, there are no simplifications made based on Gaussian dynamics. Each particle maintains a 'weight', indicating its probability estimate for representing the current state. Upon each time-step, the particles are moved forward through the 2 step recursive Bayesian filtering process: first using the transition model to generate a prediction, then adjusting this

estimate with the observation seen from the battlefield. The resulting probability estimate is stored in the particle's weight. A new group of N particles are then randomly selected from the old particles with the probability of selecting a particle proportional to its own weight. This process is known as sample importance re-sampling (SIR). The resulting particles are adjusted to be unweighted for the next time-step and represent the non-linear state distribution similar to the ensemble in the EnKF. The update process for the particle filter does not attempt to utilize any Gaussian approximation or covariance estimate, as is done in the Kalman filter, but instead allows any known probability density function to represent model dynamics.

# The Problem

The tracker uses probabilistic reasoning over time to effectively parse the battle updates and update each futures graph's likelihood accordingly. Likelihood estimates are updated to assist the commander and staff in effectively making better decisions at decision points and times to re-sketch a group of possible COAs. Essentially, the tracker is the only component available to help the commander and staff make decisions during plan execution by providing incite on enemy behavior. While it uses an intricate algorithm to constantly update futures graphs with each new battle update, it is not transparent enough to allow the end-user to see exactly what is going on in the algorithm. Previous to this project, the tracker had only been exposed to black-box testing analysis. Because testing the tracker's performance is noticeably difficult, requiring a message stream of battle updates and corresponding futures graphs, one might feel hesitant about tracking implementation and reliability. In order to effectively allow the developer, as well as the commander, to understand tracking implementation, the visualization and analysis tool described in the following sections

was created to monitor performance metrics, analyze performance across implementations and generally make tracking behavior more apparent.

# Related Work

In general, a futures graph and the commander's decision points are very similar to an acyclic, directed Markov decision process (MDP) [9], where situations are the Markov states and actions are decision point options. Over time, situations progress in one direction, eventually terminating at some end state node. Unlike MDPs, however, futures graphs contain many (nearly all) state transitions based solely on probability. The reason that a futures graph contains mostly probabilistic transitions is because most of the 'decisions' the commander has already assumed to make when originally planning the COA. In this sense, a futures graph is a MDP with many of the actions predetermined. The remaining actions are decomposed from the decision points the commander makes during the planning phase. In addition, MDP models typically involve a decision policy to dictate the optimal action to take at each decision point. Optimality is specified by a function that takes into account the utility of the various outcomes from each possible action. The Deep Green model accommodates a decision policy by marking each situation with a percentage combat power, denoting the relative number of casualties. By doing so, a commander's decision policy is defined by choosing the action that maximizes the percentage combat power in resulting situations. This way, the commander is able to perform the action that is likely to result in the lowest number of casualties.

While an MDP may effectively describe a futures graph containing decision points installed the COA, an MDP does not accurately take into the commander's second decision capability – to re-plan and re-sketch enemy and friendly COAs.

Because the commander only considers a small finite set of possible COAs for the enemy, the state space for possible enemy behavior is hardly covered. A futures graph accurately describes the total state space of a combat scenario conditioned on the fact that the enemy behavior is known. While it is impossible to consider every possible enemy behavior, the commander considers a subset of this possible space by sketching a few enemy COA possibilities. When he feels this is not a sufficient representation of enemy behavior, he chooses to re-plan and re-sketch new possibilities. Since the state space of possible enemy behavior is unknown, a partially observable Markov decision process (POMDP) may more accurately depict the entire Deep Green model. In a POMDP model, an MDP is generalized to take into account the fact that the state cannot be directly observed. Such a model is very similar to the Deep Green battle state space, where the actual enemy behavior is not directly observable.

Rao and Georgeff construct a similar model in the paper Modeling Rational Agents within a BDI-Architecture called a 'time tree' with states called 'situations'. The situation transitions in the time tree represent the 'choices' available to the agent at each moment in time, much like the idea of a decision point in Deep Green. While a futures graph details the world with both decision points and Markov probabilities, time trees are composed entirely of choices as branches, much like an MDP. Both graphs effectively model situations and decision branches in an appropriate layout for examining the world over time.

