FOI
TOTALFÖRSVARETS
FORSKNINGSINSTITUT

FOI-D--0184--SE
November 2004

Simon Ahlberg, Pontus Hörling, Christian Mårtenson, Göran Neider, Hedvig SIdenbladh, Pontus Svenson, Per Svensson, Johan Walter

System Documentation on the FOI Information Fusion Demonstrator 03 (IFD03)

Ledningssystem
SE-581 11 Linköping

TOTALFÖRSVARETS FORSKNINGSINSTITUT FOI-D--0184--SE

Ledningssystem November 2004

Box 1165

581 11 Linköping

Simon Ahlberg, Pontus Hörling, Christian Mårtenson, Göran Neider, Hedvig SIdenbladh, Pontus Svenson, Per Svensson, Johan Walter

System Documentation on the FOI Information Fusion Demonstrator 03 (IFD03)

Utgivare Totalförsvarets Forskningsinstitut - FOI Ledningssystem Box 1165 581 11 Linköping	**Rapportnummer, ISRN** FOI-D--0184--SE **Forskningsområde** 4. Ledning, informationsteknink och sensorer **Månad, år** November 2004 | **Projektnummer** E 7097 **Delområde** 41 Ledning med samband och telekom och IT- **Delområde 2**
Författare/redaktör Simon Ahlberg Pontus Hörling Christian Mårtenson Göran Neider Hedvig SIdenbladh Pontus Svenson Per Svensson Johan Walter	**Projektledare** Per Svensson **Godkänd av** Johan Mårtensson **Uppdragsgivare/kundbeteckning** FM **Tekniskt och/eller vetenskapligt ansvarig** Per Svensson
Rapportens titel System Documentation on the FOI Information Fusion Demonstrator 03 (IFD03)	
Sammanfattning (högst 200 ord) Denna rapport innehåller systemdokumentation över informationsfusionsdemonstratorn IFD03. Bakgrunden till utvecklingen av demonstratorn beskrivs kortfattat, följt av kapitel om utvecklingsmiljön och hur man visualiserar resultaten från demonstratorn. Terrängmodulen, simuleringsmodeller i Flames och analysmetoderna dokumenteras här och på den medföljande CD-n.	
Nyckelord	
Övriga bibliografiska uppgifter	**Språk** Engelska
	Antal sidor: 86 s.

System Documentation on the FOI Information Fusion Demonstrator 03 (IFD03)

Simon Ahlberg, Pontus Hörling, Christian Mårtenson, Göran Neider, Hedvig Sidenbladh, Pontus Svenson, Per Svensson, Johan Walter

(November 2004)

TABLE OF CONTENTS

System documentation IFD 03

The IFD03 uses a combination of methods and program to show a conceptual view of how an information fusion system might work in the future.

1 Reading guide

This document is based on a linearised version of the online Web documentation for IFD03. The Web documentation is also available on the accompanying CD. It has been edited somewhat, but all duplications, omissions or formatting errors may not have been found. For the more detailed version (which includes documented source-code and call-graphs), see the html-version currently at

\\Flamma\Infofusion\IFD03_frozen_documentation

Section 2 lists papers that are directly relevant for IFD03. Sections 3 and 4 contain background information on IFD03 and some information on the scenario that is used. Section 5 describes how to get source-code and compile the system, while section 6 has instructions on how to use the Visualizer to look at the output of IFD03. This section also contains information on how to develop the Visualizer. The detailed workings of the IFD03 are described in sections 7-9, which list and describe all Flames modules, the analysis modules implemented in Matlab and the connections between the C and Matlab parts of the code. The online version of section 9 also lists all the Matlab source code.

The rest of this section consists of suggested reading for different possible uses. It is *strongly* suggested that this document be used only as a starting point for getting to know the system –please refer to the html-version for details and cross-references.

1.1 To be able to perform a demonstration using Flash

Skim sections 3 and 4 to understand some of the background of IFD03.

Read sections 5.3 and 6.1 and make sure that you can run the visualizer.

1.2 To be able to run IFD03 and generate new data

Skim sections 3 and 4 to understand some of the background of IFD03.

Read section 5 and make sure that you can compile and profile the system. Note in particular the instructions for setting environment variables in section 5.3.

Read section 6.1 and make sure that you can run the visualizer.

1.3 To be able to make changes to the scenario

Skim sections 3 and 4 to understand some of the background of IFD03.

Read section 5 and make sure that you can compile and profile the system. Note in particular the instructions for setting environment variables in section 5.3.

Read section 6.1 and make sure that you can run the visualizer.

Read Flames documentation to understand how to use Forge to change the scenario.

1.4 To be able to add new sensors

Skim sections 3 and 4 to understand some of the background of IFD03.

Read section 5 and make sure that you can compile and profile the system. Note section 5.1 for information on how to store data files needed for your sensor and the instructions for setting environment variables in section 5.3.

Read section 6.1 and make sure that you can run the visualizer.

Read section 7. Note section 7.2 on terrain data.

Read sections 8.1-4 to understand how to interface your module to FusionNode. Read relevant sections on how to communicate with the FusionNode Flames module.

Read other material from section 8 as necessary to understand how to write your sensor.

1.5 To be able to add new or change old analysis modules

Skim sections 3 and 4 to understand some of the background of IFD03.

Read section 5 and make sure that you can compile and profile the system. Note in particular the instructions for setting environment variables in section 5.3.

Read section 6.1 and make sure that you can run the visualizer.

If your method requires “3rd screen” output, read section 6.3. Read section 6.2, 7.1 and 7 to understand how to provide output from your module to the visualizer.

Read section 7.2 to understand how to use terrain data, if this is necessary for your module.

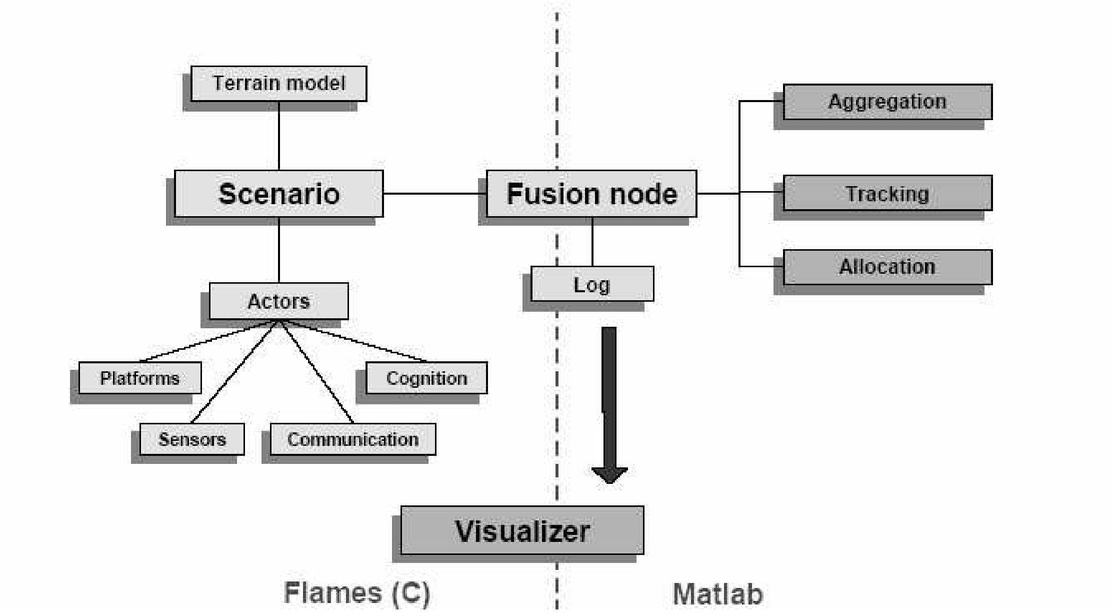
Read sections 9.1-4 to understand how to interface your module to Control.

Read other material in section 9 and paper A from section 2 to get background information on current analysis modules.

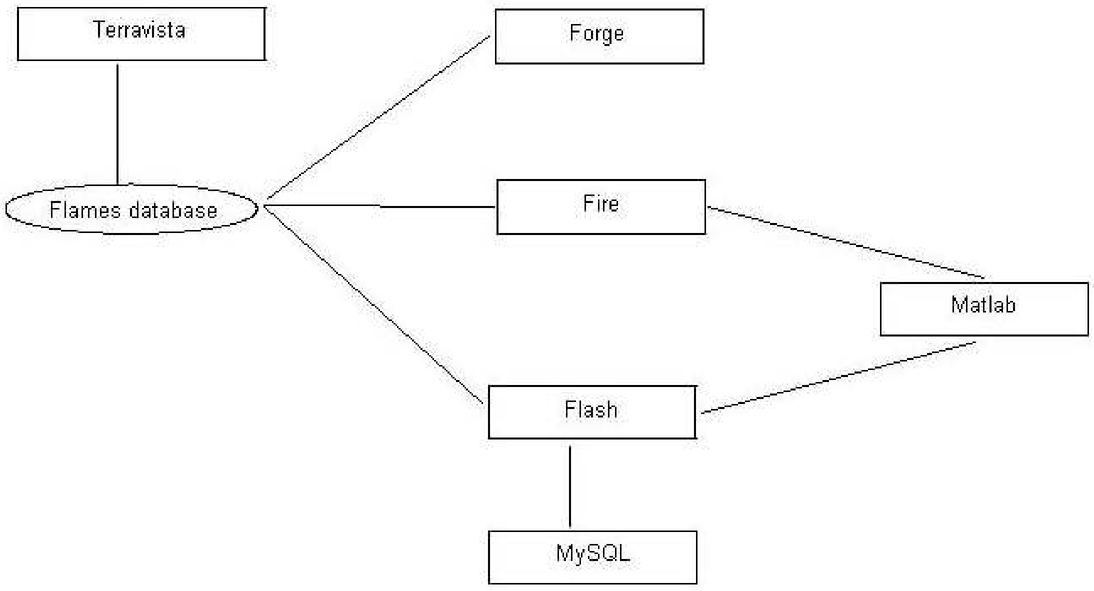
1.6 Map of logical connections of fusion node and simulator

IFD03 consists of a number of executable files. Forge is the modified Flames-program used to build a scenario. Scenario, sensor and analysis modules are linked into the fire program which runs the simulation and produces output data. Output is inserted into a SQL-database using a set of program described in the section on Visualization. That section also describes the modification we have made to the Flames visualization program flash. Terrain information is created by the Terravista program. Its output is stored in the Flames database.

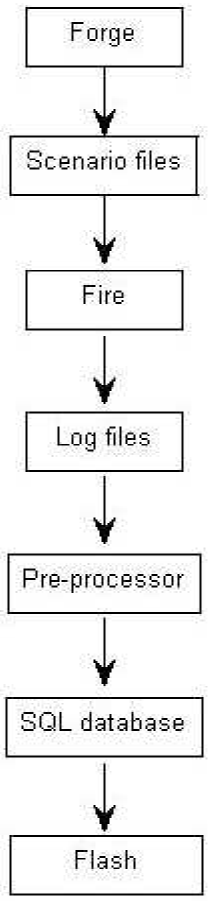
The following image gives a conceptual overview of IFD03:



The following figure aims to give a view of the relationships between different components used in IFD03. It can be used to determine which other components are affected by changes in one of the commercial software products used to develop IFD03. Note thatIFD03 uses modified versions of Forge, Fire, and Flash.



A view of the data flow in IFD03 is



Note that IFD03 also relies on configuration files and parameters that are set in the initialization routines of Fire. Configuration files are used for defining parts of the scenario and for defining sensor characteristics; see the corresponding documentation.

2 Articles directly relevant for IFD03

A. Schubert, J., Mårtenson, C., Sidenbladh, H., Svenson, P. and Walter, J.
Methods and System Design of the IFD03 Information Fusion Demonstrator
In *Proceedings of the Ninth International Command and Control Research and Technology Symposium,* Copenhagen, Denmark, 14-16 September 2004, US Department of Defence CCRP, Washington, DC, 2004.

B. Ahlberg, S., Hörling, P., Jöred, K., Neider, G., Mårtenson, C., Schubert, J., Sidenbladh, H., Svenson, P., Svensson, P., Undén, K., and Walter, J.
The IFD03 Information Fusion Demonstrator
In *Proceedings of the Seventh International Conference on Information Fusion* (FUSION 2004), Stockholm, Sweden, 28 June-1 July, 2004. International Society of Information Fusion, 2004.

C. Schubert, J.
Clustering belief functions based on attracting and conflicting metalevel evidence using Potts spin mean field theory
Information Fusion Vol 5, No 4, 2004.

D. Sidenbladh, H.
Multi-target particle filtering for the probability hypothesis density
In *Proceedings of the Sixth International Conference on Information Fusion* (FUSION 2003), Cairns, Australia, 8-11 July 2003. International Society of Information Fusion, 2003, pp. 800-806.

E. Schubert, J.
Evidential Force Aggregation
In *Proceedings of the Sixth International Conference on Information Fusion* (FUSION 2003), Cairns, Australia, 8-11 July 2003. International Society of Information Fusion, 2003, pp. 1223-1229.

3 Background

Excerpts from papers describing and motivating the design of IFD03:

Our project aims to complete the development of a demonstrator system for tactical information fusion applied to simple ground warfare scenarios, and to perform a demonstration using this system in the Fall of 2003. The system will be called Infofusion demonstrator 03 (IFD 03). In the information fusion area there does not yet exist a scientific basis for the development of integrated systems which could be put to practical use after restructuring for robustness, security certification etc. The main purpose of the demonstrator project is to provide a research platform for experimentation with specific research issues, in particular the interplay between different modeling techniques used to address subtopics in this research area, as well as to create a means of spreading knowledge to interested parties about the current state of research in information fusion.

IFD 03 will integrate methods related to different fusion "levels", specifically multisensor-multitarget tracking, force aggregation, and multisensor management. It will exchange data in simulated real time in both directions between the scenario simulator and the fusion system. It will have three closely associated main capabilities: to provide a test bed for new methodology in information fusion, to provide a supporting scenario simulator for the generation of adequately realistic sensor and intelligence reports used as input to the fusion processes, and to offer general-purpose software tools, terrain models, and other prerequisites for visualization both of the development of the scenario over time and of selected characteristics and effects of the fusion processes.

Over the past few years FOI has acquired a simulation development platform, based on the commercial simulation framework Flames^{TM} , suitable for test, experimental evaluation, evolutionary development, and demonstration of many kinds of event-driven scenario-based simulations. To adapt the Flames framework to the needs of information fusion research, advanced terrain modeling facilities were included, allowing fully integrated ("correlated") topographical and thematical models of geography to be used in the simulations. Recently, this platform was further extended by allowing program modules, developed using the problem-solving environment Matlab^{TM} to be tightly integrated. Thus, the resulting development platform allows comprehensive reuse of commercially available software, as well as both program modules and scenario specifications previously developed by our own group or by other projects at FOI.

Key to achieving successful demonstrations will be appropriate visualization methods which can render concrete and tangible concepts like scenario, fusion node, sensor network, communication system, and doctrine. In future projects the demonstrator system may be extended with methods for the solution of new problems, such as generation and analysis of alternative forecasting and action options. The combined Flames-Matlab development environment should significantly facilitate the development and integration of such methods.

3.1 Why build an information fusion demonstrator?

While any scientific approach to understanding specific aspects of reality will have to be based on abstraction and isolation of each aspect considered, on the other hand, in the scenario-based forecasting models we want to build based on understanding obtained by reductionist approaches, all significant complexities of the real system need to be reflected. Thus, e. g., during the last half-century, weather forecasting has gradually developed, not primarily by discoveries of new, meteorologically significant physical phenomena, but by a combination of better mathematical models of the underlying physics, improved algorithms for their evaluation, improved data collection and exploitation in models, and last but not least, a gradually increased complexity and sophistication of integrative, scenario-based forecasting models, made possible by the exponential growth in computational capacity.

Granted that information fusion adds the serious complication of hidden, antagonistic human decision-making to the purely physical processes of weather forecasting models, the success of such modeling could anyway, we believe, provide some inspiration for information fusion research, although this research certainly has a long way to go before it can claim any comparable success. So when will information fusion methodology have progressed sufficiently to make meaningful use of synthetic environment scenario simulators? Out of conviction that all necessary ingredients of complex forecasting models need to evolve together, we argue here that this is already the case.

The above-mentioned concept of reactive multisensor management requires that sensor control messages based on fusion results can be fed back to the sensors in (simulated) real time. This suggests an architecture where the entire multisensor data acquisition and fusion process is an integrated part of the scenario, in the guise of an acquisition management and information fusion function of a simulated C2 centre. Such an architecture is employed in IFD03.

We view the new demonstrator system as an extensible research and demonstration platform, where new methodological ideas can be realized, evaluated and demonstrated, and where various aspects of increasingly complex network-based information fusion systems can be tested in complex and reasonably realistic scenarios. Whereas our previous information fusion projects have focused on method and algorithm development for various specific problems, in particular clustering, aggregation, and classification of force units and sensor management , the development tools associated with the new platform are intended to support substantial reuse, including evolutionary extension and rewrite, of both software and simulation scenario descriptions.

3.2 Research goals and issues

In line with recent meta-models of multisensor-multitarget fusion, we view Level 2 information fusion as the interpretation of a flow of observations in terms of a model of a physical process in space and time. This process describes the stochastic interaction between an observation system, a complex target system (such as a hierarchically organized enemy unit) and a complex environment. According to this view, what distinguishes Level 2 from Level 1 fusion is mainly the much higher complexity of the target and environment models, involving imperfectly known doctrines which influence the behaviour of the target system in a way that needs to be stochastically modeled.

The purpose of the interpretation process is partly to estimate current and near-future values of a set of possibly unobserved behavioural parameters of the target system, partly to improve the estimates of measured parameter values. In IFD 03, no attempt is made to estimate other doctrinal parameters than force structure. In the not too distant future, however, it may become feasible to estimate a larger set of behavioural parameters, such as for example, our belief in the proposition "the enemy is aware he has been detected".

In information fusion applications based on complex ground scenarios involving interaction between several, partially antagonistic complex systems, scenario-based simulation is often the only methodology available for systematic characterization and analysis of system behaviour. This methodology permits experimentation according to a top-down approach with various methods, configurations, and parameter values, evaluation of the effectiveness and efficiency of algorithms and modeling methods in relation to a reasonably realistic approximation of the final application environment, as well as verification that all problem-relevant components have been included and modelled on an adequate level of resolution. Also, it supports the establishing of a balanced system design, by allowing the discovery and early elimination of vague concepts and unsolved or inadequately treated subproblems, as well as system performance bottlenecks. Design proposals which do not work even in a simplified synthetic environment can be identified and quickly eliminated, while methods which seem to be promising can be selected for a deeper analysis.

The IFD 03 project rests on a small number of basic methodology principles, i.e., cooperation between methods on fusion levels 1, 2, and 4 in the JDL model, a tight coupling between a qualified synthetic environment and models of sensor behaviour, target force behaviour, and communication.

