DGIIIAI – Intelligent Delta Glider

for the Orbiter Space Flight Simulator

Pablo Edronkin, 2013 - 2019

https://orcid.org/0000-0001-8690-7030

Abstract

DGIIIAI is a variant of the stock Delta Glider III included with the Orbiter Space Flight Simulator[6.17.] that incorporates an expert system and artificial intelligence engine based on SQL and a relational database. While AI projects based on expert systems might look out of fashion, the fact is that they do have several advantages over other artificial intelligence technologies in specific realms.

Moreover: the design of this project in which rules and facts related to the expert system knowledge base, being expressed as SQL snippets contained in a relational database as data, and also being the fact that SQL is used to manipulate the data within the knowledge base, turns the said SQL snippets into homoiconic code.

This means that new rules not only can arise from programmers or user input, but that the system can generate its own rules using whatever AI means the developer might want to add to the system. In this regard, adding neural networks, using deep learning techniques or whatever to modify or expand the knowledge base is perfectly feasible.

As a collateral benefit, the integration of a relational database system within Orbiter makes it easier to develop new modules able to exchange data between each other in an easy, standardized way.

The project is targeted towards developers interested in AI systems for OSFS and not the general user. Its goal is to ultimately produce a semi or nearly – autonomous spacecraft model for OSFS.

Keywords

AI, simulation, artificial intelligence, expert systems

Acknowledgements

This project could not have taken place without the work done by:

- Dr. Martin Schweiger for developing Orbiter[6.17.].
- The developers of Sqlite3[6.18.] and OpenMP[6.8.].
- The Orbiter community of users and developers.

License and info for contributors

Please read the following files included with this project:

- README.md: contains info on setting up your system and installing the files.
- CONTRIBUTING.md: if you wan to contribute to this project.
- COPYING: for license information.

Table of Contents

1. Introduction

Since the first version of Orbiter Space Flight Simulator was published many years ago, hundreds of add - ons had been developed by enthusiasts, designers and researchers, but so far they were lacking in two aspects with regards to what this project intends to achieve:

1.1. Intelligence

The essentially made no use of artificial intelligence.

1.2. Communications between modules

Data exchange between modules and programs that take part in each simulation session has been rather modest.

In the case of **[1.1.]** the cause lies probably in the fact that AI demands a lot of work to get it done right and usually enthusiasts do not have the time and resources to achieve that. In the case of **[1.2.]** while there have been some attempts in this regard they suffer limitations.

It would be incorrect to pretend that DGIIIAI would be a solution for everyone and everything but it has a twofold potential: first, since the capabilities of the expert system can be expanded by developers and even able users, the ship has essentially limitless “intellectual growth” potentiality. And second, since it does incorporate AI and “data link” capabilities by means of its SQL - based expert system[6.4.] that also allows for inter - module communication, it could work external modules far more sophisticated than those used in a singe user, singe – computer session: you could use modules that run calculations on HPC clusters, for example, as long as they are able to communicate via the database with DGIIIAI.

It is important to stress that this is a complex add – on that still requires fine tunning and development. However, for a quick demo of the project, jump to section **[5.]**; for everything else, read on.

2. Conventions, caveats and base conditions of development

2.1. Origins

DGIIIAI is Based on DGIII OSFS 2010. The reason for this is that development of DGIIIAI started before OSFS 2016 was published and the code for the stock Delta Glider has been completely rewritten following a more object – oriented paradigm in the case of OSFS 2016. It made little sense to rewrite the code of DGIIIAI only for that reason.

2.2. Multi threading

Since DGIIIAI requires OpenMP libraries, a version of MSVS that is able to handle OpenMP is required. Free MSVS editions generally do not allow for that.

2.3. Terms and abbreviations used in this text

- AI: artificial intelligence.

- AI plan: artificial intelligence plan. DGIIIAI can use plan for activity and movement on the ground, in atmospheric flight or space. These plans can be manually created or using a planning module. At this point the planning modules is not available.

- AIE: AI Engine. It is comprised of the ES and the KB.

- CB: celestial body.

- DB: database.
- DGIII: the original stock Delta Glider.
- ES: expert system. Interprets the data contained in the KB.
- HPC: High performance computing.
- KB: knowledge base. The database of rules and facts interpreted by the ES.
- ML: machine learning.
- NN: neural network.
- OSFS: Orbiter Space Flight Simulator.
- RBML: Rule based machine learning.
- RCB: reference celestial body.
- RDB: relational data base.
- RDBMS: relational DB management system.
- SQL: Structured Query Language.
- SQLC: SQL Control. The MFD designed as a parallel project to DGIIIAI in order to provide an interface to the AIE and its KB.

- VTOL: Vertical Take Off and Landing.

2.4. Target audience

DGIIIAI is not intended for the general Orbiter user. While it can be used by anyone as is, in order to take advantage of the possibilities of this particular ship it is recommended to have an above – average knowledge of the related concepts.

2.5. HPC

Since DGIIIAI uses OpenMP libraries it is highly recommended to have some understanding of HPC general concepts before attempting to edit the code.