An interesting facet of Rao and Georgeff's paper is the introduction of the belief, desire, intention (BDI) model. Originally coined by Michael Bratman in his theory of human practical reasoning, BDI attempts to model a world in which an agent maintains a set of 'beliefs' about the world and a set of 'desires' broadly outlining states for which the agent has goals. Like a POMDP, Bratman's BDI model more accurately addresses the second type of decision a commander makes in Deep Green; planning and re-planning. While enemy COAs are unknown, the commander sketches multiple enemy

COAs that he 'believes' are integral in COA planning analysis. It would be impossible for the commander to sketch every possible enemy COA, which is why he adopts these beliefs about the 'world', or battle, in order to simply the problem at hand.

Analyzing probabilistic estimation techniques is a largely unformalized and seemingly unexplored area in software engineering, which makes this research difficult to relate. However, Desurmont, Machy, Severin and Delaigle address a slightly similar concern in evaluation techniques for a visual tracker in their paper 'Effects of Parameters Variations in Particle Filter Tracking' in 2007 [10]. While they claim there exists many implementations of visual tracking over the years, they express particular concern in the 'lack of standard evaluation process' in comparing implementations. Desurmont et al focuses on the need for objective criteria to evaluate the visual tracker's particle filter over time. They also note that good functionality may be evaluated through the use of both qualitative and quantitative metric data. The use of such analysis techniques allowed the group to successfully identify the superior tracking configuration for their needs, which aligns closely with the goals of this project.

Lastly, it is important to note that software testing is an integral part of enterprise software development. While there are dozens of different frameworks and styles of software testing, the design of the tracker makes testing particularly difficult. For one, the tracker requires actual field data from combat scenarios in specific OneSAF document with PASS message formats as its input. The tracker also requires a group of futures graphs to track against given the field data. Authoring a futures graph is a lengthy process and the tracker requires up-to-date futures graphs due to the constantly evolving standards and implementations considered throughout the software development cycle. In other words, the system is not backwards compatible with futures graph documents and they frequently need re-authoring. In addition to these resource constraints creating limited input, the tracker is also based on a largely complex probabilistic model. All of these facets contribute to the difficulty in relating

to existing software testing methods.

# Design and Implementation

The tools created to effectively analyze tracking behavior within the testing and analysis suite are outlined in this section. The first part of this section is devoted to discussing the customized set of test data and associated meta-data used for comprehensive testing. Following this discussion, an outline of the metrics used for analysis is introduced in 6.2. The metrics, and more specifically the cost metric, define the core components in analyzing tracking behavior. In addition, real-time graphing strategies and regression analysis are addressed to extend the communication and comparison of metric data during testing. In 6.3 the software implementation of the testing framework is discussed as well as the technologies involved in creating a robust interface for testing the tracker. The last part of this section gives an assessment of the tracker testing and analysis suite and how it was used by the development team.

## 6.1 Customized Test Data

One of the most significant assets in weather forecast research and development is that there is more than enough data for testing reasoning models. With hundreds of sensors constantly deployed, weather researchers have ample opportunity to critique the performance of their models and estimation techniques, making it a very successful field for using AI and estimation. For this reason, weather forecast is the most dominant use of AI and estimation techniques in today's research.

In contrast, being able to test the Deep Green tracker is a rare opportunity. As mentioned earlier, testing the tracker requires a OneSAF document containing thousands of messages describing a battle scenario. PASS messages are generated by

hardware installed on the vehicles operating in the field during combat scenarios. As one could see, such data is not readily available in large numbers, making development a much more challenging process. To make matters worse, the OneSAF log format changes frequently and it is important to keep up with the latest standards as older formats may produce undesirable behavior when inputted into the tracking system.

In addition to the OneSAF log, the tracker also requires a set of futures graphs for tracking. While this has already been described before, it is important to reiterate the necessity of having at least one futures graph strongly correlated to the events described in the OneSAF log. If the events laid out in the futures graphs used for tracking do not correspond to the PASS messages, tracker tests will not be practical or effective.