The project focuses on analysis, evaluation, and presentation of new methodology for a collection of important subproblems in automatic information fusion, i.e., ground target and group target tracking, force aggregation, multisensor management, and short term situation prediction.

Successively for various scenarios, in the future we also expect to create by this approach the capability to address various effectiveness issues, which might be generically described as:

- what improvement in effectiveness (measured, perhaps, as increased information quality, or information gain) of various aspects of situation modeling can be expected from specified information fusion methods?
- what improvement in effectiveness can be expected from a network-based defence technology, with and without information fusion methods?
- how do delays and "inertia" of various kinds, arising from, e.g., information transmission or information processing, influence expected improvements in effectiveness?

3.3 Conceptual overview

The FOI project Information fusion in the command and control system of the future network-based defence recently completed the development of a reusable information fusion demonstrator system, the Infofusion demonstrator 03 (IFD03). In IFD03, level 2 information fusion is treated as the interpretation of a flow of observations, realized as a scenario-based simulation of a physical process in space and time. This simulation describes the stochastic interaction between an observation system, a complex target system, in this case a hierarchically organized enemy unit, and a complex environment.

Information is transmitted from simulated sensors to a simulated Command and Control, C2, site. At the C2 site information is fused and interpreted. Some of these interpretations are then used by the C2 site as basis for issuing control messages intended to improve sensor utilization in relation to a predefined surveillance objective. A key component of the demonstrator is the visualizer, which provides a movie-like, interactively controllable multi-screen playback display of a set of parallel views of the prerecorded simulation.

The IFD03 system was used to perform a demonstration in mid-December 2003, based on a simple battalion-level ground force attack scenario.

The development methodology that was partly reused, partly developed by our project in order to facilitate the construction of the demonstrator proved to be very cost-effective although far from problem-free.

The demonstrator implementation is based on three large development environments, the problem solving environment MATLABTMthe simulation framework FLAMESTM and the terrain modelling system TerraVista Pro BuilderTM. In the project, FLAMES and MATLAB were tightly integrated, and FLAMES' new handling of advanced terrain models, generated by TerraVista, was specified and at least partly financed. Finally, the FLAMES software for visualization of simulation results using the new terrain modelling feature was restructured and both functionally and computationally substantially improved.

3.4 Demonstrator objectives

The demonstrator is not a design and certainly not a prototype of a deployable system. To create a prototype, a significant additional R&D effort would be required. Our primary purpose has been, instead, to investigate how IF methods can be combined into a system and work together in the context of that system. We also wanted to create and exercise an effective mechanism whereby information fusion concepts can be communicated to our customers and other interested parties. Finally, we wanted to create a basis for discussions with customers and prospective users about how research in the IF area should be prioritized.

We chose to build our demonstration on the assumption that "Moore´s law", stating roughly that the capacity of computers doubles at least every 24 months, will be valid until 2015, the year when our scenario is presumed to take place. However, since key algorithms can easily be parallelized, the computational performance of the methods employed in the demonstrator should be sufficient for real time use even today, if large (but still reasonably affordable) computer clusters were employed.

3.5 Use cases

The major use cases we had in mind when creating the system were:

(1) performing a demonstration addressing a possibly "infofusion-naive" audience. This is communication, not research, but could be developed into a methodology to present, visualize, and later analyze in detail properties of new components and subsystems,
(2) performing studies and experiments with sensor models, terrain and other environment models, fusion methods, doctrine models, scenario assumptions, etc, in various combinations, to test different hypotheses about possibilities and limitations related to Network-Based Defence (NBD) and information fusion (IF),
(3) developing and testing methodology and models for IF, i.e., specification, development, and test of new concrete methods and fusion concepts. The size and complexity of a demonstrator can be a severe drawback here, at least early in the research and development process, which leads to the question: how could detail studies in a separate test environment best be combined with system tests involving the demonstrator platform? The demonstrator provides at least a partial answer to that question.

Potential advantages from using such a simulation-based R&D process include:

- shorter turn-around time and lower cost for the modelling activity; this can be exploited to create a better dialog with prospective users/customers,
- higher quality through such a dialog and improved opportunities to pre-test a proposed system in synthetic but increasingly realistic and perhaps ultimately dangerous situations,
- improved basis for the estimation of total system construction costs.

3.6 Evolutionary development and its environment

In general terms, evolutionary system design and development may be described as a methodology where large development projects are partitioned into an organized collection of separately agreed subprojects or phases. Each phase is developed according to a predefined design contract, which can and must be operationally verified by a set of "users" representing the "customer" organization. Originally, the main rationale of evolutionary design and development is to facilitate close customer and end-user involvement in the development process. But a similar iterative design and development process is also well suited to the needs of a research group which develops comprehensive software while striving to retain much individual responsibility for design and work planning.This was also our experience in the IFD03 project.

3.7 Combining C and MATLAB

The decision to use MATLAB instead of C or some other language (CommonLisp was a seriously advocated alternative) for developing the FusionNode was taken because we wanted to spend as little time as possible developing and debugging the implementation of our algorithms, and focus our work instead on algorithm design. The fact that most of our group has significantly more experience in the use of MATLAB than of C influenced this choice. The decision was made easier by the availability of the MATLAB Compiler software, which generates C or C++ code from MATLAB code, enabling us to easily integrate FusionNode code into the FLAMES framework. In summary, the decision proved successful, contributing significantly to the productivity of the project.

Using MATLAB had both positive and negative consequences. On the pro side, new ideas may be quickly implemented using MATLAB's rich variety of built-in functions. MATLAB algorithms could often be conveniently debugged by loading input data, previously generated and then saved during execution of the compiled system, into an interactive MATLAB session. Also, test code could very easily be added, such as plotting the input or output of a function.

On the con side, MATLAB provides neither low-level handling of memory allocation nor complete automatic garbage collection. Instead, the MATLAB system handles allocation and deallocation of memory for objects automatically, using a heap mechanism where objects that are no longer used are released. Storage is automatically re-allocated when an object grows. This caused severe memory fragmentation problems. To diagnose and fix such problems, MS WindowsTM functions had to be used to obtain information on memory availability. Ultimately, the cause of these problems may be found in a MATLAB programming style not adjusted to the development requirements of large systems. MATLAB allows preallocation of matrices that will contain a large number of data. This is the style to be preferred when developing large systems in MATLAB.

MATLAB, designed as an interactive environment, will not catch many errors when using the MATLAB Compiler. Even simple things like misspelling a function or variable will cause run-time errors. However, MATLAB Compiler does issue compilation warnings for many errors like these. Thus, MATLAB Compiler messages should be closely watched.

The large size of our terrain database meant that we ran close to the Windows upper limit of 2 GB per process memory size. Using a larger terrain database size would thus not be possible using current technology. Conceivable solutions of this problem include switching to a computer system with 64 bit address space, or changing the terrain database part of FLAMES to use a disk-stored database. In a short term perspective, both approaches seem unrealistic.

3.8 Code versioning and documentation

The CVS (Concurrent Versions System) configuration manager played an essential part in our system development process. While the use of CVS requires considerable discipline from developers (e. g., not committing untested code, writing proper change logs), we would probably not have been able to interface the different parts of the system without using it, or some similar system. We are currently investigating alternatives to CVS for source-code management. We would, for instance, like to be able use MS Visual StudioTM for controlling also the MATLAB Compiler. Visual SourcesafeTM might then be a viable alternative to CVS.

Since the project became quite hectic as the date for the demonstration drew closer, comprehensive system documentation had to postponed. Since several problems would likely have been avoided if a previous research project had properly documented its detailed procedures, we were brusquely reminded that such documentation is indispensable also in small-to-medium-scale computational research work.

4 Scenario and background

The scenario is imagined to take place in May 2015. Tension in the Baltic Sea area has grown gradually over several years and the state of alert of the Swedish defence has been raised. At the outbreak of the war a number of concurrent events occur. Of these, a "trojan horse" enemy landing at the ferry harbour at Kapellskär is judged to constitute the greatest threat. If the enemy is allowed to move further inland towards the town of Norrtälje and occupy the lake passes there, he will be difficult to defeat with available own resources.

When the defending batallion commander has received his action orders he wants to obtain as fast as possible a detailed picture of the enemy's size, composition, and activity in order to be able to judge the enemy's action options and decide his own. The only intelligence sources available at the time of the landing are four Home Guard patrols deployed at strategic points along the enemy advance routes. The battalion's UAV group is ordered to immediately direct two UAVs for reconnaissance above Rådmansö, to obtain as quickly as possible a more detailed picture of the enemy situation. Figure 2 shows the situation at 18:45. The two UAVs directed to Rådmansö have by then delivered a number of reports which have contributed to the rather detailed situation picture. The chief intelligence officer is able to state that the enemy force consists of a mechanized batallion reinforced by antiaircraft and artillery units, advancing along two roads towards Norrtälje. However, since the bridge across Åkeröfjärden was demolished by the Home Guard at 18:30, the enemy advance along the main road has been decisively delayed.

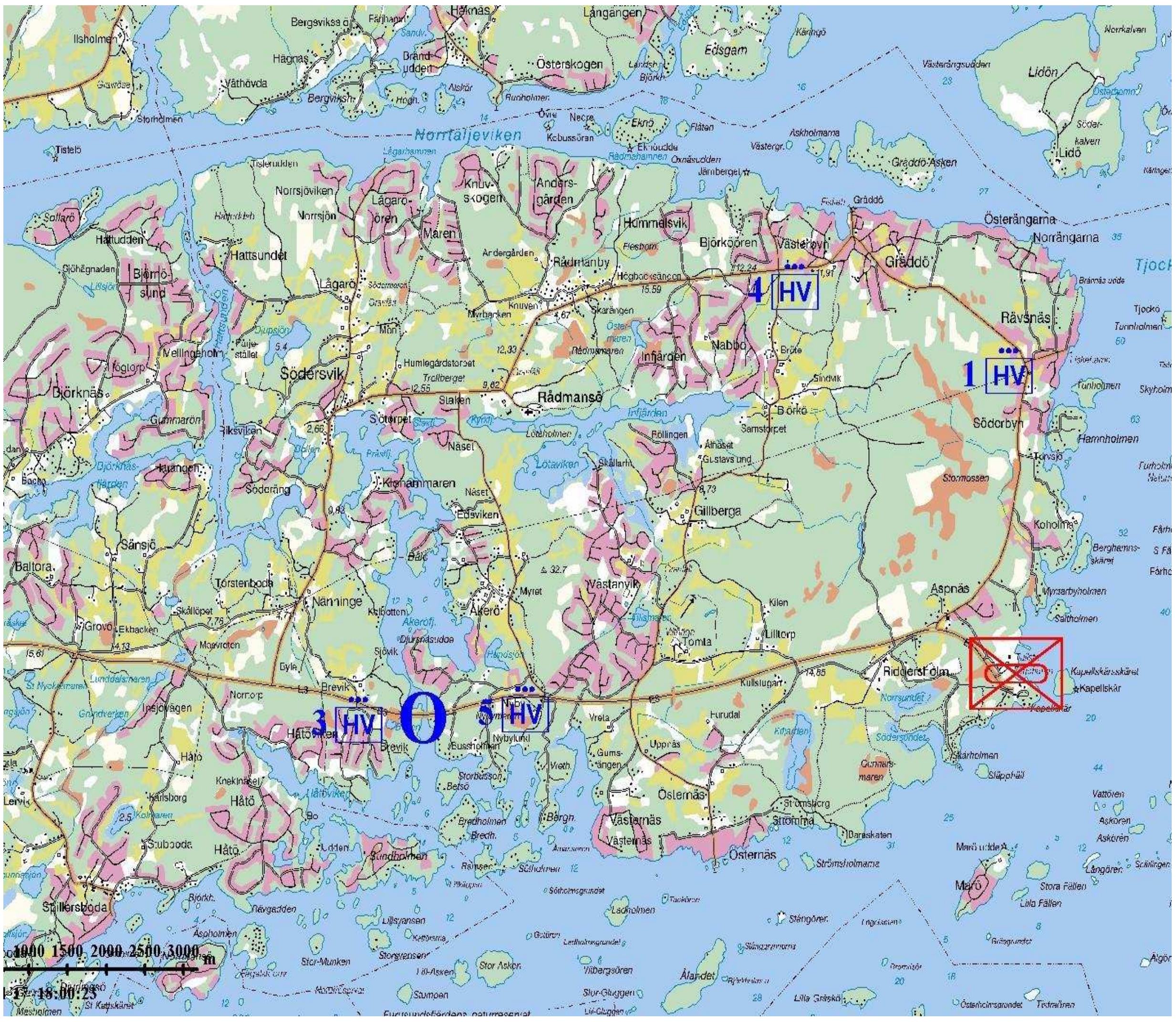


Figure 1 Information collection situation at Rådmansö 18:00.
Four Home Guard (HV) patrols are located at critical points along the enemy's approach route.

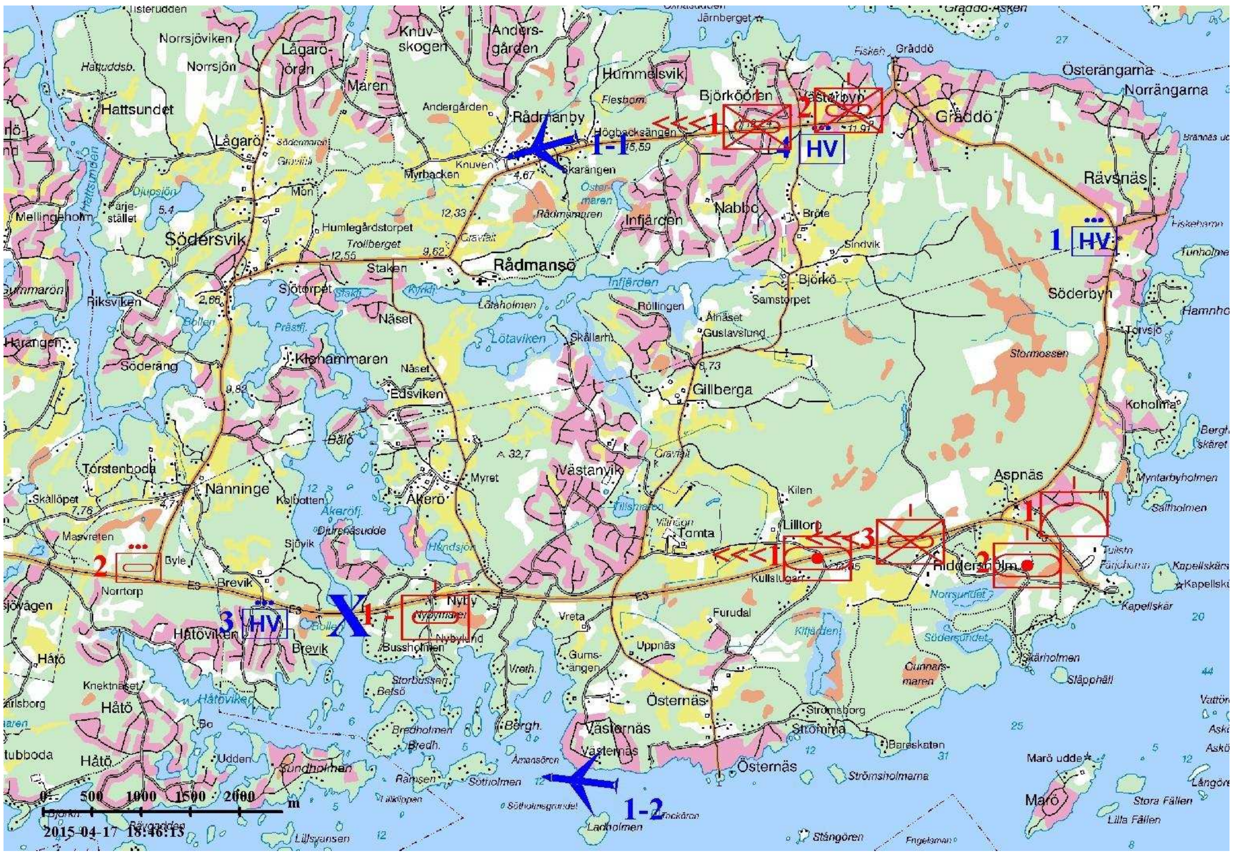


Figure 2 Situation picture at 18:45.

4.1 Organisation, motion and communication models of "red" forces

The model describes the behaviour and motion of enemy ground forces according to their doctrine (i.e., the set of tactical rules that is expected to guide the behaviour of the opponent's army). This includes telecommunication and transportation along a road network of mechanized forces in hostile territory. Ground force objects, which consist of behaviourally connected object models, are able to move along the road network, guided by a limited number of parameters, including destination, road names, starting time, marching speed.

- ground units move autonomously along the road network
- guided by parameters including formation and preferred roads
- a force unit can march in formation, make route choices autonomously or according to prespecified instructions by the user, avoid obstacles by changing route, detect enemies, and replace lost commanders
- using Dijkstra's shortest route algorithm, the model calculates those parts of the route which were not prespecified by the user

5 How to develop Demo03 using CVS

Procedure for getting source to edit:

1. Set all relevant environment variables so that you can use Flames and CVS with CVSROOT = \\Flamma\Infofusion\cvsrepository
2. Create a working directory in you own home area on Flamma or your own computer and copy the contents of the folder "Necessary files to run demonstrator" from \\Flamma\Infofusion to this workdir.
   Explanation of the directory structure:
   a. **make** -- contains MS studio workspace files that should be copied to the Simulation subdirectory.
   b. **bin** -- contains executables in Debug and Release subdirs
   c. **temp** -- contains objects files
   d. **Scenario** -- contains scenario files
   e. **run_dir** -- contains some input files needed for matlab. output files will be stored here.
      NB: Make sure that you COPY the files and don't MOVE them. The directory is write-protected, this means that if you try to copy (left-click and drag) you will get asked if you're sure. Click Cancel and move (right-click and drag, choose move in menu) the files instead. You will have to remove the write-protection for some of your files. The directories run_dir, temp, and bin must be write-enabled; it is probably easiest to write-enable your entire working directory. A quick way to do this is to first write-protect everything, click yes when Windows asks if the change should be applied to all subfolders. Then remove the write-protection and again click yes when Windows asks whether to apply it to subfolders.
3. Start a shell session, cd to your workdir.
4. run the command
   cvs checkout Demo03
   A directory named Demo03 should be created, containing the source code in subdirectories Simulation and FusionNode
5. move the files that are related to the Visual workspace from the Make subdirectory of Necessary files... or your workdir to the Simulation directory. The files that represent the workspace are those with .dsp and .dsw extensions. If you add any c-files to your workspace, you need to update the files in Necessary files...\Make to reflect this. This is done simply by copying the make-files that build a complete demonstrator to Necessary files.... Make sure that you have compiled and tested several scenarios before you do this! **Ask Christian Mårtenson or Pontus Svenson if you are unsure what files need to be copied**.
6. In Simulation, double-click make.dsw to load it into Visual Studio.
7. Select
   Project settings -> Link -> Input
   and add
   ;..\..\bin\Debug;
   to "Additional Library Path". Don't forget to save your workspace after doing this!