2.6. RDBMS

General knowledge about AI and expert systems in particular is also recommended. Also, since the rules of the expert system are stored as SQL code strings stored in a relational database, knowledge of RDBMS is also required.

2.7. AI

Artificial intelligence is a complex topic in which even seemingly minor modifications could produce significant changes in the behavior of the system. Therefore, changes in existing components, the addition of new knowledge to the system, etc. must be comprehensively tested. It is better to work in small steps, testing the results each time, rather than attempt to make big steps and fewer tests.

3. Development

3.1. AI Language choice

Originally development of DGIIIAI was done around a Prolog - based knowledge base. Also considered was Lisp. However, Prolog is a very good choice for expert systems but its files are invariably of the flat variety and these do not work well for intensive querying: opening and closing such files concurrently often lead to data loss. Relational databases on the other hand, take care of data integrity and data blocking under concurrent queries.

In the case of Lisp, there are various different flavors of it. Lisp, more than a language, is a family of languages. It s a very good one, but too hard for most programmers. Thus, using Lisp in any of its two main forms – Common Lisp or Scheme – would pose a very steep learning curve in addition to the one already present in the case of any AI system. Plus, Lisp *per se* has the same problem as Prolog regarding flat files, and while both can use to different levels of success hooks to DB systems, that adds an additional layer of complexity.

CLIPS[6.16.] was also considered, being it a mature and time – proven expert system framework and language. However, it also depends on flat files that are unsuitable for the kind of use predicted for DGIIIAI and it is based on an especially designed rule language language which could complicate future expansion of the DBS or the development of alternative uses such as data linking among modules.

However, SQL is essentially a cousin of Prolog. They were intended to solve similar problems at some point during their development, and both belong to the family of logical languages. In other words, it is essentially possible to solve with SQL all the problems that can be solved with Prolog.

Since DGIIIAI uses Sqlite3, the version of SQL used here corresponds to the Sqlite3 dialect, which is not totally compatible to SQL ISO/IEC (ANSI) standards[6.9.]. However, DGIIIAI follows standard SQL practices as much as possible because for whatever reason, this project in the future or other related projects might require a shift to a different RDBMS such as MySQL[6.14.].

In order to avoid unexpected obsolescence it is better to play on the conservative side. Installing a brand new RDBMS on a computer system, while not trivial as in the case of most software systems,

can be done. The problem starts when you want to make sure that the SQL code used with the prior platform works fine with the new one.

Every RDBMS on the market has its own quirks regarding SQL ISO/IEC standards so it is very unlikely that such a change would take place without any hassle, but by following standard practices bugs derived from differences in dialects might be minimized as much as possible.

3.2. Database choice

On a personal note, I consider Prolog to be more elegant and easy to use, but SQL and RDBMS offer a sound construction based on relational algebra, meaning that any properly-designed RDB precludes data loss and, as Prolog does, could be used to manage an AI system.

Moreover: since Orbiter has been written in C++ and there are several libraries that can be used to connect C++ programs to any major RDB in a much easier way than Prolog systems, an RDB could be combined with C++ to provide a new route for expansion of modules and capabilities for OSFS.

On these assumptions it was considered how to adapt RDB and C++ to produce a true KB and AIE.

First MySQL database was used as a test bed. MySQL worked well but required a full server installation and significant optimization to manage locking of data at record level. On tests, DGIIIAI reached a level of work as intense over the RDB as in a medium size company server, and it would get worse as more rules would be created on the KB. C and C++ libraries for MySQL are excellent.

PostgreSQL[6.15.] offered even finer control over records but was even slower than MySQL. This is an absolutely excellent product but on average machines and in the case of average users it would complicate things too much without any real benefit over the other two. C and C++ libraries for PostgreSQL are a little less flexible than those for MySQL but nevertheless good.

Lastly, Sqlite3[6.18.], while offering less granularity regarding blocking – it cannot block records inside a table individually – proved much faster and less resource hungry that the other alternatives. Speed up was palpable without the need of complex tests.

Also, it does not require any sort of installation or server configuration and copying, compressing and publishing systems using Sqlite3 is very simple. Perhaps a server-based RDBMS would prove better in some contexts, but at the present time the hassle of installing and running such a system for DGIIIAI doesn’t seem t be advantageous.

Native libraries for C and C++ are, however, inferior to those available for MySQL and PostgreSQL, There are third-party libraries available but since in the case of long - term projects such as this one, such libraries might be orphaned, meaning that the whole RDBMS could become obsolete and require a complete redesign using a different library; thus it was better to stick with the C libraries provided by the authors of Sqlite3, which are more low - level and cumbersome to use, but are safer in the long run. Thus, Sqlite3 became the final choice for this project.

3.3. Multi threading paradigm choice

DGIIIAI is multi threaded following a shared memory model. It needs a CPU with at least two cores to work properly. If you open *DeltaGlider.cpp* on your editor and take a look at the code of *DeltaGlider::clbkPostStep*. You will see:

#pragma omp parallel sections

This is OpenMP code and means that *clbkPostStep* is multi threaded: one thread is assigned to all the tasks that the function performs in a stock DGIII. The other thread is reserved for AI. In past versions of this project I experimented with multi threading in other functions as well, but the complexity of the code and its behavior grows exponentially as you add new multi threaded sections. So, I decided to just multi thread *clbkPostStep in toto* in order to find a balance between speed and code complexity.