It is important that the tracker be provided with ample test data in order to sufficiently critique and adjust its implementation on a scale comparable to weather forecast research. One of the most necessary components in the tracker analysis and testing suite addressing this concern is the OneSAF document generator. The OneSAF document generator uses a futures graph as input to create a representative document in the correct OneSAF format. Being able to produce a document representing a path through a futures graph of choice was a highly desired tool for consistent tracker testing. The tool created was easily modifiable for keeping up to OneSAF standards and ensured as strong a linkage as possible between messages and futures graph. The capability for generating OneSAF test data provides Deep Green with the advantage of testing the tracker's probabilistic reasoning models on par with that of current weather prediction research groups. Since the creation of this component, tracker development activity has increased dramatically.

Another reason why OneSAF generation has improve the testing process is that it removes the belief factor in relying on actual combat scenarios. Since OneSAF documents represent actual field data, it does not provide underlying truth for the

correct enemy COA being demonstrated. In actual planned combat scenarios, it is nearly impossible to completely replicate both friendly and enemy COAs due to typical human error and miscommunication. While this approach to generating tracker test cases was previously the best approach, it was riddled with uncertainty and error. In addition, actual combat scenarios (as opposed to planned walk-throughs) are less useful based on the fact that observation data is the only evidence available for guessing the correct enemy COA. In a sense, providing the tracker with actual combat scenario data relies too heavily on the trusting the system's inference. This data was helpful for general system testing, such as whether or not the tracker throws an error during simulation, but it does not help to hone the qualitative and quantitative inference calculated in the Bayesian filtering process.

A strong benefit to the OneSAF document generation approach is that the output data may include various meta-data for categorization and ground truth. For instance, a OneSAF generated log may include a tag describing which futures graph was used for its creation. It also contains information detailing the number of messages, per message type, and which of the possible paths was taken through the graph. The system highly benefits from the inclusion of test data meta-data for generating a comprehensive analysis of tracking performance.

## 6. 2 Analysis Techniques

### Metrics

In order to be able to evaluate the performance of the tracker during execution, a set of comprehensive metrics needs to be available for analysis. In large and complex software projects, such as Deep Green, it is extremely important to emphasize the abstraction of various components, and the testing infrastructure is no exception. Although a sub-contractor is responsible for the implementation of the tracker, a comprehensive set of metrics should allow analysis regardless of its implementation. In

addition, the tracker's implementation may vary widely from version to version. This puts limitations on the implementation of the analysis tool.

It is important, however, to be able to maintain a set of metrics consistent with the invariable components of the tracker. From a black box testing standpoint, a core set of metrics may be developed for analysis of both the input and output variables for the tracker. For instance, it is important to understand the general concept of suboptimal Bayesian filtering to gather the sense of particles (or ensembles) as they are an integral part of the non-linear state estimation algorithm. With this understanding, a black box approach to tracker testing will be able to identify the number of particles the tracker used during testing, without knowledge of the filter being used, to better understand its performance (more about this is discussed in the performance metric section). While these metrics define the core component of tracker analysis, a second set of metric analyzers may be created to support further analysis of the various state variables (clear box testing), that define the trackers specific implementation.

**Cost Metric**

With the particle filter dictating military action in combat, the tracker's accuracy in determining futures graph likelihoods is the most important characteristic to be analyzed during a simulation. Since it is also what the developer and commander and staff are most concerned with, it must be analyzed thoroughly in the testing framework.

One of the best ways to critique the tracker's performance across various simulations is by developing a cost metric to analyze accuracy over time in the simulations where we know the outcome. Previous tracker testing analysis conducted a go / no-go judgment by simply checking to see if the correct futures graph, assuming it was known, was predicted at the end of an entire battle simulation. However, a commander relies on accurate futures graph likelihood updates *throughout* the battle to

effectively make decisions in real-time. Because of the lack of effectively analyzing accuracy over time, the testing suite produces an accuracy metric that takes into account futures graph likelihood estimates over the course of the entire battle.

The simplest way to produce such a metric is to calculate the ratio between the number of times the tracker was able to correctly identified the most likely futures graph and the total number of likelihood updates. When testing with OneSAF generated documents, this is an easy process as the testing framework is capable of extracting the correct futures graph from the generated meta-data in order to calculate this ratio. When using an actual OneSAF document created in the field, the tester is required to manually input the correct futures graph information as an optional parameter to the testing framework. Through the use of this simple metric, the prediction accuracy of various iterations of tracker implementation can be analyzed and compared through the testing framework.