Note 031030: you may not need to do this any longer. Note that it may be necessary to change default project to fire instead of forge.

8. Build fire.
9. Start matlab in FusionNode and run compile.m.
10. Build fire again from Visual Studio.
11. Run fire.

When you are ready to commit your changes to the CVS repository, make sure that your changes don't break someone else's code!
Note that not all modified files should be checked in. If you have changed some other person's file in order to debug your code, you shouldn't check that file in!
Note that you can either delete your workdir and do another checkout to continue work later, or you can use cvs update. cvs update is probably better.

Suggestion
Before checking in files, another person should review the changes that you have made. This would make it easier for us to catch bugs and minimize the risk of files containing simple mistakes that cause compile-time errors to be checked in.

5.1 Storing common data files

Some data used for the sensor models is stored in text files. In order to access these files, the following code-excerpt can be used:

```
#include <stdlib.h>
#include <string.h>
#include FMH_USE
#include FVARIABLEMANAGER_USE

FILE *fp;

char *pathptr, buf[128],*fullfilename;
buf = FVariableManagerGetStringValue ("DataFilesPath");
if(buf == NULL) {
        strcpy(buf,"Confusion matrix load error: DataFilesPath
scenario variable not found.");
                FMHAnnounce(0 ,0, buf);
                return(**FELTYP**);
 }
 pathptr=getenv(buf);
 if(pathptr == NULL) {
        strcpy(buf,"Confusion matrix load error:
 DataFilesPath scenario variable contains nonexisting
 environment variable.");
                FMHAnnounce(0 ,0, buf);
                return(**FELTYP**);
 }
 strcpy(buf,pathptr);
 strcat(buf,"\\");
 fullfilename=strcat(buf,filename);
 fp = fopen(absolutepath,"r");
```

The code requires a scenario variable to be set in forge: DataFilesPath should be set to the environment variable FLAMES_Datadirectory. This variable in turn must point to the proper directory.

5.2 Profiling IFD03

In order to find the most time-consuming parts of IFD03, follow this guideline:

Preparation mcc:

- In a terminal window, run ”mbuild –setup”. This creates the file needed for the next step. Ignore any error messages relating to uninstallable add-ins.
- Copy the file compopts.bat from: C:/Documents and Settings/<user name>/Application Data/MathWorks/MATLAB/R13 to the directory containing compile.m (the file that creates fusionnode.dll).
- Edit compopts.bat: append the flag /PROFILE to any of the lines starting with ”SET LINKFLAGS”
- (If you’re using mex-files, do the same for mexopts.bat, for this case it is also necessary to remove the line et POSTLINK_CMDS=del "%OUTDIR%%MEX_NAME%.map")

Preparation fire:

- Rebuild fire with ”Generate debug info”, ”Generate mapfile” and ”Enable profiling” active, and with ”/FIXED:NO” appended to the text in ”Project options”. All options can be found in ”Settings->Link->General” in visual studio.
- Copy fire.map from /temp/obj-debug to /bin/Debug.

Preparation fusionnode.dll:

- Run compile.m with the modified compopts.bat from above in the same directory.
- CD tol /bin/debug and execute in a console window: ”PREP /OM /FT fire.exe fusionnode.dll”. This prepares fire and fusionnode for profiling, creating a number of datafiles.

Run:

- CD to run_dir and execute: ”PROFILE ../bin/Debug/fire.exe <Scenariofile>”.

Analysis:

- CD to /bin/Debug and run:
  - ”PREP /M fire” followed by:

   - ”PLIST /STC fire > outfile”

Statistics can now be found in outfile.

5.3 Environment variables to run fire and flash

In order to be able to run IFD03 or visualize the results, a number of windows environmental variables need to be set. These variables relate to either Flames or Matlab.

The system path needs to be augmented with the directory where the run-time matlab libraries are installed.

In order to run fire, FLAMES_DATADIRECTORY needs to be set to the directory where parameter files used for the sensor modules are stored. Current value is \\Flamma\Infofusion\datafiles , but this could change if you want to test other datafiles.

Flames requires several variables.

- FLAMES_DATABASE should point to the flames database directory. This is the directory information on platforms and icons are stored. Current value is \\Flamma\dfs\DB\Main.
- FLAMES_RELEASE needs to be set so that Flames can find its licence files. Current value is \\Flamma\Flames_installation .
- FLAMES_PROTO can be set to a directory containing a prototype file. This is needed to run forge; most often the prototype file is stored in the same directory as the working copy of forge, so this variable can often be ignored.

In order to run Flash without connections to the FOI network, a dongle needs to be installed in Suttung and an additional licence installed on Kvaser. Such licences can be obtained from Ternion. The file ternion.lic on both computers also needs to be changed, so that it does not refer to Flamma but instead uses either the dongle or the temporary license. In addition, the environment variables that refer to Flamma need to be changed to point to the computer's local installation of Flames. The database directory also need to be copied from Flamma and the FLAMES_DATABASE variable changed accordingly.

6 The IFD03 visualizer

The IFD03 Visualizer is a tool for visualizing the simulation output from Fire. It is a strongly modified version of the Flames original visualizer Flash aimed at illustrating the functionality of the IFD03 Fusionnode. The simulation results can be visualized synchronously in multiple parallell visualizers, making it possible to use many computers and screens simultaneously. New views can easily be created and customized. Currently the visualizer includes the following views:

- Ground truth
- Sensor reports
- Vehicle, platoon and company aggregates
- Particle filter histograms for vehicles
- Matlab view

6.1 How to visualize IFD03

This section describes how to run Flash to show the demo.

First go to the directory containing the fire output files and scenario you wish to show.

The scenario can be run using Forge, see Flames documentation. To run Flash two programs must be run:

Playbackcontrol.exe control the timing of the demonstration. Run it and open the scenario you want to show.

Flash.exe must be run on all computers that participate in the demonstration. After opening it, you must load terrain and connect to the scenario loaded by Playbackcontrol program.

Various views can be opened in Flash. For best effect, use several screens. The 3rd screen Matlab view can be opened on the same computer as Playbackcontrol is run on.

Use the buttons in the Playbackcontrol window to start, stop, pause, and single-step through the scenario.

A set of macro-files are available to run the demonstration automatically using MacroToolworks.

6.2 Program documentation for the IFD03 Visualizer

The IFD03 Visualizing system is originally designed and implemented by Björn Lindström. It consists of four entities, out of which three are executable applications.

- The **database**, handled by a MySQL database manager stores simulation result data to be visualized. Do not confuse this database with the FLAMES database.
- The **Postprocessor** application is responsible for creating tables in the database and for converting and transferring simulation result data into the database. It is a Windows console application, meaning it has no graphical user interface, and is suitable for execution inside batch scripts.
- The **Playback Control** application is responsible for synchronizing the playback of the scenario across the different connected visualizers. The user controls the playback of the scenario from this application. The user can move freely in scenario time and the clients will be updated accordingly. The application works as a server to which the visualizers, clients, can connect.
- The modified **FLASH** application is responsible for the actual visualizing of the data. The different kinds of data are visualized in views and FLASH connects to the playback control as a client and fetches the data to be visualized from the database.

The design is based on the FoRMA/Anabasis Post analysis tool. The views in FLASH are based on the example view code provided by Ternion. For detailed descriptions of the different components and for instructions on how to add new views in FLASH, see ***VisualizerDesign.doc*** in the html-documentation. Output from the IFD03 in two separate forms are used in the visualizer. The matlab view uses data saved in a matlab file, while the other views rely on a SQL database. The PostProcessor must be run on the .data-files written by IFD03 in order to add them to this database. For a description on how the simulation output is logged, see the Log Module in the Fusionnode documentation.

For most of the views only minor modifications have been made. The exception is the "third view", implemented in Matlab. For a description of this, see

6.3 Documentation for Matlab vizualizer module in IFD03

The "third view" is a Matlab vizualization of

- the report stream
- the estimated number of vehicles and units from the aggregation module
- the estimated number of vehicles and units from the tracking module
- the selected path in the sensor management module

During execution of the simulation, the analysis methods save parameters in the files matlabplot_sensor_adaption.mat, matlabplot_track.mat and matlabplot_aggregate.mat. Before visualization the files have to be moved from the Demo03/run_dir directory to the Visualizer/Matlabview directory. The files are during visualization loaded by the "third view" matlab module.

Implementation

The list shows the call structure.

- init_matlabview, calls
- convert_time_to_julian
- draw_matlabview, draws the parameters.

init_matlabview

Loads the reports from the file matlabplot_track.mat generated by the tracking module. Rewites them as strings suitable for display and saves them in another (smaller) .mat file. This function is called by the Visualizer once at the start of a Visualizer session.

convert_time_to_julian

Converts seconds, minutes, hours, days etc. to date and time in a certain string format.
Input: year, month, day, hour, minute, second.
Ooutput: Julian time string.

draw_matlabview

Loads the reports saved in init_matlabview, displays the few latest reports. Loads the estimated number of vehicles and units saved in .mat files by the aggregation and tracking modules, displays the latest minutes of estimated number of vehicles and units. A a certain visualization time, loads the selected UAV track saved in a .mat file by the sensor adaption module during simulation, displays the selection. This function is called by the Visualizer once every 10 seconds.
Input: Visualization time.

7 IFD03 system and terrain

7.1 Introduction

The IFD03 consists of several programs, each of which is documented in subpages. Flames is used to simulate the scenario, while all of the analysis is done in Matlab code that is compiled to C and linked to fire (the generator part of Flames).
Sensor reports are generated by Flames-cognitive-models that send them to a Control module in Flames. This translates the reports into Matlab format and send them to a Matlab function. This function only stores the reports. Analysis is performed whenever the Matlab function Flames_Analyze is called from the Control module in Flames.
Flames_analyze dispatches the report to the various submodules; see the subpage on the fusion node.

The Flames modules and the FusionNode are compiled into the fire program. Running this will produce a number of output datafiles:

- SQL-data files that should be inserted into the SQL data base using the PostProcessor from Visualizer. (*.data-files)
- A matlabplot_aggregate.mat file containing data for the Matlab view in Visualizer.
- A rapport_loggade.mat file containing all reports that have been received by the fusion node.

7.2 Documentation on the Terrain Model in IFD03

The geographical source data used in the Rådmansö scenario comes from the Swedish Land Survey. In order for Flames to import the data it has to be converted to shape-files. For this process the TerraVista Pro Builder is used.

Terrain Models for FLAMES EFO and CFO

Description of the development process

Source data

The source data for this terrain model consisted of conventional off-the-shelf geographic data from the Swedish Land Survey. Source data used in the project come from GSD (Geographical Data of Sweden). The following categories are used:

- Topographic information – elevation map (50 m post spacing)
- Thematic mapping / terrain classification (vector)
- Road networks

Source data preparation

In some cases the source data had to be pre-processed in order for FLAMES to be able to process these correctly. As an example, FLAMES requires road segments to be continuous and tagged with a name, otherwise the line topology function in

FLAMES will fail. This is a major drawback, since we would like to be able to use the source data “as-is”, without the need for pre-processing. In this way, only the major roads will be present in the resulting terrain model, which probably wouldn’t yield satisfactory results or indeed realism.

Terrain database generation tools

We used TerraVista Pro Builder by TerrEx, inc. for the terrain database generation. TerraVista Pro Builder is the software of choice for creating terrain databases for FLAMES CFO and EFO. TerraVista can handle a large number of source data types and is able to generate output in a variety of formats.

Tool setup

TerraVista had to be extended to include specific processing passes for GSD feature data. This was accomplished using the scripting functionality of TerraVista.

Gaming Area

East 1687500
West 1642500
North 6635000
South 6615000
Block Size 5000m
Tile Size 5000m

Output polygon attribution

For FLAMES to consume TerraVista polygon output, the polygons needed to be attributed with specific fields as described below.

- FLAMES_COD
    - Specifies which feature class the terrain area belongs to, eg. ”landregion” or ”road”
- FLAMES_RGB
    - Colour for visualisation in FORGE (in the format ”xRRGGBB”)
- FLAMES_NAM
    - Feature name, eg. road name..
- FLAMES_SID
    - Main driving side of road
- FLAMES_WD1
    - Inner width of road
- FLAMES_WD2
    - Outer width of road

These fields are added to the output process using TerraVista’s scripting functionality.

Feature types and processing passes

The table below shows how the different feature classes in GSD are processed in TerraVista.

LM-kod	Name	Comment
1	Vattenyta (Body of Water)	Areal
2	Barr/blandskog (Coniferous/Mixed forest)	Areal, canopy
3	-	
4	Åker (Cropland)	Areal
5	Annan öppen mark (Grassland)	Areal
6	Hygge (Clear-cut area)	Areal
7	Fruktodling (Fruit Farm)	Areal
8	Kalfjäll (Bare mountain Region)	N/A
9	-	
10	Fjällbjörkskog (Mountain Birch)	Areal, canopy
11	-	
12	Sluten bebyggelse (Urban area)	Areal, canopy
13	Hög bebyggelse (High rise urban area)	Areal, canopy
14	Låg bebyggelse (Low suburban area)	Areal, canopy
15	Industriområde (Industrial Area)	Areal, canopy
16	Fritidsbebyggelse (Recreational area)	Areal, canopy
17	Annan öppen mark utan skogskontur (Open land)	Areal
18	Vattenyta med diffus strandlinje (Water surface)	Areal
19	Lövskog (Hardwood forest)	Areal, canopy
20	Ej karterat område (Uncharted area)	
>5000	Vägar (Roads)	Major named roads
>700	Byggnader (Buildings)	Points, specific models for each building type
>300	Vattendrag ((Water)	

Output

Several different output formats were generated for use in FLAMES and for 3D visualization. TerraVista ensures that all output is correlated. The output formats are listed below.

- Vector files (ESRI Shapefiles)
- 2D orthophoto (grayscale)
- Topographic model (TIN, ESRI Shapefile)
- Terrain model for real-time visualization (OpenFlight)

Coordinates

FLAMES uses WGS84

Problems and issues

Several issues were encountered during the development process, leading to support notifications and testing of unofficial versions of TerraVista.

Empty (non-attributed) polygons (1)

By default, TerraVista creates a 10m buffer around eg. water and forests. The buffer polygons will be void of attribution and will use the standard material as defined in Terrain Parameters. This is undesirable and has been disabled in the affected processing passes.

Empty (non-attributed) polygons (2)

On rare occasions, the generated polygons can be void of attribution, even though every GSD feature is processed in TerraVista. This is mostly evident in forest canopy areas. This issue is yet to be resolved, and it has been reported to TerrEx as a serious issue. Some of this can be worked around using different settings for "polygons/block".

ASL colouring

If the colouring of features is based on ASL (Above Sea Level), small islands are likely to disappear, since they only contain polygon nodes on their outer boundary. Since the boundary is set to be at Sea Level (0m), the island polygons will be flat (with no z coordinate above zero) and when coloured based on ASL they will be the same colour as the surrounding water.

Large datasets

The Shapefile output for the area which is consumed by FLAMES exceeds 2.0 Gb. This takes a long time for FLAMES to consume. Most of the attribution in the polygons is internal TerraVista-specific information which is not need for FLAMES. It has been suggested to TerrEx that they allow the user to specify which attributes are propagated down to the polygon output.

Erroneous triangulation

On rare occasions, TerraVista’s triangulation engine will fail. This is probably due to the fact that it is forced to create extremely long and thin triangles. This can occur eg. when a small building footprint is integrated into a very large flat terrain area, causing the side of the building to be a triangle edge several magnitudes smaller than the other edges. One quick way of eliminating this is not to generate footprints for building models that are too small.

8 IFD03 simulator

The simulator part of IFD03 consists of C code and scenario files used with Flames. Flames consists of three different parts. Forge is where a scenario is created, fire is used to run a scenario, and flash to visualise it. We use a custom-designed Flash, described in section 6. The models described in this section are linked into fire.

8.1 Flames Fusionnode Model

Overview

The FOIIFFusionnode is a Flames Cognitive Model that serves as an interface between Flames and the internal Matlab functions in the FusionNode. The functions of the model can be divided into the following categories:

1. **Analysis**
   When executing a scenario simulation in Fire the Flames Kernel serves as the simulation engine. At certain time intervals the Kernel calls the control module to determine if some or all analysis methods are to be performed. The function called is FOIIFFNAnalyze and the frequency of the calls is set in the Flames Prototype for FOIIFFNAnalyze.
2. **Message processing**
   The Control Module continuously recieves reports from the sensors in Flames. The reports are processed in the FOIIFFusionnode model and translated to a Matlab representation. Two types of reports are currently supported and shipped to the Matlab module, target reports and reports on sensor status. For a description on how to send reports to the Fusionnode, see **Sending reports in Flames**.
3. **Requesting Sensor Status**
   The Fusionnode sometimes needs to ask a sensor for its current status. To handle this it is possible to send a message to a specifik sensor (each sensor has a unique ID). The sensor (hopefully) answers by sending a sensor status report.
4. **Sensor control**
   The Fusionnode has a **Sensor Management function**. The result of the sensor management is a number (1-4) of the optimal path planned for a UAV. Each number corresponds to a specifik airspace (plan1-plan4) predefined in the scenario. The FOIIFFusionnode has a function that manages a devoted UAV (named "UAV_controlled") to fly the chosen path. It also controls the dropping of a Ground Sensor Network ("GroundSensorNetwork_dropped") on a spot hardcoded for each path.
5. **Utility functions**
   The FOIIFFusionnode model includes a bunch of utility functions. Some are called from Matlab to access information in Flames, such as the terrain or unit positions. The FOIIFFusionnode model also includes a logging function for time and truth data, see **Log Module**.

Example Script

```
PERFORM STARTUP USING FOIIFFusionNodeStartup;
PERFORM ANALYZE USING FOIIFFusionNodeAnalyze;
PERFORM SHUTDOWN USING FOIIFFusionNodeShutdown;
PERFORM CONTROL_SENSOR USING FOIIFFusionNodeControlSensor;

TRANSMIT FOIIFMsgRequestSensorStatus AT REQUEST_SENSOR_STATUS;

RECEIVE FOIIFMsgREPORT USING FOIIFFusionNodeProcessREPORT;
RECEIVE FOIIFMsgSensorStatus USING
FOIIFFusionNodeProcessSensorStatus;
```

Cognitive Methods

- FOIIFFusionNodeStartup
- FOIIFFusionNodeShutdown
- FOIIFFusionNodeAnalyze
- FOIIFFusionNodeProcessREPORT
- FOIIFFusionNodeProcessSensorStatus
- FOIIFFusionNodeControlSensor

Methods called from Matlab

- **FOIIFFNRequestSensorStatus**. Sends a request message for a sensor status report to a specifik sensor, given its ID as input. The function is reached from Matlab through Flames_Request_Sensor_Status.
- **FOIIFFNExecuteSensorPlan**. The FOIIFFNExecuteSensorPlan takes an integer (1-4) as input. The number corresponds to a specifik airspace (plan1-plan4) predefined in the scenario, and FOIIFFNExecuteSensorPlan tells an UAV ("UAV_controlled") to fly the chosen path. It then initiates CONTROL_SENSOR so that FOIIFFusionNodeControlSensor can handle the dropping of a Ground Sensor Network ("GroundSensorNetwork_dropped").