OpenMP is arguably the most efficient way to achieve HPC operations on shared – memory systems. On the other hand, at least in this project I did not intend to modify the code for indistinct distributed or shared memory systems since it wouldn’t create enough benefits considering the cost of attempting to do so, given that the OS on which OSFS runs has not been designed to run efficiently – or at all possibly – on computer clusters.

It would be interesting, however, to have one such flight simulator one day.

3.4. Flight characteristics

Flight characteristics in DGIIIAI are the same as those of DGIII except for:

3.4.1. Landing gear retraction speed

The landing gear retraction and extension speed is faster in DGIIIAI. This produces less drag in approaches and departures in real aircraft and spacecraft because the landing gear is exposed less time to the atmospheric flow around the ship as it moves.

3.4.2. Artificial intelligence engine (AIE)

The AIE of DGIIIAI is essentially constantly turned on, which means that some vegetative functions such as yaw damping would be always active unless they are specifically turned off by the user or an user – developed program.

Therefore, regarding the flight characteristics and aspects of DGIIIAI other that the topics described int his text, please refer to the documentation of the original Delta Glider III, included with the files of this project.

4. Expert system

4.1. Basic concepts

DGIIIAI uses artificial intelligence based on an expert system model. Exsys is its name of the on-board expert system that DGIIIAI carries; it is an embedded version of an ES that was originally written in Prolog but later rewritten in C++. It is characterized by:

4.1.1. Database format

Instead of using flat text files, the knowledge base is written on top of a relational database.

4.1.2. Query language

Rules are written in SQL and stored as strings within the KB. As described in **[3.2.]** the main reason to select SQL over Prolog and Lisp was that flat files such as those used by the two latter languages to store data are unsuitable for intensive and concurrent access, while relational databases are precisely made for that. SQL is the language used to access the information, and it is the language in which the information itself is described. Since SQL belongs essentially to the same programming paradigm as Prolog, choosing the former over the latter was not problematic.

4.1.3. Homoiconicity of rules

Having its rules grouped in sets or programs that are dynamically loaded and unloaded as required. These programs, in turn can be grouped into chunks or programs of yet higher level of abstraction. The set of rules is homoiconic[6.5.], meaning that there is no difference between data and code or in other words, that code is data and data is code at the same time. Also, rules can be used to generate new rules or alter existing ones. I have experimented a bit with evolutionary algorithms to achieve that, but decided not to include that code in DGIIIAI to retain focus at least until rules for all flight regimes could be developed.

4.1.4. Learning capabilities

The expert system has some machine-learning capabilities already integrated, and there is no limit as to what it can learn, and given the stated in **[4.1.3.]** rules can create new rules, and these rules can use any ML method, neural networks, etc. to conform a true RBML system[6.6.][6.19.]. Exsys manages rules; how these rules come into existence is an independent issue.

4.1.5. Heuristics

The ES operates heuristically.

4.1.6. Data input

Parameters and data coming from the ship are stored in the KB as facts. Rules operate over those facts, make decision based on them and according to the applicable rules and in some cases, change the values of those facts that in turn, are sent back to the ship converted into C++ values.

4.1.7. Communication with other modules

The capabilities described at **[4.1.6.]** mean that the KB can also be used for inter process and inter module communication. For example, two or more MFD modules can exchange data between each other using the KB by accessing via SQL commands like SELECT, INSERT and UPDATE. This makes it theoretically possible to construct simulations that are more complex than one OSFS session running one ship. Instead, a simulation could be performed on various machines running each one a specific, more complex module that could exchange data with the ship running on the main server of the simulation network.

4.1.8. Knowledge base modularity

The whole KB is contained in one file, as is the case with every Sqlite3 database. You can exchange any KB completely just by replacing the database file by another one bearing the same name.

4.1.9. Accessing the knowledge base

You can access the data within the KB by using Sqlite3 from the shell or command window of your OS, by means of a database editor, or directly from within an OSFS session by using SQLC, which is an MFD designed especially for that purpose. SQLC is available as a separate download.

4.1.10. Importing data

The KB can be used in conjunction with external sources such as the Hipparcos catalog[6.11.] or Celestia files[6.10.] in order to produce new simulation scenarios on the fly, based on any sort of existing start system. Reading and storing data into a relational database from such sources is not problematic. Also ran some experiments in this regard using Celestia's flat files. Importing such data worked very well, but again, in order to focus and simplify thei project I decided not to include such capabilities in this version of DGIIIAI

Exsys essentially runs once on each *clbkPostStep* call. Since this function is called repeatedly during the simulation, so does the ES. On each iteration, the ES reads each rule contained in the table *sde_rules*, sees if the rules apply to the data contained in the table *sde_facts*, and modifies some of those facts or rules, deletes them or add new ones,so that in the next iteration the behavior of the system will be different.

It is important to stress that rules are loaded ad unloaded into *sde_facts* dynamically, but that doesn't mean that they are deleted from the system. Rules are kept in a long term storage table.