While this cost metric analysis provides a good overall analysis of the tracker's behavior, it does not quite paint a complete picture of the tracker's true accuracy. The assumption that is made in this simple analysis is that the correct futures graph should be the most likely futures graph at all points in time. However, this may not always be true. As mentioned before, the entire state space for enemy behavior is not fully defined during the commander's COA sketching phase. While an enemy COA may not be *completely* accurate, it may still be a valid representation of the enemy's behavior. This might mean that battle update messages may not completely correspond to the 'correct' futures graph at all times. Due to this incomplete matching, an incorrect futures graph may very well be the most likely futures graph for short durations during the battle. For instance, the tracker might encounter a scenario where the incoming evidence updates actually correlate strongly with an incorrect futures graph for a short duration at the beginning of a simulation. Using the simple cost metric analysis will produce bad results for the correct futures graph during this time period and skew the

overall results of the tracking performance.

In order to resolve this problem, one must have some predefined idea of what to expect at various times during the simulation process, or simply a general knowledge of the futures graphs and evidence variables and be able to input this knowledge into the system before conducting the test simulation. The best solution was to apply a time-sensitive weighting function to tracking errors. Currently, in order to provide the most flexibility for various utility functions, a Java method allows the tester to customize the implementation of a mapping between time and utility for use in cost metric analysis. For instance, an unbiased cost metric would need an implementation that returns a constant value for any time input given.

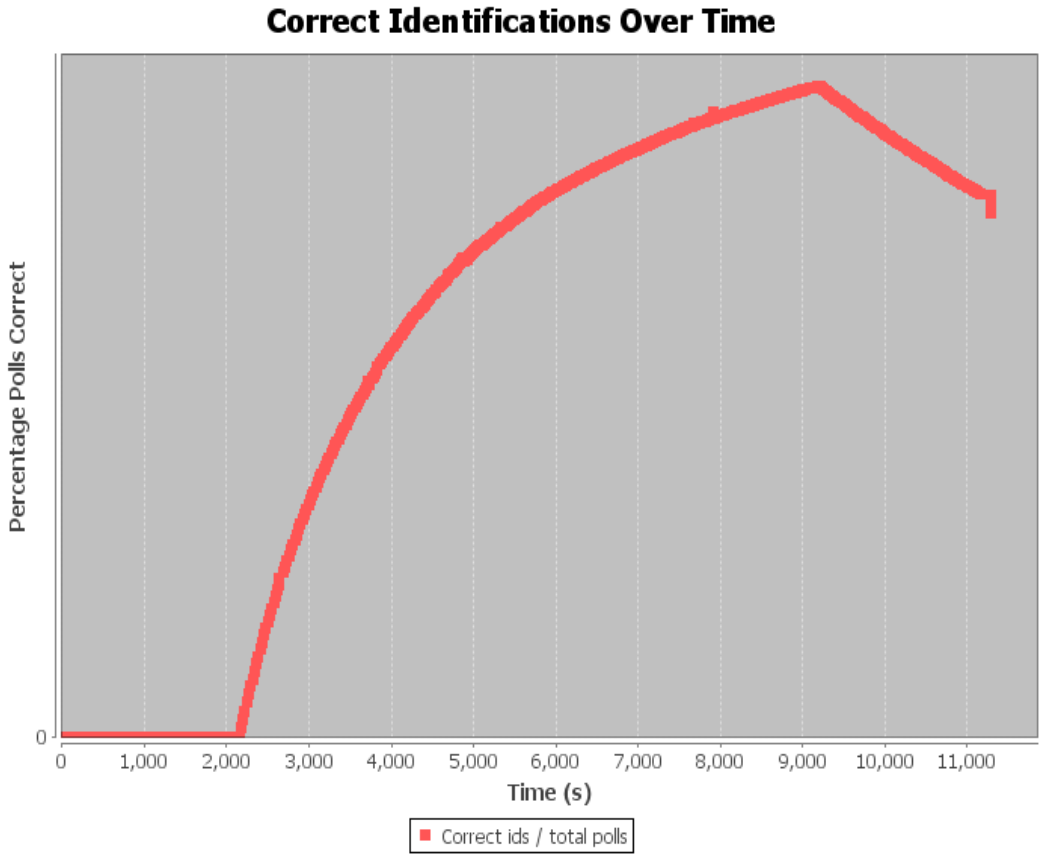


Figure 1. A simple constant cost metric.