Utility functions

- **foiiffn_mterrain.c** contains two terrain utility functions used by the Terrain Module:
  - **GetElevation** is reached from Matlab through flames_get_elevation
  - **GetStringAttribute** is reached through flames_get_terrain_type.
- **getBluePositions** returns an mxArray with the positions of all blue units. It is reached from Matlab through flames_get_blue_positions
- **LogTimeAndTruthData** Logs the scenario time to file. Log unit truth data for all active, alive, and present units in the scenario at the current sim time.

FOIIFFusionNodeStartup

Initializes the FOIIFFusionNode object. Must be executed before any other method of this class. Initializes the matlab fusionnode dll and calls matlab init file Flames_Startup.

- **Intended Function**
  STARTUP
- **Execution Mode**
  SINGLE
- **Inputs**
  None
- **Parameters**
  None
- **Function Initiation Points**
  None
- **Message Generation Points**
  None

FOIIFFusionNodeShutdown

Calls Flames_Shutdown.

- **Intended Function**
  SHUTDOWN
- **Execution Mode**
  SINGLE
- **Inputs**
  None
- **Parameters**
  None
- **Function Initiation Points**
  None
- **Message Generation Points**
  None

FOIIFFusionNodeAnalyze

Calls the matlab fusionnode function Flames_Analyze. The function is executed continuously with timestep given in the FLAMES prototype. It also calls the utility function LogTimeAndTruthData.

- **Intended Function**
  ANALYZE
- **Execution Mode**
  SINGLE
- **Inputs**
  None
- **Parameters**
  None
- **Function Initiation Points**
  None
- **Message Generation Points**
  None

FOIIFFusionNodeProcessREPORT

Processes a target report message. The report is transformed into a Matlab format and shipped to Flames_Report.

- **Message**
  FOIIFMsgREPORT
- **Execution Mode**
  SINGLE
- **Inputs**
  The message attributes currently recognized by the Fusionnode (FOIIFFusionNodeProcessReport) are listed in the table below. Note that for the target position error only one of TARGET_POS_ERR and TARGET_POS_ERR_NSWEROT should be used.

Variable Name	Datatype	Description
SENSOR_POS_LLA	FVectorType	Sensor position in (lat,lon,alt)
SENSOR_POS_ECR	FVectorType	Sensor position in ECR Coordinates
SENSOR_ID	FEquipmentIDType	The unique Equipment ID of the sensor.
SENSOR_TYPE	char *	Name of the sensor type.
SENSOR_CONTINUOUS_TRACKING	FIntegerType	Set to "1" it signals that the target is being tracked and that a report previously has been sent. Currently not used.
TARGET_CORRECT_NAME	char *	The true target unit name
NBR_OF_TARGET_FOCALS	FIntegerType	The number of arguments in TARGET_FOCALS and TARGET_PROBMASS
TARGET_FOCALS	FMALObj	MAL with strings, each representing a subset of possible targets. Use "," as delimiter.

		MARGName arbitrary.
TARGET_PROBMASS	FMALObj	MAL with the probability masses corresponding to each hypothesis in TARGET_FOCALS. To connect the correct prob. mass with each hypothesis, use the same (arbitrary) MARGName for both.
TARGET_POS_LLA	FVectorType	Target position in (lat,lon,alt).
TARGET_POS_ECR	FVectorType	Target position in ECR Coordinates.
TARGET_POS_ERR	FRealType	Standard deviation of the Gaussian target positional error.
TARGET_POS_ERR_NSWEROT	FVectorType	Elliptic error. X - stddev north-south, Y - stddev east-west, Z - clockwise rotation of ellipse.
TARGET_SPEED	FRealType	Estimated target speed (m/s).
TARGET_HEADING	FRealType	Estimated target heading (rad).
TARGET_SPEED_ERR	FRealType	Standard deviation of the Gaussian target speed error.
TARGET_HEADING_ERR	FRealType	Standard deviation of the Gaussian target heading error.
DETECTION_TIME	FJulianType	Time of detection.

- **Parameters**
  None
- **Function Initiation Points**
  None
- **Message Generation Points**
  None

FOIIFFusionNodeProcessSensorStatus

Processes a sensor status message. The message is transformed into a Matlab format and shipped to Flames_Sensor_Status.

- **Message**
  FOIIFMsgSensorStatus
- **Execution Mode**
  SINGLE
- **Inputs**
  The message attributes currently recognized by the Fusionnode (FOIIFFusionNodeProcessSensorStatus) are listed in the table below. Note that for the sensor coverage either COVERAGE_POSITION + COVERAGE_RANGE or NUMBER_OF_VERTICES + VERTEXn should be used.

Variable Name	Datatype	Description
SENSOR_ID	FEquipmentIDType	The unique Equipment ID of the sensor.
SENSOR_TYPE	char *	Name of the sensor type.
STATUS_TIME	FJulianType	Time of the status report.
COVERAGE_POSITION	FPositionType	Centerpoint of the current sensor coverage (ECR-coordinates).
COVERAGE_RANGE	FRealType	Radius of a circular coverage.
NUMBER_OF_VERTICES	FIntegerType	Number of vertices in a polygon describing a coverage area.
VERTEXn	FCoordinateType	The coordinates of the coverage area vertices. n ranges from 1 to NUMBER_OF VERTICES.

- **Parameters**
  None
- **Function Initiation Points**
  None
- **Message Generation Points**
  None

FOIIFFusionNodeControlSensor

The CONTROL_SENSOR is initialized from FOIIFFusionNodeExecuteSensorPlan, which is called from matlab. It then continuously watches the position of the UAV_controlled. When the UAV flies over the predefined drop zone, the FOIIFFusionNodeControlSensor launches GroundSensorNetwork_droppped.

- **Intended Function**
  CONTROL_SENSOR
- **Execution Mode**
  CONTINUOUS until the Ground Sensor Network is dropped.
- **Inputs**
  None
- **Parameters**
  None
- **Function Initiation Points**
  None
- **Message Generation Points**
  None

8.2 Instructions on Sending Reports in IFD03

Sending reports in Flames

There are two types of reports a sensor can send to the fusionnode. One is the target report containing information on a detected target, such as target position and classification. The other is the report of sensor status information, including information on current sensor operability and coverage. The procedure to generate both message types are identical, except for the included message attributes. For lists of recognized attributes, see the documentation on the FOIIFFusionNode.

To send a report from a sensor model to the fusionnode you have to create two MALs (Model Argument Lists). In the first, which you name "DATAMAL", you put all attributes of the report that you want to include. Then you add the DATAMAL to a second MAL, which you name "MSGMAL". This is the MAL that you use as argument when generating the message. The reason for using two MALs is to make it possible to add new report attributes without changing the message prototype. (In the prototype for a message you have to specify its exact constituents, but with this method it will always only contain a single MAL, the DATAMAL).

Example code for a cognitive model sending (target) reports:

1. DATAMAL = FMALCreate();
2. FMALAddFMAL(DATAMAL,"TARGET_FOCALS",FOCALMAL);
3. FMALAddFMAL(DATAMAL,"TARGET_PROBMASS",PROBMASSMAL)
   ;
4. FMALAddFInteger(DATAMAL,"NBR_OF_TARGET_FOCALS",2);
5. etc.
6. MSGMAL = FMALCreate();

7. FMALAddFMAL(MSGMAL,"MSGDATA",DATAMAL);
8. FBEGenerateMessage ("REPORT",MSGMAL ,0,0,0);

NOTE: If generating the message from an equipment model FBEGenerateMessageFromEquipment must be used instead of FBEGenerateMessage, otherwise a runtime error will occurr!

Example script code in **F**orge

(Fusionnode)

- PERFORM STARTUP USING FOIIFFusionNodeStartup;
- RECEIVE FOIIFMsgREPORT USING FOIIFFusionNodeProcessREPORT;
- RECEIVE FOIIFMsgSensorStatus USING FOIIFFusionNodeProcessSensorStatus;

(Sensor unit)

- TRANSMIT FOIIFMsgREPORT AT REPORT;
- TRANSMIT FOIIFMsgSensorStatus AT SENSOR_STATUS;

8.3 Sensor Manager Model

Overview

The Sensor Manager Model serves as an interface between a unit and its sensors. When a sensor makes a detection the Sensor Manager will transmit the detection data as a FOIIFMsgREPORT message. The Sensor Manager also processes sensor status requests. Given an EquipmentID it queries the corresponding sensor for information and transmits it as a FOIIFMsgSensorStatus message.

Example Script

```
PERFORM STARTUP USING FOIIFSensorManagerStartup;
PERFORM DETECT USING FOIIFSensorManagerReport;

TRANSMIT FOIIFMsgREPORT AT REPORT LOG;
TRANSMIT FOIIFMsgSensorStatus AT SENSOR_STATUS;

RECEIVE FOIIFMsgRequestSensorStatus
               USING FOIIFSensorManagerProcessRequestSensorStatus
LOG;
```

Cogni**t**ive Methods

- FOIIFSensorManagerStartup
- FOIIFSensorManagerReport
- FOIIFSensorManagerProcessSensorRequest

FOIIFSensorManagerStartup

FOIIFSensorManagerStartup is initiated by the kernel at STARTUP and sends a status report to the fusionnode for each sensor attached to the unit. The status info is queried from each sensor and passed on to the fusionnode with messages.

- **Intended Function**
  STARTUP
- **Execution Mode**
  SINGLE
- **Inputs**
  None
- **Parameters**
  None
- **Function Initiation Points**
  None
- **Message Generation Points**
  SENSOR_STATUS

FOIIFSensorManagerReport

FOIIFSensorManagerReport takes a MAL as input from a sensor and sends it as a message to the fusionnode.

- **Intended Function**
  DETECT
- **Execution Mode**
  SINGLE
- **Inputs**
  MSGDATA - MAL with the data to send
- **Parameters**
  None
- **Function Initiation Points**
  None
- **Message Generation Points**
  REPORT

FOIIFSensorManagerProcessRequestSensorStatus

Processes a RequestSensorStatus Message for the FOIIFSensorManager cognitive model. Queries sensor SENSOR_ID for status info and passes it to the fusionnode in a message.

- **Message**
  FOIIFMsgRequestSensorStatus
- **Execution Mode**
  SINGLE
- **Inputs**
  SENSOR_ID - The ID of the sensor to ask for sensor status

- **Parameters**
  None
- **Function Initiation Points**
  None
- **Message Generation Points**
  SENSOR_STATUS

8.4 Message models

8.4.1 FOIIFMsgREPORT Model

The FOIIFMsgREPORT is used to send detection data from a sensor to the Fusionnode.

Input

- MSGDATA (FMAL) In this you can put anything at will. It is up to the sender and receiver to agree on content.

Content

- MsgData (FMALObj) A clone of the input MAL.

8.4.2 FOIIFMsgSensorStatus Model

The FOIIFMsgSensorStatus is used to send sensor status data from a sensor to the Fusionnode.

Input

- MSGDATA (FMAL) In this you can put anything at will. It is up to the sender and receiver to agree on content.

Content

- MsgData (FMALObj) A clone of the input MAL.

8.4.3 FOIIFMsgRequestSensorStatus Model

The FOIIFMsgRequestSensorStatus is sent by the Fusionnode to ask a unit to return sensor status information on a specific sensor.

Input

- SENSOR_ID (FEquipmentID) The ID of the sensor to ask for status info.

Content

- SensorID (FEquipmentIDType) The ID of the sensor to ask for status info.

8.4.4 FOIIFMsgComIntReveal Message Model

The FOIIFMsgComIntReveal is sent by the process model FOIIFComIntTlkInit, and the FOIIFComIntTlk loaded by units that should be able to reveal itself by initiating, or answering on transmissions. This message is received by other similar units (judging themselves if they should answer with the same message type) by including RECEIVE FOIIFMsgComIntReveal USING FOIIFComIntProcessTalk; in the unit script. The message is also intercepted by also by ComInt intercepting units that have included EMPLOY COMDEVICE COMINTInterceptor; (the intercepting ComDevice) and also RECEIVE FOIIFMsgComIntReveal USING FOIIFComIntProcessReveal; in their unit scripts.

Input

• TalkNumber	(FInteger)
• Receiver	(FUnitID)
• AnswProb	(FReal)

Content

• TalkNumber (FIntegerType) Number of message in a potential message sequence. The unit initiating a conversation sets this to one (1) in his message, then it is incremented in each message generated in the conversation; **Dialogue** (ping-pong) or **broadcast type** conversation

• Receiver (FUnitIDType) Set to NULL if there should be no unique receiver (this is broadcast mode to next lower subordinates, they are resolved from the Unit name of the sender), or to a unique receiver Unit whose name is given in this parameter

• AnswProb (FRealType) If < 1.0 , this is the probability for an answer. In Dialogue mode it can be >= 1, in which case it marks the number of messages to be exchanged.

8.4.5 FOIIFMsgComIntFuse Message Model

The FOIIFMsgComIntFuse is sent within the process model FOIIFComIntReveal by the intercepting units to the fuser unit which push:es the message onto a list for periodical processing.

Input

• Footpoint	(FCoordinate)
• Bearing	(FReal)

- BearingErro**r** (FReal)
- RevealedUnit (FUnitID)
- RevealedTime (FJulian)

Content

- Footpoint (FCoordinateType) Lat/Lon footpoint of intercepting unit
- Bearing (FRealType) Estimated Bearing in radians clockwise from north direction towards intercepted emitter
- BearingError (FRealType) Estimated Bearing Error in radians
- RevealedUnit (FUnitIDType) ID of Unit carrying intercepted emitter
- RevealedTime (FJulianType) Time for interception of emitter

8.5 Object hierarchy description

Overview

The *hierarchy file*, which is common for a whole Flames scenario, describes the hierarchical structure of the object identities that carries *FOIIF… type signatures*. These all depend on this hierarchy definition. Together with dataprocessors of type *FOIIFIDlibrary,* this forms an environment where sensors detecting these signatures can reason about the identity of objects carrying the signatures. How well an identification is made depends on the signal level required in the signature set-up. The signal level is the “energy”strength (light in the case of a video sensor, vibrations in the case of a seismic/acoustic sensor) that discerns the object from its background and, hence, make it detectable. If the signal level from an object (say, an T-80U tank) is not high enough for an identification of a T-80U, it might be high enough to imply the presence of a tank (the one step less detailed identity of the T-80U visual signature definition). The precense of a tank means that the object can be an Abrams, Centurion, T-80, T-72, Leopard, Stridsvagn-S,… The list is in theory an enumeration of all existing tank models ever built. In practice, the list only consists of the tanks that a real sensor should be able to identify, given enough signal strength.

The hierarchy definition should be stored as a text document in a directory pointed to by an environment variable whose name is stored in the Flames Scenario Variable (found in Forge under Scenario->Variables…) “DataFilesPath”. For instance, if DataFilesPath is set to the text string “FLAMES_Datadirectory” (must be set to be valid for the first run pass), and the operating system environment variable FLAMES_Datadirctory is set to point on \\Kvaser\Infofusion\Datafiles, this directory must contain the text file with the hierarchy definition. The name of the hierarchy file is given in another Flames Scenario Variable with name “IDHierarchyFile”.

(NOTE: see html-documentation for exact contents of these files.)

8.6 ID Library Dataprocessor Model

Overview

The confusion matrix is not a flames model, but a stand-alone datafile that has to be present for the classification algorithm to work properly. The reason is that this algorithm uses information from the signature carried by an object to determine if the object is detectable and, if so, at which level ("T-80", a " tank", a "tracked vehicle", or simply a "vehicle") it should be reported. The confusion matrix would not be needed if the classification algorithm was content with this. But it is not. There is also a probability (after the algorithm has decided at which level it should report, let's say it is "tracked vehicle") to do a complete mistake, and "jump over" to another object hierarchy and, in this case, report a "wheeled vehicle". The probabilities (confusion or mixing probabilities) are stored in the confusion matrix. Its functionality is the following: If the detection algorithm decides that it has enough signal from the object to say that it has detected a, lets say, T80 tank. But in reality, a classification can always be wrong. The algorithm then asks the identification library of type IDlibrary to check if there can be any misclassification. The IDlibrary enters the row in its matrix (see below) which has the same name as the detected Object class in the signature table, in this case T-80. The matrix elements in this row equal the probability for confusion with the Object class in the top of corresponding column, so that object name will be reported instead. This is the case at "full confusion". The degree of confusion is actually depending linearly on a random fuzziness parameter F (uniform between 0 and 1). F is 1 if the chosen classification was infinitely close to be one step less precise. F is 0 if the chosen classification was infinitely close to be one step more precise. F varies linearly in between. If F=1 the confusion follows the values of the confusion matrix ("full confusion"). If F=0 there is no confusion at all. The confusion is now done according to this scheme, and the resulting object class name is expanded (if not itself a singleton) into a text string containing its comma-delimited singletons, and returned as the focal element to which the sensor wants to assign probability mass. The value of the probability mass assigned to is computed as follows: If F=0, the probability mass is 1.0. If F=1, the probability mass is the same as that for the diagonal element (i.e. correct identification) of the confusion matrix. The probability density varies linearly in between. Finally before reporting, the probability density is "smeared" within a Gauss bell with the width entered in the GUI. This can of course be set to zero to inhibit this effect.

The confusion matrix should be stored as a tab-delimited text document in a directory pointed to by an environment variable whose name is stored in the Flames Scenario Variable (found in Forge under Scenario->Variables…) "DataFilesPath". For instance, if DataFilesPath is set to the text string "FLAMES_Datadirectory" (must be set to be valid for the first run pass), and a operating system environment variable FLAMES_Datadirctory is set to point on \\Kvaser\Infofusion\Datafiles, this directory must contain the text file with the confusion matrix.

To simplify the build of a confusion matrix, it can be built in an excel document and, upon completion and storage as an excel worksheet, also be stored as a tab-delimited text document in excel (File->Save as…->Text (Tab delimited)).

Parameters

Name	Unit	Description
Library filename	**-**	**The name of the confusion matrix tab-delimited text file used by this dataprocessor**
Stand dev of reported prob masses	**-**	**Standard deviation smearing of reported probability mass.**

Button “Check file” can be pressed to get an immediate check of the consistency of the file confusion matrix in the file, so this can be done directly from Forge, and not only during dataprocessor load of the FOIIFIDlibraryin Fire.

For an example of a confusion matrix, see HTML documentation.