Table *sde_rules* only contains during any given iteration the rules applicable to those specific circumstances in order to increase performance. Each rule is comprised essentially of a condition and an action. If the condition is met, then the action is executed.

For example, if given a certain speed an overspeed condition is met, based on the speed data contained in *sde_rules*, plus the maximum speed allowed for the flight condition in which the ship is in, then an action implying a reduction of thrust level might be applied and the corresponding value in *sde_facts* is modified. On the final part of the iteration, the new values are passed to the applicable OAPI functions as arguments and hence, thrust is reduced.

4.2. AIE

The AIE is essentially divided in tow parts, which are the knowledge base and the ES; this ES started as an experimental, independent development written originally in Prolog and was called Exsys; later I wrote variants in C++, and this is the one embedded in DGIIIAI, another in R and more recently, in Scheme.

The knowledge base is a regular, relational database whose tables and fields within those tables are organized to store information in form of rules and facts. The ES provides those facts as values that are stored in the KB, reads the rules, decides whether each rule is applicable at any given moment, applies those rules an then sends the values corresponding to facts back to the OAPI.

To a great extent, the ES has been coded within *DeltaGlider.cpp* and *DeltaGlider.h*. You will find a couple of additional cpp and h files, which contain support and Sqlite3 – related functions. Aside from that, the code of the original Delta Glider has been left as it was.

4.3. Knowledge base

The KB is contained in one Sqlite3 file called *DGIIIAI.db*. Other RDBMS use different structures, files and locations depending on several factors. One of the advantages of Sqlite3 is that each database can be self - contained, but still can access other databases. This is the case of DGIIIAI: it

uses primarily the database *DGIIIAI.db*, but can use other databases as well. Indeed, using the SQLC MFD, DGIIIAI also accesses the MFD's own database called SQLC.

In this text we will not deal with SQLS and its database. For that you should review the documentation of that related but distinct project. In this case we well see how *DGIIIAI.db* is organized, beginning with the fact that it has several tables:

4.3.1. *sde_asoc_facts*

Useful for statistical analysis of selected facts and their values.

4.3.2. *sde_astro_cat*

Astronomic catalog.

4.3.3. *sde_data_dictionary*

Data dictionary for database DGIIIAI. It contains brief descriptions of all tables, programs and flight plans stored in the database.

4.3.4. *sde_experiments*

For experimentation purposes, currently concerning with the connection of DGIIIAI with a QPU in order to obtain true random numbers for simulation as opposed to the common pseudo - random number generators.

4.3.5. *sde_facts*

Contains facts of the expert system.

4.3.6. *sde_mem_facts*

Default facts and their values.

4.3.7. *sde_prg_rules*

Sets of rules (programs) that are loaded into *sde_rules* on demand.

4.3.8. *sde_prg_wpt*

Sets AI plans that are loaded into sde_wpt on demand.

4.3.9. *sde_rules*

Contains the rules in use at any given time.

4.3.10. *sde_wpt*

Contains the AI plans in use at any given time.

All these tables are important, but the most important ones are *sde_facts* and *sde_rules*. Most other tables are derivatives of these for special purposes.

When the ship starts up, Exsys purges the data contained in *sde_facts* and loads the default values stored in *sde_mem_facts*. Then, as rules start to be loaded and enforced, it modifies those values in *sde_facts* in real time.

A certain number of rules, identified by a field *Context = 'ship'* are always kept in *sde_rules*, no matter what. These rules are the most essential ones and include rules that load other sets of rules or programs, identified by a common *Context* value.

4.3.11. Structure of *sde_facts*

4.3.11.1. *Id*: primary key. This is an unique number that unequivocally identifies each record corresponding to a fact.

4.3.11.2. *Context*: identifies sets of facts. All facts with *Context = 'ship'* are permanently stored in this table.

4.3.11.3. *Status*: this field, in the case of this table indicates Exsys how it should handle each fact depending on the stage of each inference iteration.

4.3.11.4. *Item*: the name of the fact in question. Many facts have the same or very similar name as the arguments of OAPI because are internal KB representations of those arguments.

4.3.11.5. *Value*: the numeric value of the fact that corresponds to a given record.

4.3.11.6. Prob: the probability associated with the interpretation of a given fact. Normally this value is 1 (one), which means that the fact will be taken into consideration by the AIE every time.

4.3.11.7. *Asoc*: if this value equals 1, statistical analysis regarding the associated fact could be performed.

4.3.11.8. *sde_facts* record example: this is an actual example of the contents of one field in *sde_facts* (see also [**4.3.12.8.]**):

- Id = 1
- Context = ‘ship’
- Status = ‘sentodb’
- Item = ‘is_first_iteration’
- Value = 582
- Prob = 1
- Asoc = 0

Id = 1 means that this is the first record contained in the data table. *Id* is always the primary key in the tables of this KB.

Context = 'ship' means that it is a fact – i.e. a record – that will not be deleted from the table, no matter what. Its *Value* field may change, but the record will stay.

Status = 'sentodb' means that the first step of the data gathering from the OAPI has been completed and this value is ready for interpretation.