In Figure 1 above a simple constant cost metric is used and the accuracy issues at the beginning of the battle give the tracker a bad overall correctness percentage. The futures graphs being used in this simulation all maintain similar starting situations. About a fifth of the way though the battle, however, the futures graphs were differentiated based on the evidence. It is clear from the graph that the end of the battle it fails to identify the correct futures graph and its accuracy degrades. In this case, a linear cost function may be more suitable to assess its performance. A linear function, returning a value proportional to the time input, would yield a cost metric weighting tracker accuracy more heavily as it nears the end of the battle. This type of utility function is popular in testing and solves the issue of putting less focus on tracker accuracy early in the battle. In summary, using a time-sensitive cost metric typically gives a better analysis of the tracker's accuracy.

**Message and Alert Counting**

Another metric was created to further explore tracking behavior by providing information about the number of PASS messages parsed over time. This is useful because the messaging client, which receives PASS messages from the battlefield, operates asynchronously and often times drops messages during simulation. The messaging client is handled by Apache's ActiveMQ and stores incoming messages in a queue for parsing. After the tracker reads a message and updates its estimation model accordingly, it will dequeue another message from the queue and repeat the process. The problem with this system is that occasionally messages will enter the queue faster than the tracker can dequeue them. One reason this might happen is when the tracker takes too long to parse each message. Another reason for this behavior to occur is when battle update messages are fed too quickly to the messaging client, which might happen for a variety of reasons. Since the queue is only of limited size, when it reaches capacity it will purposely drop messages because it has no other place to put them. A

metric detailing the message count shows the total number of messages, along with the amount of messages per message type, over time. Furthermore, the total number of messages received after a complete simulation may be compared to the meta-data in the generated OneSAF log to understand if messages were dropped, yielding possibly inconsistent tracking behavior.

Alerts are another integral component in the Deep Green system and are created for various reasons throughout the combat scenario. An example is the commander's critical information requirement (CCIR) alert. The CCIR alert notifies the commander when a particular event occurs that he wants to be notified about and defines in the creation of a COA. A CCIR created for a friendly unit crossing a phase line will alert the commander when position updates for that particular unit reveal the phase line has been crossed. Similar to the message count metric, the alert count metric provides details on the number of alerts, per alert type. The alert count metric is useful for regression testing and making sure alert behavior is consistent throughout repeated tests.

Using the total message count and alert times provides the developer with additional tools for debugging the tracker's behavior. Since debugging code line by line is a tedious and difficult process and loses all intuition when considering many particles or ensembles, it is beneficial to refer to message counts and alert times to diagnose behavior. For instance, if the cost metric reveals odd behavior at a specific point in time, such as the spike in Figure 1, the developer can refer to the message count at that particular time. This may be beneficial for locating and inspecting messages surrounding the spike. In general, providing the developer with metrics that give a better idea of the state of the system, such as these, is a majority of the debugging process.

**Performance Analysis**

One of the biggest concerns in probabilistic reasoning models is processing time. The original Kalman filter, as described in the tracking methods section, is rarely implemented due to the difficulty in computing the Kalman gain matrix in a reasonable time after each time-step. For this reason, many sum-optimal algorithms have been created and are more commonly used in practice. In the case of using a particle filter, ensemble Kalman filter or other particle based algorithms, it is important to consider the trade-off between number of particles being used and processing time. As the number of particles being used increases, estimation typically becomes more accurate but takes longer to process after each evidence update.

The Deep Green tracker most frequently uses a particle filter for state estimation and provides a convenient abstraction for adjusting the number of particles being used in its implementation. A performance metric calculates the ratio of total tracker processing time to battle length. This helps to understand the tracker's processing performance in terms of speed over various implementations.

The performance metric also outputs the number of particles being used to help find the best balance between accuracy and processing time. When comparing the performance of different implementations, it is important to take into account the number of particles each implementation used at the time. It wouldn't make sense to compare the performance ratio of two different implementations when they are not using the same number of particles. In addition, noting the number of particles used helps to determine the threshold for what the system can successfully process. If it is discovered that messages are being dropped, it is generally a good idea to reduce the number of particles, to reduce computation time. Reducing the computation after each message helps to reduce the risk of not parsing messages quickly enough and overflowing the message queue.