8.7 Signature models

8.7.1 Visual signature signature Model

Overview

FOIIFVisualSignature is a signature description class for visual and **IR** relevant signatures.

Parameters

Name	Units	Description	Keyword
Apparent length	meters	The length (the longest dimension) of the object to carry this signature	-
Apparent width	meters	The width (the intermediate sized dimension) of the object	-
Apparent height	meters	The height (the smallest dimension) of the object	-

The lower panel with columns denoted “Object class” and “Signal” is used as a hierarchical literal description of the object carrying this signature, together with the apparent strength of each class description. This latter value (“Signal”) can conceptually be interpreted as an estimate of how “easy” it is for a trained human, or (rather) a trained machine to recognize an object to be of the corresponding type **“**Object class”. More precisely, “Signal” can, relying on the Johnson criterion, be interpreted as the reciprocal of the number of double bars to be resolved over the object’s minor projected dimension to be recognized as belonging to “Object class”.

This table should be ordered, that is, with the most specific description at the top (carrying the lowest value of “Signal”), and less exact (more general) description of the object (with corresponding higher signal values) when traversing the table downwards. The names of the objects must be the same as those described in the confusion matrix which is a part of the IDlibrary dataprocessor. The most specific object at the top of the table must be the physical object itself that carries the signature.

Example:

Object class	Signal
T-80U	0.2
Tank	0.5
Tracked Vehicle	0.8
Vehicle	1.5

Here, “T-80U” is the highest resolution of an object that the sensor measuring this type of signature can discern. T-80U is a (singleton) subset of “Tank”, which could also contain vehicles as the T-72, Abrams, and Leopard tank types. These tanks should then have signatures of their own, with the name (T-72, Abrams or Leopard) in the first row of the table, with the corresponding signal strength. The rest of the table should have the same names (Tank, Tracked Vehicle, Vehicle).

8.7.2 Acoustic Signature Model

The FOIIFAcoustic Signature contains a list of signature attributes ('Object class') with corresponding signal strengths ('Signal'). To be recognized by a sensor the attribute must be contained in the library of the sensor's dataprocessor. The top attribute must always represent a single platform. The following attributes should be names that in the dataprocessor library represent sets of platforms. These sets must always include the elements of the preceding attribute.

Example:

Object Class	Signal
T-80U	0.00
Bandfordon	1.50
Fordon	2.20

8.8 Image Sensor Equipment Model

Overview

This sensor is designed to be used as a downwards-looking video sensor attached to an airplane or a UAV. It is a development of the Flames FQSGeometricSensor.

Parameters

Name	Units	Description	Keyword
Library ID Data Processor	-	Name of a Data Processor of the FOIIFIDlibrary class to perform target identification processing using a signature description of the FOIIFVisualSignature class	-
Max Range	meters	Maximum range within which a detection can occur	-
Azimuth FOV	0 to 360 degrees	Maximum angle of azimuth coverage for a rectangular FOV	-
Elevation FOV	0 to 360 degrees	Maximum angle of elevation coverage for a rectangular FOV	-
Pointing Mode	-	Specifies whether the sensor will be pointed relative to the frame of the platform it is placed on (Relative) or fixed in relation to the earth (Absolute).	-
Pointing Azimuth	-180 to 180 degrees	Azimuth of the boresight of the sensor	-
Pointing Elevation	-90 to 90 degrees	Elevation of the boresight of the sensor	-
Detect Mode	-	If set to “Normal”, detection tests are only done at each scenario time sample increment. If set to “Dense”, a check is done whether the Scan Period is faster than the scenario increment rate. If so, that faster detection scheme is used in that respect that the unit carrying this sensor is moved back along its negative velocity vector to the position where it should have been located at one scan period after the previous scenario time sample. Then, stepwise, the carrying unit is moved forward, and detection tests are done for each next scan time. This scheme stops before next scan time will occur in the future. Units checked for detection are not moved back in this manner; they are assumed ground units moving slowly. Dense mode could be applied when the sensor is attached to an air plane, and being combined with Sweeping Mode “Sweeping” to simulate a fast back-	-

		and-forth side sweeping down-looking reconnaissance sensor.	
Sweeping Mode	-	If set to “Fixed, staring”, the boresight of the sensor is fixed along the settings of Azimuth and Elevation FOV. If set to “Sweeping”, the boresight sweeps back and forth in its azimuth gimbal according to azimuth = Azimuth FOV +/- Half Sweep Angle * SIN(sweepphase).	-
Half Sweep Angle	degrees	If “Detect Mode” is set to “Dense”: One half of the total Sweep Angle of sensor azimuth gimbal. Note that the Elevation gimbal is closer to the coordinate system of the unit to which the sensor is attached. Thus the sweeping will be along a tilted plane.	-
Number of scans per sweep	-	If “Detect Mode” is set to “Dense”: Number of detection tests in the FOV per sensor sweep period. This is the way to set set the sweep period.	-
Scan Period (Sec) / Matched Mode	sec	Number of detection tests in the FOV per second. If the checkbox “Matched mode” is checked, the sensor matches the detection tests to the speed of the unit to which it is attached, so it tries to take “photos” of regions of the ground as close to each other as possible, with minimum or zero overlap. This option can be used (and should only be used) for cases where the sensor is looking down from an aerial vehicle.	-
Detection threshold	pixels	Targets whose images subtend less than this amount of pixels on the “CCD plate” of the detector will not be detected	-
No of pixels in Az FOV	-	Number of pixels along the Azimuth (horizontal) direction of the “CCD plate”. Number of pixels along the Elevation (vertical) direction is computed by multiplying this number with the ratio ”Elevation FOV” / “Azimuth FOV”.	-
Used aver wavel.	micrometers	Used wavelength for detection. This number is currently only used for	-

		estimating the Rayleigh resolution criterion.	
Aperture (objective) diameter	meters	Diameter of the objective. This number is currently only used for estimating the Rayleigh resolution criterion.	-
Az Err	degrees	Gaussian random error of measurement of azimuth direction to target	-
Az Err Drift	degrees	Systematic error of measurement of azimuth direction to target	-
Elev Err	degrees	Gaussian random error of measurement of elevation direction to target	-
Elev Err Drift	degrees	Systematic error of measurement of elevation direction to target	-
N-S Err	meters	Gaussian random error of position of carrying unit in North – South direction	-
N-S Err Drift	meters	Systematic error (positive is towards north) of position of carrying unit in North – South direction	-
E.W Err	meters	Gaussian random error of position of carrying unit in East – West direction	-
E-W Err Drift	meters	Systematic error (positive is towards west) of position of carrying unit in East - West direction	-
Alt Err	meters	Gaussian random error of altitude of carrying unit	-
Alt Err Drift	meters	Systematic error (positive is upwards) of altitude of carrying unit	-
Size of footprint memory	footprint	Size of memory of ground footprints; that is the four corners given in Lat/Lon of the FOV on ground. A value of zero inhibits storing of this information.	
Plot sweeps in matlab	-	Check if a Matlab plot window should be opened to (in fire) show how the footprints are distributed on ground	
Sigmoid Scale	-	The scaling factor in the expression for target classifiability, $\mathbf{P}_{\mathbf{det}}$. This value divides the number-of-double-bars before input in the expression	
Sigmoid Exponent	-	The exponent in the expression for target classifiability. A value of 3.76 corresponds to the Johnson criterion (in	

		that case, Sigmoid Scale = 1.0).	
Velocity reports	-	Check if the sensor should be able to give reports about target velocity	
Velocity meas. bearing error	degrees	Gaussian random error of velocity bearing measurement	
Velocity meas. speed error	meters/second	Gaussian random error of speed measurement	

Function Initiation Points

DETECT

Output Variable	Datatype	Description
SENSOR_ID	FEquipmentIDType	The unique Equipment ID of this sensor.
SENSOR_TYPE	char *	Delivers the string "Video".
TARGET_CORRECT_NAME	char *	The true target unit name
NBR_OF_TARGET_FOCALS	FIntegerType	The number of arguments in TARGET_FOCALS and TARGET_PROBMASS. Currently the sensor only delivers one argument.
TARGET_FOCALS	FMALObj	The MAL currently only contains one string argument (named "H1") representing a subset of possible targets.
TARGET_PROBMASS	FMALObj	The MAL currently only contains one FReal argument (named "H1") representing the probability mass corresponding to the single hypothesis in TARGET_FOCALS. That is, the probability proposed by the sensor that the real target type will be found in the TARGET_FOCALS list of singletons
TARGET_POS_LLA	FVectorType	Target position in (lat,lon,alt)
TARGET_POS_NSWEROT	FVectorType	Error ellipsoid sigma of reported target position. X: Error in north – south direction

		(m). Y: Error in east – west direction (m). Z: Clockwise rotation of this ellipsoid (degrees).
TARGET_SPEED	FRealType	Estimated target speed (m/s). (Only delivered if “Velocity reports” checkbox is checked)
TARGET_HEADING	FRealType	Estimated target heading (rad) clockwise from north direction. (Only delivered if “Velocity reports” checkbox is checked)
TARGET_SPEED_ERR	FRealType	Standard deviation of the Gaussian target speed error (m/s). (Only delivered if “Velocity reports” checkbox is checked)
TARGET_HEADING_ERR	FRealType	Standard deviation of the Gaussian target heading error (rad). (Only delivered if “Velocity reports” checkbox is checked)
DETECTION_TIME	FJulianType	Time of detection.

Message Generation Points

None

Method Implementation

The Image Sensor recognizes the visual signature FOIIFVisualSignature. This signature consists of a list of attributes with corresponding “normalized signal strengths” (actually corresponds to the inverse of the number of double bars over target minimum projected dimension that have to be resolved) for detection at that level. Each attribute represents a subset of all possible targets. The top attribute is the correct target type alone and the following attributes always include the subset of the preceding attribute. Each attribute of the signature is checked for detection, starting with the most specific level. The detection probability P_{det} for an attribute with signal strength *Sig* is given by the sigmoid curve

$P_{det} = (Sig/q)^e / (1 + Sig/q))^e$

where *q* and *e* is the above mentioned Sigmoid Scale and Sigmoid Exponent, respectively. Values q = 1, e = 3.76 corresponds to the so-called *Johnson criterion* for a trained human to be able to detect/classify/identify an object.

The signal strength is computed with several checks for simple detection. That is, checks that the received signal strength is larger than the minimum signal strength for detection at the most general level in the corresponding visual signature list. The projected image of the object (approximated by an ellipsoid with the axis widths given in the signature model) onto the “CCD plate” is computed. Now a check is done to ensure that the number of pixels of the “CCD plate” that are illuminated by the projected image of the object is at least as large as the number of pixels set in the GUI. Then a check is done to ensure that the Rayleigh criterion (using the wavelength and objective diameter set in the GUI) allows the object to be resolved with the required number of double bars over the minor dimension of the projected image for detection at the most general level. If any of these checks fail, there will be no detection at all. The temperature difference (IR case) or light contrast (visible case) “TD” is in the current code currently hard-wired to 0.2 degrees Celsius. A check is done for the type of terrain where the object is positioned. The TD is reduced if the object is watched through tree foliage. Ground grazing angles of the Line-of-sight are unfavoured. The typical clutter level of the terrain is also is into merged into this reduction. The more clutter, the larger reduction (lower probability to discern the object). The effective number of resolvable double bars is finally computed.

A uniform random number generator is sampled for values ranging between 0 and 1.

The sigmoid function, with the scaling factor and exponent given in the GUI, is now used to estimate the probability of detection for the different classification levels, using the effective number of resolvable bars as input.

The first attribute that has a probability of detection higher than the random number obtained from the random number generator will alone determine the target classification and belief. It is chosen and shipped to the dataprocessor where it is expanded to its singletons and confused. The degree of confusion depends linearly on a random fuzziness parameter F (uniform between 0 and 1). F is 1 if the chosen classification was infinitely close to be one step less precise. F is 0 if the chosen classification was infinitely close to be one step more precise. F varies linearly in between. If F=1 the confusion follows the values of the confusion matrix. If F=0 there is no confusion at all. The resultant probability mass equals the degree of confusion. If none of the signature attributes is detected there will be no detection report.

The target position, speed and heading estimates are calculated from the position of the unit carrying this sensor. First, fixed drift errors, as well as Gaussian random errors are added to the North – South, East – West and Altitude position of the unit. The unit is put in this “hypothetical position”. Then, in the same manner, errors are added to the Azimuth and Elevation of the (correct) line-of-sight (LOS) towards the target. The resulting LOA is now extended from the “hypothetical position” until it intersects the ground. The latitude/longitude/elevation of that point will be reported as the position of the target. The reported position error will be the ellipse on the ground which is obtained if a elliptical-conical bundle of rays (extending from a focus in the sensor) with azimuth and elevation widths equal to the Gaussian random errors of those quantities is extended along the LOS.

Commands

None

Query entities

The following Queries can be sent to the image sensor using the **FEntityQuery** Kernel function in the C-API:

MEMORYDUMP returns the set of stored footprints in the MAL:

- DUMPSIZE Number of scanned footprints to be delivered

For each footprint there is one MAL with name FPx, where x is footprintnumber, with range 1 < footprintnumber < DUMPSIZE which contain:

o SENSOR_ID Equipment ID of the instance of this sensor

o C1, C2, C3, C4 four FCoordinateType Lat/Lon coordinates of the footprint corners, ordered counter-clockwise

o FOOTPRINT_TIME the simulation time when the footprint where scanned

o DETECTION_PROB the estimated detection probability of a “standard target” (currently a BMP-3) located on the ground, oriented pointing towards north, and in the boresight of the sensor

MEMORYSIZE returns in the MAL:

- MEMORYSIZE Number of scanned footprints currently in memory

SENSOR_STATUS returns last scanned footprint in MAL:

- STATUS_TIME Current simulation time when issuing the query
- SENSOR_ID Equipment ID of the instance of this sensor
- VERTEX1, VERTEX2, VERTEX3, VERTEX4 four FCoordinateType Lat/Lon coordinates of the footprint corners, ordered counter-clockwise
- NUMBER_OF_VERTICES is always 4 for this sensor
- DETECTION_PROB the standard detection probability as in MEMORYDUMP above
- SENSOR_TYPE is currently only “Video” for this sensor

8.9 Ground Sensor Network Model

Overview

The Ground Sensor Network (GSN) models a network of acoustic sensors. The model is capable of detection and classification with probabilities dependent on target signature strengths, the GSN model parameters and the confusion matrix of a FOIIFIDlibrary dataprocessor. The nodes of the network are assumed to be placed in such a way that distance to target and terrain can be ignored inside the network range. Outside, no detections are made. The position and velocity estimates are distorted and delivered with error estimates.

To function with the Fusionnode the Soldier Sensor should be used together with the Sensor Manager model.

Parameters

Name	Units	Description	Keyword
Library ID Data Processor	-	Name of dataprocessor to use for target classification. Should be of type FOIIFIDlibrary.	-
Number of Nodes	-	number of sensing nodes in the sensor network.	-
Max Range per Node	meters	Radius of a nodes detection area.	-
Max Range for Total Network	meters	The radius of the detection area for the whole network. Automatically calculated as [Max Range per Node * sqrt(Number Of Nodes)].	-
Scan Period	seconds	Number of seconds between sensor scans.	-
Stddev Position	meters	The standard deviation of the target positional error (Gaussian).	-
Stddev Speed	meters/second	The standard deviation of the error in target speed (Gaussian).	-
Stddev Heading	radians	The standard deviation of the error in target heading (Gaussian).	-
Detection Sensitivity	-	Parameter used in the detection and classification calculations. See the method implementation description of DETECT.	-
Detection Inflection Point	-	Parameter used in the detection and classification calculations. See the method implementation description of DETECT.	-

Function Initiation Points

DETECT

Output Variable	Datatype	Description
SENSOR_POS_ECR	FVectorType	Sensor position in ECR Coordinates
SENSOR_ID	FEquipmentIDType	The unique Equipment ID of this sensor.
SENSOR_TYPE	char *	Delivers the string "GroundSensorNetwork".
TARGET_CORRECT_NAME	char *	The true target unit name
NBR_OF_TARGET_FOCALS	FIntegerType	The number of arguments in TARGET_FOCALS and TARGET_PROBMASS. Currently the sensor only delivers one argument.
TARGET_FOCALS	FMALObj	The MAL currently only contains one string argument (named "H1") representing a subset of possible targets.
TARGET_PROBMASS	FMALObj	The MAL currently only contains one FREAL argument (named "H1") representing the probability mass corresponding to the single hypothesis in TARGET_FOCALS.
TARGET_POS_LLA	FVectorType	Target position in (lat,lon,alt)
TARGET_POS_ERR	FRealType	Standard deviation of the Gaussian target positional error.
TARGET_SPEED	FRealType	Estimated target speed (m/s).
TARGET_HEADING	FRealType	Estimated target heading (rad).
TARGET_SPEED_ERR	FRealType	Standard deviation of the Gaussian target speed error.
TARGET_HEADING_ERR	FRealType	Standard deviation of the Gaussian target heading error.
DETECTION_TIME	FJulianType	Time of detection.

Message Generation Points

None

Method Implementation

DETECT

The GSN recognizes the acoustic signature FOIIFAcousticSignature. The FOIIFAcousticSignature consists of a list of attributes with corresponding signal strengths. Each attribute represents a subset of all possible targets. The top attribute is the target alone and the following attributes always include the subset of the preceding attribute. Each attribute of the signature is checked for detection, starting with the most specific level. The detection probability P_{det} for an attribute with signal strength *Sig* is given by

$$P_{det} = 0.5 + 0.5 * \tanh(Sig / DetectSensitivity - DetectInflectionPoint)$$

where DetectSensitivity and DetectInflectionPoint are model parameters. The distances between sensors and target are not accounted for except that units outside the maximum range are ignored. The first attribute that is detected will alone determine the target classification and belief. It is shipped to the dataprocessor where it is transformed to singletons and confused. The degree of confusion depends linearly on a random fuzziness parameter F (uniform between 0 and 1). If F=0 the confusion follows the values of the confusion matrix. If F=1 there is no confusion at all. The resultant probability mass equals the degree of confusion. If none of the signature attributes is detected there will be no detection report.

The target position, speed and heading estimates are calculated from the real positions by adding an error. The error is drawn from Gaussian distributions defined by the user specified GSN standard deviation parameters.

Commands

None

8.10 Soldier Sensor Model

Overview

The Soldier Sensor so far only models the human as an intelligent visual sensor. The model is capable of detection and classification with probabilities dependent on target signature strengths, target distance, the model parameters and the confusion matrix of a FOIIFIDlibrary dataprocessor. The position and velocity estimates are distorted and delivered with error estimates. Outside the maximum range no detections are made.

To function with the Fusionnode the Soldier Sensor should be used together with the Sensor Manager model.