Item = 'is_first_iteration'. This is one of the counters that is incremented on each interpretation iteration made by the AIE. Once each iteration is completed, the *Value* associated to the fact identified by this Item name will be incremented.

Value = 582, means that 581 iterations have been performed and the current one is number 582. This is important because the value associated to the *is_first_iteration* fact is used to load in sequence various sets or programs of rules required, among other things.

Prob = 1 means that this fact will always be taken into account.

Asoc = 0 means that no stats will be calculated based on this particular fact.

4.3.12. Structure of *sde_rules*

4.3.12.1. *Id*: primary key of the table.

4.3.12.2. *Context*: identifies sets of rules grouped into programs. If *Context = 'ship'* it means that the rule is permanently stored in *sde_rules*. Rules with on the *Context* values will be permanently stored in *sde_prg_rules* and inserted or deleted from *sde_rules* according to the decisions of the AIE. This lowers significantly the workload of the system, given that many rules include fairly complex SQL code in some cases, and that takes quite a toll on the resources of the system.

4.3.12.3. *Status*: this field indicates to the AIE what to do with each rule. If *Status = 'enabled'* the the AIE will read the rule and attempt to enforce it.

4.3.12.4. *Condition*: states the condition that needs to be met. The SQL code snippets stored in this field should return 1 if the rule is to be applied.

4.3.12.5. *Action*: if the result of the query stated in the field *Condition* is 1 (one), then the code snipped contained in the field *Action* will be executed.

4.3.12.6. *Description*: a verbal, brief description of each rule. This makes it easy to understand what each rule is about, without having to decipher what the SQL code does.

4.3.12.7. Prob: a value of 1 (one) means that the rule will be applied with certainty.

4.3.12.8. *sde_rules* record example:

- Id = 1
- Context = 'ship'
- Status = 'enabled'
- Condition = "SELECT Value FROM sde_facts WHERE Item = 'is_first_iteration'"

- Action = “UPDATE sde_facts SET Value = ((SELECT Value FROM sde_facts WHERE Item = 'is_first_iteration') + 1) WHERE Status = 'applykbrules' AND Item = 'is_first_iteration'”

- Description = “Prg0.0. Ship level definition. Update iteration counter.”

- Prob = 1

Id = 1 means that this is the first record contained in the data table. Id is always the primary key in the tables of this KB.

Context = ‘ship’ means that it is a fact – i.e. a record – that will not be deleted from the table, no matter what. Its *Value* field may change, but the record will stay.

Status = ‘enabled’ means that this rule is active.

Condition = “SELECT Value FROM sde_facts WHERE Item = 'is_first_iteration'” will look for the value of the *Value* field in the record of sde_facts that corresponds to *Item = “is_first_iteration”*. In other words, it will extract the value of a counter. Since this particular one is a tautology, it will always be executed. In the cases of other rules, that might not be the case.

Action = “UPDATE sde_facts SET Value = ((SELECT Value FROM sde_facts WHERE Item = 'is_first_iteration') + 1) WHERE Status = 'applykbrules' AND Item = 'is_first_iteration'” is the query that will be performed if the *Condition* query is satisfied. As the *Condition* code is a tautology, the initial condition will always be satisfied and hence the action will be executed. The *Action* query, in this case, will increase the value of fact *“is_first_iteration”* by one unit. In other words, this rule is a simple counter.

Be extra careful when attempting to modify any rule because even seemingly minor changes can cause significant changes in the behavior of the ship. The rule shown as an example is a simple one and can be easily understood, but some rules can get more complicated and their implications much less apparent.

To illustrate this point and others, please take a look at this particular rule (only relevant fields are shown in this case.

...

- SELECT D.Value FROM (((SELECT A.Value FROM ((SELECT Value FROM sde_facts WHERE Item = 'thgroup_att_pitchup') AS A JOIN (SELECT Value FROM sde_facts WHERE Item = 'thgroup_att_pitchdown') AS B) WHERE ((B.Value = A.Value) AND (A.Value != 0.00))) AS C JOIN (SELECT Value FROM sde_facts WHERE Item = 'mode_crs' AND Value = 1) AS D) JOIN (SELECT Value FROM sde_facts WHERE Item = 'navmode_hlevel' AND Value = 0) AS E)

- UPDATE sde_facts SET Value = 0.00 WHERE Status = 'applykbrules' AND (Item = 'thgroup_att_pitchup' OR Item = 'thgroup_att_pitchdown')

- Prg31.4 definition. Mode crs. Cut off *att_pitchup and *att_pitchdown when their values are equal, mode crs is on and navmode hlevel is off.

...

As you can see, the condition of this rule involves nested JOIN operations along with the use of virtual tables. Reading this kind of code can be difficult, but SQL snippets can and get even more complicated.

This is why *sde_rules* should not be filled with all sort of rules at once but only those that are required at any given time. JOIN operations are resource – intensive but are very frequently used in relational database systems. Consider *sde_rules* as a working or short – term memory device, and *sde_prg_rules* as a log – term storage, like our brains has: we held a few things on our minds consciously while we are doing any task but indeed, keep in our deep memory much more.