**Engagement Likelihoods**

Since the OneSAF document generator choses a path through the futures graph, it generates its messages for a list of situations within engagements to make up the path. Due to this structure, when using a OneSAF generated log the tracker should be very capable of mapping the messages back to the situations and engagements from which they were formed.

The particle filter implementation of the tracker attempts to do this by creating particles from the possible situations that make up the futures graph. Know this implementation technique, the testing framework is capable of monitoring engagement likelihoods by summing the weight of the particles in each engagement. This creates a handy monitoring tool for breaking down the analysis of many particles into a small number of engagements. While this metric is specifically tailored to the particle filtering implementation of the tracker, it is very helpful clear box analysis metric, especially when using generated OneSAF documents.

**Real-time Graphing Visualization**

Visualizations are typically used to convey information in a convenient and appealing manner to the viewer. While most visualizations perform this task, nearly all of them do it differently, conveying different information, and generating different levels of effectiveness for the viewer. The real-time visualization makes use of a tabbed window to allow the end user to quickly flip through graphed metrics for easy inspection during testing.

While it may be clear that an analysis tool depicting tracking behavior will help developers improve the system's performance and implementation, it may be relatively unclear why such a utility would benefit the commander. One of the biggest issues in the BDI model is belief accuracy. There is no mechanism that takes into account belief accuracy, or any guidelines that dictate how much attention the commander should invest in following his futures graphs versus simply re-sketching to find a better COA to

execute. Because the BDI model only enumerates outcomes from belief states, it does not model the entire state space. In other words, the commander is blind to the many possible states that do not appear in the futures graphs.

In this regard, it is important to consider how a real-time visualization could aid the commander in making his decision to re-plan. Since the futures graphs are based on the commander's beliefs about the enemy, abandoning the beliefs and choosing to re-plan falls entirely in his own judgment. For this reason, giving the commander an up-to-date analysis of the entire battlefield is essential. In providing graphed metrics displaying data in real-time, such as envisionment and futures graph likelihoods over time, the commander can attain a better understanding of the current combat scenario. With a better understanding, the commander can potentially reduce his uncertainty about when to update his COAs for reanalysis. From this feature, Deep Green becomes a more effective decision aid for the commander and staff.

## 6.3 Testing Framework

As with most testing frameworks, it is important to keep the testing code separate from the application of concern. Existing debugging code was already integrated with the tracking algorithm and only contributed to degrading the trackers performance, as well as making the code complicated and difficult to read. A proper abstraction is necessary for future implementations of the tracker to be as painless as possible while still being available for analysis.

One way that achieved this behavior was by integrating the analysis tool with the Spring testing framework. The Spring framework allows for non-invasive and agile development with minimal dependencies on the application code. Through this, instances of the tracker classes we are concerned with, defined in the spring framework, are injected into the testing framework. Integrating the testing framework with Spring is necessary for keeping a good abstraction and separation for the various analysis tools.

The Deep Green project is developed using the Maven project management tool. A new maven module, TrackerTest, is defined to provide the necessary abstraction for keeping the testing framework separate from the rest of the production code. Maven further enhances this abstraction by providing a project object model (POM) and assembly instructions within the TrackerTest project to allow a complete construction of the testing framework without re-defining the referenced Deep Green project code. This means that the code defined in the TrackerTest module only implements the testing framework, and all references it uses to the rest of the Deep Green project, such as running necessary services and starting the tracker, are downloaded as separate jars as needed without tedious specifications by the testing coder. The assembly instructions also help to package up the testing code with the necessary jars to create a snapshot file for running the entire system with the added testing framework.

Classes responsible for handling the analysis of tracker variables, whether it is from black box testing over general Bayesian filtering or algorithm specific variables in clear box testing, as implemented as 'monitor' classes. Monitoring classes are each responsible for keeping track of the tracker's variables and outputting metrics for inspection by developers and the commander and staff. Each class is implemented independently and can be instantiated within the Spring framework. By instantiating the classes within Spring, the user can conveniently decide which metrics to deploy for tracker analysis at runtime. With a large number of metrics available for analysis, system performance would degrade if all metrics are deployed for every run. Giving the option to choose which metrics to involve further extends the convenience and capability of the tracking framework.