Parameters

Name	Units	Description	Keyword
Library ID Data Processor	-	Name of dataprocessor to use for target classification. Should be of type FOIIFIDlibrary.	-
Scan Period	seconds	Number of seconds between sensor scans.	-

Max Range Stddev Position	meters	The standard deviation of the target positional error at the maximum range (Gaussian).	-
Max Range Stddev Speed	meters/second	The standard deviation of the error in target speed at the maximum range (Gaussian).	-
Max Range Stddev Heading	radians	The standard deviation of the error in target heading at the maximum range (Gaussian).	-
Detection Sensitivity	-	Parameter used in the detection and classification calculations. See the method implementation description of DETECT.	-
Detection Inflection Point	-	Parameter used in the detection and classification calculations. See the method implementation description of DETECT.	-

Function Initiation Points

DETECT

Output Variable	Datatype	Description
SENSOR_POS_ECR	FVectorType	Sensor position in ECR Coordinates
SENSOR_ID	FEquipmentIDType	The unique Equipment ID of this sensor.
SENSOR_TYPE	char *	Delivers the string "Soldier".
TARGET_CORRECT_NAME	char *	The true target unit name
NBR_OF_TARGET_FOCALS	FIntegerType	The number of arguments in TARGET_FOCALS and TARGET_PROBMASS. Currently the sensor only delivers one argument.
TARGET_FOCALS	FMALObj	The MAL currently only contains one string argument (named "H1") representing a subset of possible targets.
TARGET_PROBMASS	FMALObj	The MAL currently only contains one FREAL argument (named "H1") representing the probability mass corresponding to the single hypothesis in TARGET_FOCALS.

TARGET_POS_LLA	FVectorType	Target position in (lat,lon,alt)
TARGET_POS_ERR	FRealType	Standard deviation of the Gaussian target positional error.
TARGET_SPEED	FRealType	Estimated target speed (m/s).
TARGET_HEADING	FRealType	Estimated target heading (rad).
TARGET_SPEED_ERR	FRealType	Standard deviation of the Gaussian target speed error.
TARGET_HEADING_ERR	FRealType	Standard deviation of the Gaussian target heading error.
DETECTION_TIME	FJulianType	Time of detection.

Message Generation Points

None

Method Implementation

DETECT

The Soldier Sensor recognizes the visual signature FOIIFVisualSignature. The FOIIFVisualSignature consists of a list of attributes with corresponding signal strengths. Each attribute represents a subset of all possible targets. The top attribute is the target alone and the following attributes always include the subset of the preceding attribute. Each attribute of the signature is checked for detection, starting with the most specific level. The detection probability P_{det} for an attribute with signal strength *Sig* is given by

$$P_{det} = 0.5 + 0.5 * \tanh(Sig / DetectSensitivity - DetectInflectionPoint)$$

where DetectSensitivity and DetectInflectionPoint are user specified model parameters. It is then lowered linearly with distance according to

$$P_{det} = P_{det} * (1 - 0.5 * distance\_to\_target / MaxRange)$$

($P_{det} = 0$ outside the maximum range). The first attribute that is detected will alone determine the target classification and belief. It is shipped to the dataprocessor where it is transformed to singletons and confused. The degree of confusion depends linearly on a random fuzziness parameter F (uniform between 0 and 1). If F=0 the confusion follows the values of the confusion matrix. If F=1 there is no confusion at all. The resultant probability mass equals the degree of confusion. If none of the signature attributes is detected there will be no detection report.

The target position, speed and heading estimates are calculated from the real positions by adding an error proportional to the target distance. The error is drawn from

Gaussian distributions defined by the user specified GSN standard deviation parameters and multiplied by *(distance/MaxRange)*.

Commands

None

8.11 FOIIFComIntInterceptor Communication Intelligence Interceptor Equipment Model

Overview

The Communication Intelligence Interceptor ComDevice intercepts messages sent by an ordinary FQPSimpleRadio ComDevice with the settings described below. The units carrying those radio ComDevices should load the FOIIFComIntTalkInit (initiates communication) or FOIIFComIntTalk (replies on messages) cognitives to send messages. These messages (specially designed to be intercepted) are expected to be of the FOIIFMsgComIntReveal type, and be processed by the FOIIFComIntProcessReveal message processing model that should be loaded by the unit carrying this interceptor.

Parameters

Name	**Units**	**Description**	**Keyword**
Channel to listen on	-	Radio channel to intercept messages on. If zero, all channels are searched.	-
100% intercept within range	meters	Guaranteed intercept of all messages within this range from interceptor	-
Linear falloff to 0% at range	meters	Intercept probability falls off linearly to zero in the range interval between above parameter and this parameter	-
1-sigma error or intercept bearing	degrees	Gaussian measurement error of the intercept bearing	-

Function Initiation Points

None

Message Generation Points

None

Method Implementation

When checking which radio emitters can be intercepted, the channel settings of them are checked against the settings of the interceptor. Also, in order to be intercepted, a radio emitter must be of the FQPSimpleRadio class, and have the (sub) string “Revealer” somewhere in its name. Network and Network Type are arbitrary. For each message that could potentially be intercepted, the distance to the emitter is calculated. If within the 100% interception range, it is always received, or “intercepted”. If it is between that distance and the 0% interception range, the probability for a message to be intercepted falls off linearly to zero. Outside that range, a message is never intercepted.

Commands

None

8.12 Communication Intelligence Cognitive Model

8.12.1 Overview

The Communication Intelligence cognitive models can give the following three abilities, or “set-ups” to units:

1. To transmit “radio messages” with a radio (“emitter”) at certain geographical regions using one of two simple schemes:
   a. Broadcast (a commander sends an order, all subordinates replicate)
   b. Dialogue “Ping-Pong” communication (a number of messages back and forth between two units)
2. To intercept these messages, and to send sorted intercepts to a local bearings fuser unit
3. To fuse bearings using bearing-crossing methods to localize the emitter, and to send a reports about positioned radio emitters

There can be an arbitrary amount of units that load the first type of cognitive. There can be an arbitrary number of groups where one all but one unit in each group load the second type of cognitive, and the last unit loads the third cognitive.Example Script for point one above; units that should be able to transmit messages, and get “revealed” by the interceptors:

Example Script for set-up scheme one above; units that should be able to transmit messages, and get “revealed” by the interceptors:

A unit which wants to be able to start communication includes the following in its script:

```
 EMPLOY COMDEVICE COMINTRevealer;

PERFORM STARTUP USING FOIIFComIntTalkStartup PARAMETER
(TALKREGIONSFILE = "talkpositions.txt");

PERFORM TALKINIT USING FOIIFComIntTalkInit;
PERFORM TALK USING FOIIFComIntTalk;

TRANSMIT FOIIFMsgComIntReveal AT COMINT_REVEAL USING
COMINTRevealer;

RECEIVE  FOIIFMsgComIntReveal                   USING
FOIIFComIntProcessTalk;

.

START ...

.

.

INITIATE ORDER;
```

while the rest of the communicating units only include the following:

```
EMPLOY COMDEVICE COMINTRevealer;

PERFORM STARTUP USING FOIIFComIntTalkStartup;

PERFORM TALK USING FOIIFComIntTalk;

TRANSMIT FOIIFMsgComIntReveal AT COMINT_REVEAL USING
COMINTRevealer;

RECEIVE  FOIIFMsgComIntReveal                   USING
FOIIFComIntProcessTalk;
```

Example script for set-up scheme two above; units (called “interceptors”) that together should be able to intercept, localize, and report about intercepted emitters:

```
EMPLOY   COMDEVICE COMINTInterceptor;

EMPLOY   COMDEVICE R11_INT_TO_FUS;

PERFORM  STARTUP                            USING
FOIIFComIntRevealStartup;

PERFORM  TEST_FOR_FUSE                      USING
FOIIFComIntReveal;

RECEIVE  FOIIFMsgComIntReveal               USING
FOIIFComIntProcessReveal;

TRANSMIT FOIIFMsgComIntFuse AT COMINT_FUSE USING
R11_INT_TO_FUS;
```

```
.

START ...

.

INITIATE TEST_FOR_FUSE;
```

Example script for set-up scheme three above; units that receives the interceptors' reports, localizes emitters by bearing-crossing methods, and report about the estimated emitter position:

```
EMPLOY   COMDEVICE R11_INT_TO_FUS;

EMPLOY   COMDEVICE RadioNtoN PARAMETER (CHANNEL = 1);

PERFORM  STARTUP                 USING
FOIIFComIntFuseStartup;

PERFORM  TEST_FOR_REPORT         USING FOIIFComIntFuse;

PERFORM  DETECT                  USING
FOIIFSensorManagerReport;

RECEIVE  FOIIFMsgComIntFuse      USING
FOIIFComIntProcessFuse;

TRANSMIT FOIIFMsgREPORT AT REPORT USING RadioNtoN;

.

START ...

.

INITIATE TEST_FOR_REPORT;
```

8.12.2 FOIIFComIntFuse

FOIIFComIntFuse periodically checks if bearings towards the same emitter have been reported from its associated interceptor units. If so, the best bearings (see detailed description on communication below) are combined in order to get an estimate of the emitters' position. The position together with an error ellipse is then reported. This is repeated for all emitters that can be localized.

- Intended Function

FUSE

- Execution Mode

CONTINUOUS

- Inputs

None

- Parameters

None

- Function Initiation Points

DETECT

- Message Generation Points

None

8.12.3 FOIIFComIntFuseStartup

FOIIFComIntFuseStartup performs startup work for the fuser cognitive such as initializing lists for received reports on intercepted emitters etc.

- Intended Function

STARTUP

- Execution Mode

SINGLE

- Inputs

None

- Parameters

MEMSIZE, ERRORTOLERANCE, MAXTIMEDIFFERENCE, PURGEAGE

- Function Initiation Points

None

- Message Generation Points

None

8.12.4 FOIIFComIntProcessFuse

Push:es reports from an interceptor onto a list for later bearings-crossing processing

- Inputs

A report from an interceptor to be put on a list

- Function Initiation Points

None

- Message Generation Points

None

8.12.5 FOIIFComIntProcessReveal

Push:es intercepted messages from a revealed unit onto a list for later judging if it should be sent to the fuser for bearings-crossing processing.

- Inputs

An intercepted message from a revealed unit. The message is to be put on a list

- Function Initiation Points

DETECT

- Message Generation Points

None

8.12.6 FOIIFComIntProcessTalk

FOIIFComIntProcessTalk processes all messages of the FOIIFMsgComIntReveal type that this unit receives. A decision is made if the below FIP will be initiated depending on if the unit should answer the message. This is depending on if it is a ping-pong, or a broadcast message. If it is a ping-pong message, answer if the unit from which the message was received is the current “talk partner” of this unit. If, on the other hand, it is a broadcast message, and the unit from which the message was received is the commander of this unit OR a subordinate within the same organizatorial unit as this unit, an answer is transmitted. In the latter case, the subordinate must then have a number one step under the number of this unit. See html-documentation for a more detailed description.

- Inputs

The FOIIFMsgComIntReveal message to be parsed.

- Function Initiation Points

DETECT

- Message Generation Points

None

8.12.7 FOIIFComIntReveal

FOIIFComIntReveal periodically checks the list of intercepted messages, and sends reports (messages) about intercepted emitters to the fuser. These messages are processed in the fuser unit by FOIIFComIntProcessFuse.

- Intended Function

FUSE

- Execution Mode

CONTINUOUS

- Inputs

None

- Parameters

None

- Function Initiation Points

DETECT

- Message Generation Points

None

8.12.8 FOIIFComIntRevealStartup

FOIIFComIntRevealStartup performs startup work for the interceptor cognitive such as initializing lists to push “measured” parameters of intercepted emitters on.

- Intended Function

STARTUP

- Execution Mode

SINGLE

- Inputs

None

- Parameters

Memsize

- Function Initiation Points

None

- Message Generation Points

None

8.12.9 FOIIFComIntTalk

- Intended Function

REVEAL

- Execution Mode

CONTINUOUS

- Inputs

None

- Parameters

None

- Function Initiation Points

DETECT

- Message Generation Points

COMINT_FUSE

8.12.10 FOIIFComIntTalkInit

- Intended Function

FUSE

- Execution Mode

CONTINUOUS

- Inputs

None

- Parameters

None

- Function Initiation Points

DETECT

- Message Generation Points

None

8.12.11 FOIIFComIntTalkStartup

FOIIFComIntTalkStartup performs startup work for the units that should be able to exchange messages that is to be intercepted by the interceptors. This includes reading the file containing positions where to initiate communication, and their initiation probability etc, see html-documentation.

- Intended Function

STARTUP

- Execution Mode

SINGLE

- Inputs

None

- Parameters

MEMSIZE, ERRORTOLERANCE, MAXTIMEDIFFERENCE, PURGEAGE

- Function Initiation Points

None

- Message Generation Points

None

8.13 Detailed description of the radio communication intelligence functionality

All cognitives related to this functionality are found in the dataset FOIIFComInt.

All messages related to this functionality are found in the dataset FOIIFCIMessages.

A unit can load (in a PERFORM statement, see example below) the cognitive models FOIIFComIntTalkStartup and FOIIFComIntTalk to receive and send answers (but not start a communication) on the “reveal” type radio messages. Those units that also load a third cognitive, FOIIFComIntTalkInit (as well as setting the parameter TALKREGIONSFILE in the above mentioned FOIIFComIntTalkStartup) are also able to start a communication by sending a message in certain geographical regions defined in a specific file, whose name should be given to the TALKREGIONSFILE parameter. There is also a "base probability" to be set in this file, of initiating messages irrespective of geographical position. This TALKPOSITIONS file should be located in the directory pointed to by the environment variable whose name is given by the DataFilesPath scenario variable. Two communication modes are available; Dialogue and Broadcast. A dialogue is a ping-pong sending between two units. Broadcast, on the other hand, means that a commander (no lowest-level units can issue broadcasts) initiates a message, and his subordinates (one organizatorial level below him) answer in number order. A commander has a name "R1B2C3P4_C" meaning "Regiment 1, Battallion 2, Company 3, Platoon 4", and "_C" means commander of the lowest (here platoon) level just before the "_C". Thus, “R1B1_C” is the battalion commander of battalion 1 under regiment 1. Lowest subordinates (“privates”, or Vehicles) have numbers like R1B2C3P4Vx. where the "x" number corresponds to the vehicle number in the platoon (= 1,2,3,...). Normally, different networks or channels are used within different organization levels to separate out those units which should listen. Here, all units can use the same radio, so all units can listen to all “reveal” messages. The name of the units gives their organization position, and that name is checked by the message processing model to see if a unit should answer a message. If the organization is large, this means that many links will appear (maybe as many as the number of units squared if all of them should be able to reveal themselves). This can affect memory and performance. If so, different networks and channels should be used within lower organization units. The interceptors can be set to listen to all reveal messages irrespective of network and channel.

A unit which wants to be able to start communication include the following in its script:

```
PERFORM STARTUP USING FOIIFComIntTalkStartup PARAMETER
(TALKREGIONSFILE = "talkpositions.txt");

PERFORM TALKINIT USING FOIIFComIntTalkInit;

PERFORM TALK USING FOIIFComIntTalk;
```

```
.
START ...
.
INITIATE ORDER;
```

while the rest of the communicating units only include the following:

```
PERFORM STARTUP USING FOIIFComIntTalkStartup;

PERFORM TALK USING FOIIFComIntTalk;
```

A dialogue message is sent from an initiating sender to a receiver that answers to the sender that answers again etc. There can be a well defined number of answers, or a probability for an answer that is checked every time.

For the R1B2C3P4 platoon case, a broadcast type message from the platoon commander R1B2C3P4_C will result in an answer from R1B2C3P4V1. R1B2C3P4V2 waits until R1B2C3P4V1 has answered before he answers himself. R1B2C3P4V3 waits for the answer from R1B2C3P4V2 and so on until all units in the platoon have answered. There is a probability that a unit will fail to answer, which will lead to an interrupt of the propagation of this answering behaviour through the organization level (here platoon). If a battalion commander R1B2_C broadcasts a message, the company commanders R1B2C1_C, R1B2C2_C, … will answer. It is possible to set a probability less than 1 that a subordinate will answer. If this results in a fail to answer, none of the remaining subordinates with higher numbers will answer.

The time difference between the transmitted message from the commander and the answer from ...subordinate 1, or an answer from subordinate x and the answer from subordinate x+1 is set in the prototype for the FOIIFComIntTalk process method (the delay with its delta). This method is the one that is initiated upon receipt of a message from the unit just before in this "message chain".

As mentioned, only the process method FOIIFComIntTalkInit can initiate messages from the beginning. This process method should be called repeatedly (set in the prototype for FOIIFComIntTalkInit, in the repeat field). Every time it is called, a check is done to see if the unit executing it is located within a defined geographical "talk region" if so, a message will be transmitted with the probability set in the TALKREGIONSfile for that circular talkregion. A check is always done based on the base probability if a message should be transmitted. If so happens, no check of transmit based on geographical location will be done.

It is practical to store the above scripts as dictionary scripts that can be included by all commanders or subordinates that want to communicate. If the intention is to intercept only certain messages, the simplest way is to equip these units with the same type of transmitter sending only on a certain channel (the channel to which the interceptors listen to), for instance by equipping the unit with a RadioNtoN radio sending on a fixed channel, and let the interceptors listen only on that channel. Currently, the interceptors listen on all networks at the same time. Furthermore, in order to be intercepted, the Communication Device must be of the FQPSimpleRadio class, and have the (sub) string “Revealer” somewhere in its name. Let's say that we equip our units that should be revealed with an FQPSimpleRadio called COMINTRevealer, being a RadioNtoN, all using channel 100.

So, the following should be added in the above unit scripts for both commanders and subordinates:

```
 EMPLOY COMDEVICE COMINTRevealer;
```

What will happen when a unit initiates a communication, and when a subordinate answers, is that a message of type FOIIFMsgComIntReveal is sent.

Finally for the communicating units’ scripts, this means that the following should also be included for both commanders and subordinates:

```
 TRANSMIT FOIIFMsgComIntReveal AT COMINT_REVEAL USING
COMINTRevealer;

RECEIVE  FOIIFMsgComIntReveal    USING
FOIIFComIntProcessTalk;
```

Now, these FOIIFMsgComIntReveal messages can also be intercepted by the ComInt equipment. ComInt interceptor units should equip themselves with an instance of the class FOIIFComIntInterceptor (subclass to FComDevice) to do this, let's say that we equip our interceptor units with an interceptor equipment of this class called COMINTInterceptor. These can be tuned to be able to intercept 100% of all messages transmitted within a certain given radius from the interceptors position. Outside that radius, it intercepts messages with a probability that is linearly falling from 100% to 0% between that radius, and another given wider radius. It can be set to give intercept bearing reports with a set standard deviation. Furthermore, it can be set to listen to all channels (channel number set to zero), or to certain channel numbers (channel numbers > 0). Currently, as mentioned above, it listens on all networks.