Also you can see that in the *Action* field that *Status='applykbrules'* is mentioned. This means that the rule will only be executed when the facts with the required Item names have that status, which corresponds to the step in which the system applies the rules. The rule cannot be enforced at any other time. This is a safety measure to make sure that data has been gathered from the OAPI and external modules such as SQLC first.

And lastly, you will see that the string of the *Description* field starts with "*Prg31.4 definition. Mode crs...*" and something called modes is mentioned.

Prg31.4 stands for program 31, version 4. In the same folder as *DGIIIAI.db* you will find a number of .sql files. You can open those with a simple text editor. These contain the source code of each rule corresponding to each program and its different versions.

Those files are to write the rules grouped on each program, and then copy them to *sde_prg_rules*. There are different versions for almost every program, and all of them are kept in the long – term storage. Since even small changes can have very large effects, this practice helps roll back to a previous, well – behaved version of a program in the event on an error in the code.

If you try to make changes or add rules, it is better to do so using a new version of the applicable program(s). You can configure what programs are active by creating or modifying the *Value* field of the corresponding fact record in *sde_mem_facts*.

For example, in the case of program 31, you should look for (SELECT) the record that contains *Item = 'cur_ver_prg31'*. If you change the *Value* field from – say – 31.4 to 31.5, you will change the enabled version of prg31 from 31.4 to 31.5 and the AIE will load *prg31.5* instead of *prg31.4* when prg31 is required.

If you take a look at another fact record from *sde_mem_facts* with *Item='load_order31'*, you will see that *Value = 0*. This means that the program or chunk of rules has no synchronized loading requirement. It will be loaded orn unloaded to *sde_rules* as requested by other rules or by the user, by means of the SQLC MFD. If you find any load order fact with a value not equal to zero, that means that the associated program will be loaded on the inference iteration of the ES equal to that number. The inference iteration number is the value of fact is_first_iteration, which was also mentioned in an example (see **[4.3.11.8.]**).

As you can probably see, this particular rule sets to zero values related to the pitch of the craft in relation to something called mode crs. This is a mode that manages cruise flight in an atmosphere. Setting on an off various modes loads and unloads several programs. A mode is an abstraction layer on top of programs, which in turn are on top of individual rules, conceptually speaking.

4.3.13. General description of facts

Facts are representations of OAPI arguments as well as internal parameters of the AIE within the KB. Think of them as variables stored in a database. Most of them can be classified in groups and understood in their meaning and function based on the nature of those groups.

We will use SQL snippets here for identification of fact groups.

4.3.13.1. In general, facts representing actual OAPI arguments carry names that are the same or almost the same as the names of those arguments. This has been done in order to ease recognition of the function of as many facts as possible.

4.3.13.2. *Fact LIKE 'mode%'* - These facts manage various modes. If *Value = 1* then the mode is on, otherwise is off.

4.3.13.3. *Fact LIKE 'thgroup%'* - Corresponds to the thgroups of OAPI.

4.3.13.4. *Fact LIKE 'tgt%'* - These facts represent targets to be achieved or values to be set as attributes of other facts. For example *tgt_hdg* is a value to be passed to fact *hdg*.

4.3.13.5. *Fact LIKE 'factor%'* - These contain values that are used as operands in conjunction with other values. They are generally used to improve results of calculations by using machine learning algorithms. For example, a factor can be used to counter small defects in calibration for trim surfaces, something that could change from flight to flight. In such cases, the AIE first reads raw data for a while, compares it to what should be expected and then calculates an applicable *factor*.

4.3.13.6. *Fact LIKE 'load_order%'* - *load_order* facts determine on which iteration of the AIE inference, as defined by fact *is_first_iteration*, a program is loaded. If a *load_order* factor for a

given program number has *Value = 0*, then it is not automatically loaded based on the *is_first_iteration* fact.

4.3.13.7. *Fact LIKE 'delete_order%'* - delete*_order* facts act similarly to *load_order* facts, but instead of meaning when a program is loaded, they ensure that a program will be deleted from *sde_rules* at the corresponding iteration.

4.3.13.8. *Fact LIKE 'cur_ver%'* - *cur_ver* facts determine the current version of a program. As a program is updated, all versions are kept in *sde_prg_rules* but only the current version of each program can be loaded into *sde_rules.* Changing the cur_ver fact value of a given program implies that another version of the said program will be loaded into *sde_rules*.

4.3.13.9. *Fact LIKE 'navmode%'* - These are associated to the OAPI navmodes. If the user or AIE sets the value of a given *navmode* fact to 1, the corresponding navigation mode in the ship is activated.

4.3.13.10. *Fact LIKE '%go_%'* - Used for planning. Each one of these facts, when *Value = 1* tell the AIE that conditions are met for a specific step in an aiplan.

4.3.13.11. *Fact LIKE '%crew%'* - Related to crew management.

4.3.13.12. *Fact LIKE '%max%'* - A maximum or highest value allowed.

4.3.13.13. *Fact LIKE '%min%'* - A minimum or lowest value allowed.

4.3.13.14. *Fact LIKE 'delta%'* - Delta facts contain the results of a difference between the values of other facts.