The TrackerTest is capable of using multiple configurations for running a complete system test. A somewhat verbose, ad-hoc option allows the user to manually specify the OneSAF document and futures graph locations, as well as the other parameters necessary for running the system. The issue with this approach is that it

does not take into consideration any of the valuable meta-data created when using a generated OneSAF document. A more customized option allows for the user to input a test document containing all the necessary parameters defined as well as any meta-data that would benefit the analysis process. Similarly, after the tracker successfully finishes parsing all PASS messages, a custom test document, containing the tracking results and analysis data, is outputted for the user to inspect. The TrackerTest process is more abstractly described in Figure 2 below.

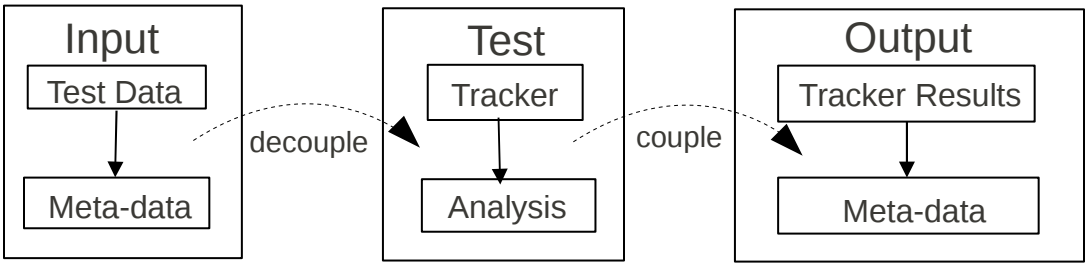


Figure 2. Testing Process Flow

A custom document, known as a *test object*, is created in order to utilize the storage of important meta-data in both input and output. Test objects are constructed using an XML schema, similar to other popular settings and configuration formats used in software applications. XML was chosen because it is highly customizable and simple to create. In addition, XML is flexible and takes minimal reconfiguration when wanting to add new meta-data to the input and output test objects.

A custom document object model (DOM) parser was implemented in Java to handle the meta-data used by the OneSAF generator in the input data, as well as the results stored in the output data. If a test object is used as input to the testing framework, the OneSAF log reference is decoupled from the 'log reference' property and the rest of the meta-data is passed to the metric classes. Similarly, the output from the tracker, a like of futures graph probabilities, is then coupled with the output from the

metric data and stored in an output test object.

**Regression Analysis**

One of the primary reasons for generating metric data for analysis is the purpose of regression testing tracker implementations. The generated output file containing the metric meta-data is capable of regression test comparison to other analysis output files of the same format. Regression comparison is completed in a separate component available in the TrackerTest module. The regression comparison component first decouples the meta-data from each output object then continues to compare and summarize like data. From regression analysis, developers and clients will be aware of the progress made in tracking.

### 6.4 Assessment

Usage of the TrackerTest and OneSAF generator tools was an informal process. With the immediate need for its functionality, a very basic implementation of the framework was put to use with only a few metrics. Over time, the framework's feature set grew to its current description in this document, and was used throughout its development. Since the TrackerTest framework requires very little integration configuration and is easy to run, all system testing done by the tracker development team used the framework. The test objects produced, containing all relevant data for analysis, were stored on the team's server. Informal meetings and conferences were conducted frequently to discuss the results and behavior of the tracker depicted in the test objects. Through this communication the tracker was able to address many performance and accuracy issues.

# Ongoing Work

The Deep Green project is an on-going process, constantly changing through code iterations and specification updates. Similarly, the tracker test analysis suite is also an on-going project and continues to aim at giving the developer and commander and staff a better idea of tracking behavior. There are a few components that are currently being explored and hope to be integrated soon for better analysis capability.