The interceptor units should load the message processing method FOIIFComIntProcessReveal. When an interceptor intercepts a message of type FOIIFMsgComIntReveal, it pushes it onto a list. It goes on doing so as long as it lives. Furthermore, the interceptor unit should load the process method FOIIFComIntReveal. That method should run in repeated mode. There is no need for a delay or repeat delta. What happens when it is initiated (with FIP TEST_FOR_FUSE) is that the list of intercepted messages (pushed since the last earlier initiation) is looked through, and the bearing of the first (most recently intercepted) instance of a revealed unit (a unit from which the interceptor unit has intercepted a message) is reported (as a message FOIIFMsgComIntFuse generated at the MGP COMINT_FUSE in FOIIFComIntReveal) to the ComInt fusion node which fuses intercept bearings from the different interceptor units which report to it. Also, the intercept time is reported. This behavior is repeted for all different revealed units found in the list. The script of an interceptor unit should contain the following:

```
EMPLOY   COMDEVICE COMINTInterceptor;

EMPLOY   COMDEVICE R11_INT_TO_FUS;

PERFORM  STARTUP         USING FOIIFComIntRevealStartup;

PERFORM  TEST_FOR_FUSE       USING FOIIFComIntReveal;

RECEIVE  FOIIFMsgComIntReveal USING
FOIIFComIntProcessReveal;

TRANSMIT FOIIFMsgComIntFuse AT COMINT_FUSE USING
R11_INT_TO_FUS;

 .

START ...

.

INITIATE TEST_FOR_FUSE;
```

Since only the last occurrence of an intercept from a certain unit is reported, there is actually no need for pushing them on a list. Indeed, all older instances are deleted from the list after the report. The reason for using a list is that one might want to rebuild the code so the bearing-estimation behavior of the ComInt equipment can be more realistically modelled by comparing several intercepts of a revealed unit, each with different time of intercepts. This is not done in the current version.

Now, let's assume that we also have a unit which fuses the different bearings (and checks for bearing crosses). To begin with, it should (as well as all interceptor units reporting to it, see the script above) be equipped with a radio transmitter/receiver of type Radio1to1, that is here called R11_INT_TO_FUS. That radio is used only for reports from the interceptors to the fuser. Channel number and network number should here be set so only this group of interceptors and fuser can communicate intercepts. This means that other groups of interceptors/fuser can be deployed in the scenario, and not interfere with each other.

The fuser unit should load the message processing method FOIIFComIntProcessFuse. When a FOIIFMsgComIntFuse message is received by this model, the message is pushed onto a list, in the same manner as in the interceptor units. Furthermore, the fuser unit should load the process method FOIIFComIntFuse implementing the function TEST_FOR_REPORT (that is, test for a target report to the central fusion node). Like the FOIIFMsgComIntReveal model of the interceptors, the FOIIFComIntFuse model of the fuser should be initiated repeatedly. There is no need for a delay or repeat delta. When initiated, it goes through the list of all reports from the interceptors. The current version of this process method goes through the list and checks for "the best combination" of any two reported bearings (from a pair of two different intercepting units of course) that have been reported as a consequence of a message sent from one and the same revealed unit, and being intercepted by two intercepting units. That is, the association is perfect in this version. There is no ambiguity about which revealed unit the interceptors actually intercepted. The statement above about two bearings "The best combination" is currently defined as the lowest product between three factors: (1) The cotangent of the angle between the two bearings, but not lower than 0.25. (2) The maximum distance from any of the two interceptors to the bearing cross position. (3) The sum of the bearing measurement errors of the two interceptors. That is, bearing angles close to pi half, and short distances between interceptors and revealed unit, and small intercept errors are favoured. No hint of the distance to the revealed unit is assumed based upon "signal strength" of the intercepted message or so. All information about the position of the revealed unit is based upon crossings of the two best bearings (of course, more than two bearings if more than two interceptors can intercept the signal could be fused, for instance using Stansfield's algorithm). An error ellipse in this geometric configuration is then estimated (taking into account the measurement error in each of the interceptor devices), and this error, together with the estimated position of the revealed object (equal to the bearing cross position) and some more info is reported to the central Fusion Node. This report is implemented as a FIP called DETECT where the report provided is transformed into a message of the standard type recognized by the central fusion node. That message is typically sent using a RadioNtoN (as below), at some specified channel (here 1) where the central fusion node listens.

In total, the bearing fusion unit should contain the following:

```
EMPLOY   COMDEVICE R11_INT_TO_FUS;

EMPLOY   COMDEVICE RadioNtoN PARAMETER (CHANNEL = 1);

PERFORM  STARTUP                 USING
FOIIFComIntFuseStartup;

PERFORM  TEST_FOR_REPORT         USING FOIIFComIntFuse;

PERFORM  DETECT                  USING
FOIIFSensorManagerReport;

RECEIVE  FOIIFMsgComIntFuse      USING
FOIIFComIntProcessFuse;

TRANSMIT FOIIFMsgREPORT AT REPORT USING RadioNtoN;

.

START ...

.



INITIATE TEST_FOR_REPORT;
```

Finally, there are a set of parameters that could be provided to the FOIIFComIntRevealStartup and the FOIIFComIntFuseStartup process methods:

For FOIIFComIntRevealStartup:

MEMSIZE is the amount of pushed intercepted messages from the interceptors that at a maximum should be stored in the interceptor. When that number is obtained, a push-front of the latest message results in a pop-back of the oldest one (that is, pop-and-forget).

SMOOTHTIMES is set to a non-zero value if one wants the interceptoirs to repeat the time-of-intercept of intercepted messages “smoothed”. If not chosen, these times will be discrete times with difference set by the simulation clock ticks. If chosen, the time reported will be smeared out in a uniform distribution centered at the reported time, and with width equal to the simulation clock tick. This is preferable if the discreteness of times is unsuitable for algorithms analyzing the time pattern of intercepted messages.

For FOIIFComIntFuseStartup:

MEMSIZE is the amount of pushed messages from the interceptors that at a maximum should be stored in the fuser. When that number is obtained, a push-front of the latest message results in a pop-back of the oldest one (that is, pop-and-forget).

ERRORTOLERANCE is the multiple of the sums of the bearing errors of two intercepting units that the bearing difference of a cross must exceed in order not to be rejected immedeately. That is, if both intercepting devices have RMS errors of 5 degrees, and this multiple is set to 2, the difference of the bearing values measured by the two interceptors must exceed 2*(5+5)=20 degrees to remain as a potential report in further computations based on "the best combination" as described above. If no crosses on an intercepted unit exceed this value (20 in this example), there will be no report at all sent to the central fusion node about that revealed unit.

MAXTIMEDIFFERENCE is the time in seconds that the difference in time for the two intercepts must not exceed. That is, the time difference between the transmission of the messages received by the currently checked pair of interceptors. Often, a checked pair of reports concerns the same received message, so this time difference is zero. If a transmitting vehicle moves fast, and intercepts from two interceptors have a large time difference, the position of the cross will be wrong. This can happen if the revealed unit travels in an area beyond the 100% intercept probability radius of at least one of the interceptors.

PURGEAGE is the "age" of the reports from time-of-intercept after which they will be purged from the memory of the fuser. Every periodic initiation of FOIIFComIntFuse includes a test-for-purge pass in the end.

Reference: Stansfield, R., "Statistical Theory of DF Fixing", Journal IEEE, Vol. 94 Part 3A, n 15, 1947, pp. 762-770.

9 IFD03 FusionNode

The FusionNode consists of a Control module (with parts in both Flames and Matlab), a Logging module, and various analysis modules.
The Control module in Flames handles the interface between Flames models representing sensors and units and Matlab.
The Control module in matlab has two entry points from Flames: Flames_report is called whenever new reports are generated. It stores them (in the future, it could also do some preprocessing).
Flames_Analyze is called when an analysis should be performed. Currently this is set to be every 10th second, but this could be changed. It decides which analysis modules to call based on a list that gives the times when variour analyses should be performed. (NOTE: Track requires that it be called regularly after it has been initialised. Changing this requires rewriting the code somewhat).
Some analysis module might require terrain information (currently just Track). THe interface to the terrain database in Flames is handled by the terrain module.
Data is logged in two ways: in text-files that are later inserted into a SQL-database and used by the Visualizer; and in Matlab .mat-files that are loaded by the Matlab-part of Visualizer.
Track is the particle filter. It tracks on several different aggregation levels, but currently has no coupling between them. It relies on the output of the aggregation module for input for higher levels.
Aggregation handles clustering and classification. It calculates a conflict matrix, clusters the report (currently the last 200) and then classifies the generated clusters. This is performed for all levels up to battalion.
Sensor management is implemented in a rudimentary version. It chooses between 4 hard-coded UAV-paths (including drop of IAM).
There is also a Display module. This was used initially to get visualization during runtime, but was later replaced with the stand-alone Visualizer application. The run-time plotting of particles was the cause of some memory-crashes, so it is not used in the current version.

9.1 Documentation for Control Module in IFD03

The control module of the Fusionnode has two main purposes

1. It serves as interface between the external Flames environment and the internal methods of the Fusionnode (implemented in Matlab).
2. It manages the different fusion algorithms and their data flows.

The implementation of the Control Module can also be separated into two parts. The interface functions are implemented directly in C as a Flames Cognitive Model and the internal functions as a Matlab module.

The control module in IFD03 implements the control handling of which analysis modules to be called.

All reports, generated by Fire, is passed to the FusionNode by calling Flames_Report.

Fire starts the FusionNode by calling Flames_Analyze, which calls the aggregation-, tracking-, and sensorallocation module, in due time. The time intervals for calling these modules are set in Flames_Startup.

- **Flames_Startup** The function is called from Fire at startup. The function initializes all global variables.
- **Flames_Shutdown** The function is called from Fire when the scenario has terminated. The function closes opened files.
- **Flames_Report** The function is called from Fire every time a new report is generated. The report is stored in the global variable 'all_reports'.
- **Flames_Analyze** The function is called from Fire at a rate set in Flames. The function calls the analysis modules, in due time, **do_aggregation**, do_track and do_sensor. The result is plotted on the screen and saved in the Flash database.
- **do_track** This is the module that performs the tracking using particle filters. The function calls timestep_track with all new reports about: vehicles, platoons, companies, battalions, and negative sensor reports (ie, sensors that haven't observed any object).
- **do_sensor** This is the module that performs the sensor allocation. The function returns a sensorplan that is passed on to Flames_Execute_Sensor_Plan
- **do_sensor_statuses** Requests new sensor status from all sensors in all_sensor_statuses.
- **save_estimated_aggregate_count** Saves number of vehicles, number of platoons and number of companies, with the current Juilan time, in the file matlabplot_aggregate.

9.2 Documentation for the report format in IFD03

The report format is the matlab structure that defines the reports sent from Fire to Flames_Report(report_format).

The format is almost identical to the structures Companies, Platoons and Vehicles, only they also contain the field: 'target_class_explainstr' (string). And a list of the units it contain.

```
report_format = struct(
    'report_id', ID (number, set by Flames_Report),...

    'sensor_id', ID (number),...
    'sensor_pos', Position,...
    'sensor_type', TYPE (string),...
    'sensor_continuous_tracking', REPORT_NOT_TO_CLUSTRING_FLAG,...

    'target_class_focals', Class_focals,...
    'target_class_masses', Class_masses,...

    'target_pos', Position,...
    'target_pos_err_type', TARGET_POS_ERR_TYPE_FLAG,...
    'target_pos_err', standard_deviation (meter),...
    'target_pos_err_nswerot', [a b theta],...
    'target_pos_err_polygon, <Nx2 matrix>,...

    'target_heading', theta (-pi <= theta <= pi, 0 is north),...
    'target_heading_err_type', TARGET_HEADING_ERR_TYPE_FLAG,...
    'target_heading_err', deltatheta (radianer),...

    'target_speed', speed (m/s),...
    'target_speed_err_type', TARGET_SPEED_ERR_TYPE_FLAG,...
    'target_speed_err', deltav (m/s),...

    'target_correct_name', NAME (string, only for debugging and
evaluation),...

    'time_detected', detection_time (string),...
    'time_detected_num', dtime (set by MatLab),...
    'time_received', recieved time (string),...
    'time_received_num', rtime (set by MatLab),...
)
```

Position
    A 3-valued vector [LATITUD, LONGITUD, HEIGHT (meter)]

TARGET_POS_ERR_TYPE_FLAG
    0 = no pos err
    1 = circle, std dev in target_pos_err
    2 = ellipse, major and minor axis + rotation angle in target_pos_err_nswerot
    3 = polygon, Nx2 matrix target_pos_err_polygon contains coordinates

Class_focals
    el_i = Class_focals{i} contains the i:th set of elements the vehicle can be.
    el_i{j} contains the j:th element the vehicle can be.
    The type of el_i{j} is string.

Class_masses
    Class_masses(i) is the mass of focal elements focals{i}

TARGET_SPEED_ERR_TYPE_FLAG
    0 = no speed error given
    1 = stddev in target_speed_err

TARGET_HEADING_ERR_TYPE_FLAG
    0 = no heading error given
    1 = stddev in target_heading_err

Before using target_pos_err, all routines must check target_pos_err_type. The field target_pos_err, target_pos_err_nswerot, targe_pos_err_polygon will ONLY be valid if target_pos_err_type is set to the corresponding value.

If size(target_speed) == 0, the report doesn't contain any speed and thus also no speed error. Note that it could be possible for a report to have a target_speed but no target_speed_err -- it is the responsibility of Flames_report to make sure that a proper value is inserted into target_speed_err in this case. (Note: this is for future extensions.

If size(target_heading) == 0 -- see above, replacing speed with head

9.3 Instructions on Sending Reports in IFD03

9.3.1 Sending reports in Flames

There are two types of reports a sensor can send to the fusionnode. One is the target report containing information on a detected target, such as target position and classification. The other is the report of sensor status information, including information on current sensor operability and coverage. The procedure to generate both message types are identical, except for the included message attributes. For lists of recognized attributes, see the documentation on the FOIIFFusionNode.

To send a report from a sensor model to the fusionnode you have to create two MALs (Model Argument Lists). In the first, which you name "DATAMAL", you put all attributes of the report that you want to include. Then you add the DATAMAL to a second MAL, which you name "MSGMAL". This is the MAL that you use as argument when generating the message. The reason for using two MALs is to make it possible to add new report attributes without changing the message prototype. (In the prototype for a message you have to specify its exact constituents, but with this method it will always only contain a single MAL, the DATAMAL).

Example code for a cognitive model sending (target) reports:

1. DATAMAL = FMALCreate();
2. FMALAddFMAL(DATAMAL,"TARGET_FOCALS",FOCALMAL);
3. FMALAddFMAL(DATAMAL,"TARGET_PROBMASS",PROBMASSMAL);
4. FMALAddFInteger(DATAMAL,"NBR_OF_TARGET_FOCALS",2);
5. etc.
6. MSGMAL = FMALCreate();
7. FMALAddFMAL(MSGMAL,"MSGDATA",DATAMAL);
8. FBEGenerateMessage ("REPORT",MSGMAL ,0,0,0);

NOTE: If generating the message from an equipment model FBEGenerateMessageFromEquipment must be used instead of FBEGenerateMessage, otherwise a runtime error will occurr!

9.3.2 Example script code in Forge

(Fusionnode)

- PERFORM STARTUP USING FOIIFFusionNodeStartup;
- RECEIVE FOIIFMsgREPORT USING FOIIFFusionNodeProcessREPORT;
- RECEIVE FOIIFMsgSensorStatus USING FOIIFFusionNodeProcessSensorStatus;

(Sensor unit)

- TRANSMIT FOIIFMsgREPORT AT REPORT;

- TRANSMIT FOIIFMsgSensorStatus AT SENSOR_STATUS;

9.4 Documentation for Log Module in IFD03

The Log Module is responsible for logging simulation data to files. Two separate procedures are used.

1. **DATA-files.** Formatted data can be written to textfiles (.data). After scenario completion these files are parsed by the Postprocessor and the data is loaded into a MySQL database, from where it can be viewed in the IFD03 Visualizer. The data logged and the corresponding log-files are listed in the table below ("scenario" is the name of the simulated scenario). The logging to each file can be individually switched on/off in Flames_Startup in the Control Module.

Data	Filename
Report data	scenario.senrep.data
Vehicle data	scenario.vehicle.data
Platoon data	scenario.platoon.data
Company data	scenario.company.data
Vehicle particle histogram data	scenario.vehicle_particles.data
Platoon particle histogram data	scenario.platoon_particles.data
Company particle histogram data	scenario.company_particles.data

2. **MAT-files.** Matlab data can be directly saved as binary MAT-files (.mat). This data can only be read by Matlab code. The fusionnode logs in the following mat-files:
   - **rapport_loggade.mat** saved from Flames_Analyze contains
     - all_reports - Cell array containing all reports received by the fusionnode.
     - all_negative_reports - Cell array containing all negative reports received by the fusionnode. OBSOLETE!
     - all_reports_to_cluster - Cell array containing all reports used for clustering.
   - **matlabplot_sensor_adaption.mat** contains variables for visualization in the Matlab-view
   - **matlabplot_track.mat** contains variables for visualization in the Matlab-view
   - **matlabplot_aggregate.mat** contains variables for visualization in the Matlab-view

For more details, see the html-documentation and the detailed list of functions below.

- convert_time_to_julian Converts ordinary date to Flames compatible Julian number.
- init_aggregate_estimated_vehicle_count Initializes variables used to count vehicles, platoons and companies.
- init_log Prepares logfiles for writing.
- log_companies Logs company data.
- log_platoons Logs platoon data.
- log_report Logs report data. log_track Logs particle histogram data.
- log_vehicles Logs vehicle data.
- save_estimated_aggregate_count Saves number of vehicles, number of platoons and number of companies (MAT-file).

Note: in addition to logging data, it is also possible to display matlab graphics during the run of fire. For examples of how to do this, see the source-code for the Display module.

9.5 Documentation for Aggregation module in IFD03

The aggregation module in IFD03 implements clustering and classification. It is called from the Control module. The interface to aggregation consists of a number of functions that handle the different aggregation levels:

- reports_to_vehicles
- vehicles_to_platoons
- platoons_to_companies
- companies_to_batallions

For each of these levels, a conflict matrix is first calculated. This gives the conflict (in the Dempster-Shafer-sense) of putting two objects in the same super-object. Clustering is then performed. In order to determine the correct number of clusters, several trial clustering are done and the number of clusters to use is then determined according to the procedure described below.
After clustering, each cluster is classified and the template superobject that best fits it is determined. If the fit here is too bad, the superobject is removed.

For more details, see the local and global call graph in the html-documentation. For details on how the number of clusters to use is determined and on data management within the Aggregation module, see paper A from section 2.

- **do_aggregation** The function perform aggregation at all levels. From reports to batalions.
- **reports_to_vehicles** The function first decides the smallest, and the largest number of platoons we have, using set_report_cluster_interval. We caluculate the conflict matrix, using SetJreports2. We then call try_cluster. The resulting vehicles are sent to vehicle_record for storage.
- **vehicles_to_platoons** The function first decides the smallest, and the largest number of platoons we have, using set_vehicle_cluster_interval. We caluculate the conflict matrix, using SetJvehicles. We then call try_cluster.