4.3.13.15. *Fact LIKE '%quantum%'* - Related to values obtained from real or simulated QPU.

4.3.13.16. *Fact LIKE '%rst%'* - Resets values from associated facts.

4.3.13.17. *Fact LIKE '%prg%'* - Related to program management.

4.3.13.18. *Fact LIKE 'submode%'* - Submodes (see **[4.4]**).

4.3.13.19. *Fact LIKE '%alt%'* - Related to altitude.

4.3.13.20. *Fact LIKE '%abs%'* - Contains an absolute value, i.e. given a real number r, *Value* = $|r|$.

4.3.13.21. *Fact LIKE '%avg%'* - Contains a mean averaged from the *Value* of another fact record.

4.3.13.22. *Fact LIKE '%err%'* - Related to error or deviation calculations, generally in association with ML to calibration of values.

4.3.13.23. *Fact LIKE '%_t'* - Related to trim.

4.3.13.24. *Fact LIKE '%_l'* - Related to left or port side.

4.3.13.25. *Fact LIKE '%_r'* - Related to right or starboard side.

4.3.13.26. *Fact LIKE 'l_%'* - Lights or illumination.

4.3.13.27. *Fact LIKE 'v%'* - Generally related to standard aerospace v speeds.

4.3.13.28. *Fact LIKE '%att%'* - Related to attitude control.

4.3.13.29. *Fact LIKE '%_x'* - Related to x axis.

4.3.13.30. *Fact LIKE '%_y'* - Related to y axis.

4.3.13.31. *Fact LIKE '%_z'* - Related to z axis.

4.3.13.32. *Fact LIKE '%_t'* - Related to t axis or time.

4.3.13.33. *Fact LIKE '%_d%'* - Related to axis corresponding to dimensions higher than four. *_d* should be followed by a number indicating the dimension in question (example *_d5*).

4.3.13.34. *Fact LIKE '%counter%'* - Counter. Can be incremental or decremental. Counters usually are used in the AIE to synchronize programs.

4.3.13.35. *Fact LIKE '%hato%'* - Horizontal atmospheric takeoff operation.

4.3.13.36. *Fact LIKE '%halo%'* - Horizontal atmospheric launch operation.

4.3.13.37. *Fact LIKE '%hvto%'* - Horizontal vacuum takeoff operation.

4.3.13.38. *Fact LIKE '%hvlo%'* - Horizontal vacuum launch operation.

4.3.13.39. *Fact LIKE '%vato%'* - Vertical atmospheric takeoff operation.

4.3.13.40. *Fact LIKE '%valo%'* - Vertical atmospheric launch operation.

4.3.13.41. *Fact LIKE '%vvto%'* - Vertical vacuum takeoff operation.

4.3.13.42. *Fact LIKE '%vvlo%'* - Vertical vacuum launch operation.

4.3.13.43. *Fact LIKE '%wpt%'* - Waypoint.

4.3.13.44. *Fact LIKE '%upd%'* - Update.

4.3.13.45. *Fact LIKE '%plan%'* - AI plan.

4.3.13.46. *Fact LIKE '%learn%'* - Related to machine learning.

4.3.13.47. *Fact LIKE '%onb%'* - On board.

4.3.13.48. *Fact LIKE '%proviso%'* - Transient, used momentarily for development purposes.

4.3.13.49. *Fact LIKE '%limit%'* - Value of the maximum deviation acceptable from a reference fact. Given a fact *f*, *f_min_limit* would be *f – limit* and *f_max_limit* would be *f + limit*. *f_min_limit* and *f_max_limit* could be implicit or explicit facts.

4.3.13.50. *Fact LIKE '%keep%'* - Describes facts, programs, etc. whose values should be kept as they are. Programs with an associated, explicit *keep* fact should be left loaded on *sde_rules* as long as *Value = 1* for the said *keep* fact record. It can be said that rules with *Context = 'ship'* have implicit keep facts associated to them.

4.3.13.51. *Fact LIKE '%make%'* - Create. These modes are used to create new programs.

4.3.13.52. *Fact LIKE '%quadrant%'* - Refers to the four quadrants defined around the ship relatively from its geometric center, each one of them defined by a square angle. Quadrant 1 is defined as the angle interval [0:90), quadrant 2 is [90:180), quadrant 3 is [180:270) and quadrant 4 is [270:360). 360 is always converted to 0 degrees. Since the AIW works heuristically, any value corresponding to *tgt_hdg* is interpreted as being located in any of the four quadrants, and the decision to turn and appropriate rules depend on that. For example, if the ship has *hdg Value = 0* and *tgt_hdg Value = 120*, it will turn to starboard, since the turn will be smaller (120 degrees) as opposed to turning left (180 + 60 degrees).

4.3.13.53. *Fact LIKE '%debug%'* - Used for debugging purposes.

4.3.13.54. *Fact LIKE '%test%'* - Used for test purposes; bear in mind that a test not necessarily implies debugging (see **[4.3.13.53.]**), but all instances of debugging constitute tests.

4.3.13.55. *Fact LIKE '%turn%'* - Related to navigation, turns.