## 7.1 Interaction & Query

Most futures graphs that Deep Green operates with contain up to thousands of different pieces of data for all situations. In addition, with combat situations lasting multiple hours, the tracker needs to parse extremely large amounts of data. This creates a limit on the amount of information that the viewer is able to interpret at once. While part of the solution lies in filtering out important data to the visualization, as mentioned in the previous section, another useful functionality will involve the viewer being able to query specific bits of information in real-time. One particularly useful instance of this might involve querying for all battle updates during a specific time range. For example, using a time range between two consecutive alerts, the developers and commander might be able to locate messages causing the creation of the second alert. Information must then be structured for interaction and query in a simple and elegant manor.

## 7.2 Messaging Abstraction

As mentioned earlier, one of the key goals of this project was to allow active monitoring of the tracker's performance in real-time. In order to perform active monitoring, however, the monitoring framework must respect the processing demands of the tracker itself. Due to the intractability concerns in probabilistic reasoning models and general desire for maximum accuracy, the tracker will certainly need as much of the

processing power it can manage. It is important that monitoring the tracker does not hurt tracking performance.

While Spring allows a handy abstraction, it still requires the code used for analysis to be run in the same virtual machine as the code it is monitoring. In the use case of Deep Green, this poses a potential issue due to the tracker's high demand for processing power. With both the analysis tools and tracker processing large amounts of data on the fly in the same virtual machine, the tracker's processing performance is limited.

To handle this issue, the analysis suite is being further modularized into data handling and graphing utilities. Data handling uses the monitor classes to retrieve state data from the tracker. From here, data will be passed through a separate messaging pipe for further processing and graphing. The use of a messaging pipe allows the analysis suite to put processing and graphing utilities in a separate virtual machine. This way, the processing demand for tracker analysis will not take away from the tracker's performance. After its completion, the new framework may be compared to the old framework through the performance metric in regression analysis. This separation also allows the tracker to run on a high performance server without a display, while a client computer with a display can view the real-time visualization.

# Conclusion

The TrackerTest framework provides a powerful abstraction used by many members of the Deep Green team, not only for tracker analysis but also for general system testing. The OneSAF document generator was already demonstrated at a recent review by a client and has gained their interest for testing other C2 integrated military software. While the particle filter was practically the only tracker implementation

tested in production code for a very long time, new alternative Bayesian filtering implementations are currently being worked on. With the testing framework setup for both clear box and black box testing of the tracker, these new implementations will have a much easier time being tested.

Through the use of a real-time tracker visualization and analysis tool, the design and implementation of the Deep Green tracker can be evaluated over time in an abstract, yet comprehensive manor. Both the tracker developers and commander and staff can benefit from the use of this tool and the metrics that it produces. Overall, this project has made unique advances for dynamic testing and interactive visualization support for probabilistic tracking in the Deep Green project.

# Bibliography

[1] Thomas Hinrichs, Kenneth Forbus, Johan de Kleer, Sungwook Yoon, Eric Jones, Robert Hyland and Jason Wilson. Hybrid Qualitative Simulation of Military Operations. Technical report, 2010.

[2] U.S. Department of Defense announcement No. 466-08, June 02, 2008. Online: http://www.defense.gov/Contracts/Contract.aspx?ContractID=3787

[3] Kenneth Forbus. Qualitative Process Theory. *Artificial Intelligence, 24* pages 85-168, 1984.

[4] Johan de Kleer. Multiple representations of knowledge in a mechanics problem-solver. *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 299-304, Cambridge, MA, August 1977.

[5] Robert L. Wittman, Jr. and Cynthia T. Harrison. OneSAF: A Product Line Approach to Simulation Development. Technical Report, The MITRE Corporation, February 2001.

[6] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Third Edition, Prentice Hall, 2010.

[7] Greg Welch and Gary Bishop. An introduction to the Kalman filter. Technical report, University of North Carolina, Department of Computer Science, 1995.

[8] Geir Evensen. The Ensemble Kalman Filter: Theoretical Formulation and Practical

Implementation. *Ocean Dynamics*, pages 343-367, 53-2003.

[9] Gerardo I. Simari and Simon Parsons. On the relationship between MDPs and the BDI architecture. *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1041–1048, 2006.

[10] Xavier Desurmont, Caroline Machy, Celine Mancas-Thillou, Derek Severin and Jean-Francois Delaigle. Effects of Parameters Variations in Particle Filter Tracking. *IEEE International Conference on Image Processing*, pages 2789-2792, Mons, Belgium 2006.