- **platoons_to_companies** The function first decides the smallest, and the largest number of companies we have, using set_platoon_cluster_interval. We caluculate the conflict matrix, using SetJplatoons. We then call try_cluster.
- **companies_to_batalions** The function first decides the smallest, and the largest number of batalions we have, using set_company_cluster_interval. We caluculate the conflict matrix, using SetJcompanies. We then call try_cluster.
- **vehicle_record** The function keeps track of all vehicles after they don't leave any reports. After a certain time they are kicked out.
- **set_report_cluster_interval** The function uses doctrine data to give an interval of how many vehicles that could have been observed.
- **set_vehicle_cluster_interval** The function uses doctrine data to give an interval of how many platoons that could have been observed.
- **set_platoon_cluster_interval** The function uses doctrine data to give an interval of how many companies that could have been observed.
- **set_company_cluster_interval** The function uses doctrine data to give an interval of how many batalions that could have been observed.
- **SetJreports2** The function calculates the report conflict matrix. The conflicts are based on the geografical distance. The type conflict, using Dempsters rule of combination. And difference in direction. The old SetJreports could cause data fragmentation, since a new matrix was created at every call. SetJreports2, instead moves data around in the same matrix.
- **SetJvehicles** The function calculates the vehicle conflict matrix. The conflicts are based on the maximum geografical distance, two vehicle ever had, and the type conflict. Each known platoon are defined by its vehicles and their maximum distance in VehicleTypes. The conflict is calculated in template_conflict.
- **SetJplatoons** The function calculates the platoon conflict matrix. The conflicts are based on the maximum geografical distance, between any vehicles in the platoon, and the type conflict. Each known compay are defined by its platoons and their maximum distance in PlatoonTypes. The conflict is calculated in template_conflict.
- **SetJcompanies** The function calculates the company conflict matrix. The conflicts are based on the maximum geografical distance, between any vehicles in the company, and the type conflict. Each known battalion are defined by its companies and their maximum distance in CompanyTypes. The conflict is calculated in template_conflict.
- **template_conflict**. Distance conflict depending on types of objects. See papers for details.
- **try_cluster** The function decides which cluster function to be used. cluster_only_one or try_spin_cluster. This file contains the '**cluster hack**'. That is, if the number of objects are small enough, they are put in one cluster.
- **cluster_only_one** If the conflict matrix only consists of 0 and MAX_C. cluster_only_one will try to find the smallest possible number of cluster, while the total conflict is zero. The timecomplexity is O(n)
- **try_spin_cluster** Wrapper for do_spin_cluster. It first decides the number of cluster and then calls do_spin_cluster.
- **find_plateau** finds the plateau, between two given number of clusters, and returns the clustering at the plateau.

- **DirAndDistVehicles** This function calculates the maximum difference in direction, in degreas, and the largest distance, in meters, the two vehicles ever had.
- **DirAndDistPlatoons** This function calculates the maximum difference in direction, in degreas, and the largest distance, in meters, between any two vehicles, from the two platoons.
- **MAX_C** A constant. The maximum value in the conflict matrix. Typically set to 5.
- **correct_num_of_vehicles** Returns the number of vehicles found in the last 'max_number_to_cluster' reports in 'all_reports_to_cluster'. For debugging use only.
- **DistCoordErr** The smallest distance, in meter, between two coordinates, when considering the error.
- **make_vehicle_reports** This function calls make_vehicle_report for all generated clusters of reports.
- **make_vehicle_report** This function tries to classify a vehicle. It does not use get_classification.
- **make_platoon_reports** Calls make_platoon_report for all clustered vehicles.
- **make_platoon_report** This function uses get_classification and platoon-templates to determine the type of a platoon. If the match to template is too bad, the cluster is removed. Note that this function must first add Theta to the mass functions of the clustered vehicles!
- **make_company_reports** Similar to make_platoon_reports, but for companies.
- **make_company_report** Also similar to platoon level.
- **make_bat_reports** See platoon level.
- **make_bat_report** See platoon level.
- **do_spin_cluster** Wrapper function for da-clustering. Takes a number of arguments that control how to do the clustering. These arguments are differens for different levels, and are set in the appropriate function above. Arguments that can be set include: maximum outer and inner loops in the da-algorithm, allowed_conflict (the algorithm stops when this conflict has been reached),and a vector of different n_clusters to try. The function tries all n_clusters is has been given until it has found a conflict below the maximun allowed. Alpha, gamma, tau and epsilon are parameters for dacluster. Outputs are lists mapping reports to clusters, giving all report indices in the clusters, the conflict reached, the number of o uter and inner loops performed and an errorstatus. Errorstatus should be STATUS.ok if everything went fine, otherwise it indicates what went wrong. This output could be used to dynamically change the parameters in order to get a correct clustering. Note that some of the other output-argument MAY be valid even if errorstatus is not OK; see source code documentation for details.
- **dacluster** Implements the actual clustering code. Basically, this consists of two loops. The outer is exited when the spins are sufficiently peaked, while the inner exits when spin distribution has converged. The conflict matrix is used to calculate a mean field effective force on the spin distribution which changes the spins in the inner loop.
- **get_classification** This function is given a cluster and tries to classify it. Note that the mass functions in the cluster must first be modified so that they include Theta explicitly! The algorithm works by generating all possible

combinations of types in the clusters. The number of such combinations grows exponentially with the size of the cluster. Each of these combinations is then compared to all templates and two fitnesses are calculated. One of these is base on the match between total number of objects in the cluster and the template, the other also takes into account that types should be compatible. The fitnesses are weigthed by the probability mass of the combination, and total fitnesses for each template are calculated. If the fitness to the best template is too low, a flag is set so that the cluster can be removed. An additional output is the explainstr, which could be used to give the user an indication of why the system classified the cluster the way it did. Currently, this output is used for giving the second best template.

- **cellequal** Returns true if the given cell arrays are the same.
- **getnumcompatible** Given a template and a specific type, this function calculates the number of objects in the template that are compatible with the type.

9.5.1 A definition of templates

Templates are used to describe the organization of the enemy force. Their format should be as follows.

```
Notation:

read "::=" as "consists of"
"list of" means one or more occurrences of
"(a,b)" means the tuple a and b
text within <> are terminals in the grammar
"a|b" means either a or b

template ::= (template_name, list of  (type, number))

template_name ::= <string describing the unit>

number ::= <natural number >= 1>

type ::= list of object_type

object_type ::= <vehicle_type> | template_name
```

Note that type is currently just a list of one object_type, but it could be extended to be a list of several in the future. (This is the reason for the doube {{ and }} in the matlab code below. It also means that all code must use the somewhat awkward "{1}" addition to type below.)

A future definition of template would include other information, eg

```
template ::= (name, list of (type, number), characteristic_size,
behaviour_pattern)
```

where characteristic_size and behaviour_pattern would describe other information needed for the fusion node. This should be added as extra struct field in the matlab code.

9.5.1.1 Example matlab code:

Note that the distinction between () and {} is VERY important. It is also necessary to use double {{ and }} for type, in order to be able to easily extend the definition of templates.

Note: As written, it would be possible to just have one set of templates covering units at all levels in the hierarchy and containing an arbitrary number of other templates. For efficiency, the fusion node will contain different lists of templates for different leves: platoon_templates, company_templates, battalion_templates. But all code should be written so that they don't rely on "knowing" what kind of object it classifies. That is, it should work equally well if we replace its list of templates with the list [platoon_templates; company_templates; battalion_templates]. This enables us to use just one list of templates for all levels.

```
Example definition of some templates:

(templates.m)

% unit foo contains 2 T80 and 1 TG13

unit1.type(1) = {{'T80'}};
unit1.number(1) = 2;
unit1.type(2) = {{'TG13'}};
unit1.number(2) = 1;
unit1.name='foo';
unit1.size = 2;

% unit bar contains 1 T80, 2 LBIL and 1 TG11

unit2.type(1) = {{'T80'}};
unit2.number(1) = 1;
unit2.type(2) = {{'LBIL'}};
unit2.number(2) = 2;
unit2.type(3) = {{'TG11'}};
unit2.number(3) = 1;
unit2.name='bar';
unit2.size = 3;

% unit fubar contains 3 LBIL and either 1 T80 or 1 TANK
% (templates such as this are currently not needed,
% but we may need to support it in the future.)

unit3.type(1) = {{'T80','TANK'}};
unit3.number(1) = 1;
unit3.type(2) = {{'LBIL'}};
unit3.number(2) = 3;
unit3.name='fubar';
unit3.size = 2;

unit_templates1 = [ unit1; unit2; unit3];

% unit company contains 2 foo and either 2 bar or 2 fubar

unit4.type(1) = {{'foo'}};
unit4.number(1) = 2;
unit4.type(2) = {{'bar','fubar'}};
```

```
unit4.number(2) = 2;
unit4.name = 'company';
unit4.size = 2;

unit_templates2 = [unit4];
```

Code that uses the templates. This code displays all templates; its purpose is to show how to use the templates in matlab code.

NOTE: it is necessary to have the extra{1} after type in the innermost for-loop. Matlab requires this since we use {{ and }} in the definition of type.

```
function showtemplates(template_list)
% 030805 ponsve
% displays info on all templates in template_list
% also shows how to access parts of a template

for n=1:size(template_list,1)
    template = template_list(n);
    name = template.name;
    disp(sprintf('template nr %d contains %d types and is named
%s',n,template.size,name));
    for m=1:template.size
        number = template.number(m);
        type = template.type(m);
        disp(sprintf('    type nr %d. the unit should have %d
subunits which are one of',m,number));
        for b=1:length(type{1})
            disp(sprintf('            %s', type{1}{b}))
        end;
    end;
end;
```

Output from Matlab given code above:

```
>> showtemplates(unit_templates1)
template nr 1 contains 2 types and is named foo
    type nr 1. the unit should have 2 subunits which are one of
            T80
    type nr 2. the unit should have 1 subunits which are one of
            TG13
template nr 2 contains 3 types and is named bar
    type nr 1. the unit should have 1 subunits which are one of
            T80
    type nr 2. the unit should have 2 subunits which are one of
            LBIL
    type nr 3. the unit should have 1 subunits which are one of
            TG11
template nr 3 contains 2 types and is named fubar
    type nr 1. the unit should have 1 subunits which are one of
            T80
            TANK
    type nr 2. the unit should have 3 subunits which are one of
            LBIL
>>
```

```
>> showtemplates(unit_templates2)
template nr 1 contains 2 types and is named company
    type nr 1. the unit should have 2 subunits which are one of
            foo
    type nr 2. the unit should have 2 subunits which are one of
            bar
            fubar
>>
```

```
>> showtemplates([unit_templates1;unit_templates2])
template nr 1 contains 2 types and is named foo
    type nr 1. the unit should have 2 subunits which are one of
            T80
    type nr 2. the unit should have 1 subunits which are one of
            TG13
template nr 2 contains 3 types and is named bar
    type nr 1. the unit should have 1 subunits which are one of
            T80
    type nr 2. the unit should have 2 subunits which are one of
            LBIL
    type nr 3. the unit should have 1 subunits which are one of
            TG11
template nr 3 contains 2 types and is named fubar
    type nr 1. the unit should have 1 subunits which are one of
            T80
            TANK
    type nr 2. the unit should have 3 subunits which are one of
            LBIL
template nr 4 contains 2 types and is named company
    type nr 1. the unit should have 2 subunits which are one of
            foo
    type nr 2. the unit should have 2 subunits which are one of
            bar
            fubar
>>
```

9.6 Documentation for Tracking module in IFD03

9.6.1 PHD-particle filtering:

The Tracking of IFD03 is an implementation of the PHD particle filter described in paper D, except that here sensor positions are taken account of (see below).

Tracking is implemented on three independent levels: vehicles, platoons, and companies. Input for the vehicle tracking is observations from the sensors, while the higher level trackers use the output of the Aggregation module.

Output is a histogram representing a discretized 2D PHD over positions.

9.6.2 Implementation

The following list shows the call-structure. This is also visualized in the call-graph in the html-documentation.

- init_track initializes parameters used in the PHD filter

- timestep_track, calls
    - report_local_format, converts reports to internal format
    - particles_global_format, converts particles
    - **there are three instances of** tidssteg, that call
        - montecarloMotion
        - montecarloObs
        - likelihood
        - montecarlo

Tracking also uses the terrain module.

- init_track
  Called by the FusionNode once at start of execution.
  Sätter (globala) konstanter för PHD-filtren. Anropas av fusionsnoden en gång innan simuleringen startar.
  Parameters that could be changed:
  N – number of particles per estimated object. Large N gives a more accurate estimation of the distribution. Recommended values: 100 - 1000.
  TERRANG_V – weight for different kinds of terrain type. The ratio v1/v2 determines how much more likely a vehicle is to be in terrain type 1 than in type 2. It is a vector with four elements: road, forest, field, water/house. Recommended values: [10 0.1 1 10^-100] (vehicles almost always on roads), [3 0.3 1 10^-100] (vehicles often on roads, sometimes in forest), [2 0.5 1 10^-100] (slightly more likely to be in forest).

- timestep_track
  This is the main procedure in Track. It is called by the Fusion Node once per time-step, currently every 5 seconds.
  Calls report_local_format to convert reports to internal forma. Then calls tidssteg with reports, and if aggreagation has been performed also calls tidssteg with platoons and companies. The particles that are output from tidssteg are converted into the Fusion Node format using particles_global_format.
  Input: Reports and output from Aggregation from the last time-step..
  Output: Lists of particles for the different levels, and histograms representing the PHD of vehicles, platoons, and companies.
- report_local_format
  Transforms a report from the format used in the fusion node to an internal format more suited for use in this module.
  Input: Report in format used by Fusion Node.
  Output: Report in local format.
- particles_global_format
  Transform a list of particles from internal format to the format used in the Fusion Node.
  Input: List of particles in local format.
  Output: List of particles in global format.
- Tidssteg
  Propagates a PHD-filter one time-step. There are three separate PHD-particle filters, so three different instantiations of tidssteg are used. Calls montecarloMotion to propagate old particles one time-step, montecarloObs to

create new particles based on old observations, likelihood to compute the likelihood of each new particle given the new observations, and montecarlo to do a monte carlo sampling.
Input: Reports on observed objects's positions and velocities and their uncertainties. The filter is independent of the origin of the reports, but has access to information on probability of detection, that is, how often the sensors or Aggregation will fail to detect an object. The filter also uses information on terrain when it propagates particles.
Output: List of particles and a 2D histogram representing the PHD. The PHD gives information on where objects are likely to be, and also on the expected number of objects. It does not give information on object identities.

- montecarloMotion
  Propagates particles one time-step, using terrain info.
  Input: List of old particles, length of time-step.
  Output: List of propagated particles.
- montecarloObs
  Introduces new particles from an old observation and propagates them one time-step using terrain info.
  Input: Old observation, length of time-step
  Output: Propagated particles..
- likelihood
  Computes the likelihood of each particle belonging to a given observation.
  Input: List of particles, observation.
  Output: likelihood for each particle.
- Montecarlo
  Given a list of random seeds, this function finds the corresponding indices of a cumulative distribution. It is used for monte carlo sampling of a list of particles. The sampling entails the following steps:
  - Normalize the probability distribution over the list of particles. (The normed probability for index i is the probability that particle i is selected in each sample.)
  - Calculate a cumulative distribution, such that its last index has value 1. (Particle i has a large probability if the difference between indices i-1 and i in the cumulative distribution is large.)
  - Produce a random number x between 0 and 1.
  - Find (using this function) the first index in the cumulative distribution whose value is larger than x.
    Particle i has now been selected for the sample. Particles with large probability will be selected more often since more values of x will lead to their selection.
    Input: Cumulative distribution, random seeds uniformly distributed between 0 and 1.
    Output: List of indices in the cumulative distribution.

9.6.3 Taking sensor positions into account

The PHD filter has a probability of not observing a vehicle, p_FN. This probability varies depending on whether there is a sensor that covers the position of the vehicle. If there is no sensor that can see it, p_FN = 1. If there is a sensor that has the object in its field of view, p_FN varies depending on sensor type and how often it sends in reports to the fusion node. This represents what was previously called “negative reports”.

9.7 Documentation for Sensor Management module in IFD03

The aggregation module in IFD03 implements a simple version sensor managenemt based on random set simulation. It has not yet been described in any published work. It is called from the Control module. The interface consists of the function select_uav_plan which is called from do_sensor in the Control module. It is given a list of unit positions as input. The positions are converted to node adresses in the road network that is hard-coded in the implementation.
The module then simulates several possible futures using a (also hard-coded) transition matrix. It produces artificial observations of these futures and uses them to simulate a tracking filter. The difference between the filter and the "real" future is then used to determine the fitness of seveal (hard-coded) sensor control plans. Metrics used are either local or global; global means it uses positions at all times, while local only use the end positions. Metrics can also be either entropic (ie, we want to be as sure as possible of the location, regardless of whether it is true or not) or they can be simply the L1 distance between truth and filter.
The module returns the uav-flight-path that is best when the metrics are averaged over a number of possible futures. It also returns a quality, that is an indication of in how many futures that the chosen sensor plan was best.

For more details, see the local and global call graph in the html-documentation and the detailed list of functions below.

- **select_uav_plan** Return which sensor control scheme (among a set of hard-coded plans) to use. This function first maps the unit positions given to it to nodes in the road-network (which is hard-coded in the subfunction getsjopassetdata -- change this function in ordet to use a different network). It then calls select_sensor_scheme for each of the starting positions, and determines which sensor scheme is best for that. The plan that is best in total is returned and also saved in order to be plotted in the 3rd screen visualization.
- **findnearestnode** This function takes a lat-lon position and a list of nodes and returns the index for the node that is closest.
- **select_sensor_scheme** The major function in the sensor adaptation method. This function creates a number of future historys of the system. For each of these, the imaginary observations using a given sensor scheme is calculated and used to propagate a simple filter. The fitness of each plan for each alternative future is then calculated using 4 different metrics, and a matrix giving the best plan for each future and metric is returned.
- **calculatey** Propagates the simulated filter using a set of observations.
- **entropy** Returns the entropy of the input vector.
- **sumofcolumnentropies** Given a matrix A, returns a vector giving entropies for each column in A.
- **getobservations** Returns the set of observations of a given future that would be made using a specified sensor plan.

- **getpath** Given a transition matrix, this function returns a possible future path of a unit.
- **makepathtomatrix** Converts a path given as a list to a matrix.

9.8 Display module

This module is used for development and could be extended to be used for debugging. See online source-code documentation for details.

9.9 Documentation on miscellaneous functions in IFD03

- print_time provides controlled debugging output. It prints the current time, memory information, and an optional message.
- dprint is a function that can be used for debugging. It evaluates and prints two expressions.
- get_global_mem_status finds memory information from Windows routines.