4.3.13.56. *Fact LIKE '%equ%'* - Equatorial.

4.3.13.57. *Fact LIKE '%rel%'* - Relative.

4.3.13.58. *Fact LIKE '%ecl%'* - Ecliptic.

4.3.13.59. *Fact LIKE '%def%'* - Defined. Facts with this characteristic have configurable values.

4.3.13.60. *Fact LIKE '%hdg%'* - Related to heading. *tgt_hdg*, for example, is a fact that contains the value of an intended or desired heading. Either the user, via an interface such as SLQC, or the AIE, by setting this fact's value to a given number, will cause the ship to turn and acquire the target heading.

4.3.13.61. *Fact LIKE '%turn%'* - Related to turns and course changes.

4.3.14. General description of rules

Much like in the case of facts, rules can be understood based in the way that they are constructed.

4.3.14.1. Rules have a long term storage – table *sde_prg_rules* – and a process or short term storage – *sde_rules*. Only a few rules are held permanently on *sde_rules*. These are identified by *Context = 'ship'*. These rules are the ones required to manage the expert system itself, aside from the management of the ship's systems, and they should not be deleted. Copies of these rules are written also in the file *Prg0_0.sql*.

4.3.14.2. Essentially, each rule has a condition and an action stored in fields Condition and Action respectively in sde_prg_rules and sde_rules according to **[4.3.14.1.]**.

4.3.14.3. For each program stored in *sde_prg_rules* there is a corresponding .sql file as backup. No longer current program versions should not be deleted from the KB. Instead, the corresponding *cur_ver** fact value should be updated. Rules that should not be used can also be disabled by changing their *Status* field to *Status = 'disabled'*. If a new version of a program is created, the corresponding .sql file should be created as well following the format and patterns used in the existing .sql files.

4.3.14.4. For a rule’s condition query contained in the *Condition* field, at least one valid record or result should be produced.

4.3.14.5. Loading programs to *sde_rules* and unloading them is handled solely by means of facts belonging to *submode_prg** for consistency and predictability.

4.3.14.6. Rules can create, edit or delete rules and facts.

4.3.14.7. The AIE applies rules sequentially on each iteration loop by reading every record in sde_rules, analyzing the *Condition* field and executing the snipped in the *Action* field if it is applicable.

4.3.15. Adding, editing and deleting records from the KB

It s possible to add, edit or delete records from any table in the KB in various ways:

4.3.15.1. Issuing SQL commands manually, using a database editor compile with Sqlite3.

4.3.15.2. Using a database editor with graphical capabilities, much like editing a spreadsheet. This option is the one that would be used when, for example, a new program is being loaded into the KB by the user.

4.3.15.3. Issuing SQL commands via SQLC. This MFD essentially works by activating and deactivating modes or passing argument data directly to fact records. However, it also lets the user enter full SQL snippets of code directly from an OSFS session.

4.3.15.4. Activating suitable modes via SQLC or an external SQLC emulator, either those existing on the base file or developed by the user, that in turn work on the table records. This should be the preferred option for working with the KB. SQLC naturally would require to run an OSFS session, while an emulator would avoid that. It is worth noticing that at this point, an SQLC emulator does not exist but one can be relatively easily developed using Gexsys[6.2.].

Also, some caveats should be considered:

4.3.15.5. Id fields in all tables are managed by the RDBMS; normally users should not update them.

4.3.15.6. By default set *Prob = 1*, *Asoc = 0*, *Status = 'enabled'*, *Context = 'ship'* unless specified otherwise.

4.3.15.7. Do not leave fields empty or with NULL value.

4.3.15.8. Do not delete records with *Context = 'ship'*.

4.3.15.9. The structure of *sde_rules* is the same as the structure of *sde_prg_rules*, but not records.

4.3.15.10. The structure of *sde_facts* is the same as the structure of *sde_mem_facts*, as well as the records, so any INSERT or DELETE operation performed should be performed on both of them if it implies modifying the field Item or adding or deleting records.

4.3.15.11. To avoid as much confusion as possible, use the programs contained in the KB for addition, edition or deletion instead of attempting to so manually.

4.3.15.12. SQL code in the KB can be self - modifying; this can be achieved on the fly[6.7.] and if fact is used extensively during each inference iteration. Programs no longer in use are downloaded as required, other programs are loaded, *sde_facts.Value* field is constantly updated and ML adjusts several parameters. Some *sde_rules.Action* SQL snippets can even modify *sql_rules.Action* snippets as well.

However, care should be taken in the case of self - referential modifications done on the fly, since unfitness of the newly – generated rules could render the system unusable. There are no fitness – based vetting programs on the system at this time, and self – modification should be considered at this moment both as amn opportunity as well as a vulnerability of the system.

4.3.16. General description of *sde_wpt*

This table contains data about waypoints. This data can be entered by the user or by means of an aiplan (*.pln file).

4.3.16.1. *Id*: primary key of the table. Note that there are other keys in this table, which are *Num* and *Next*. Those have an ostensibly equal but really different functions as this field. *Id* works as the primary key of the relational table in order to fulfill the first normal form, while *Num* and *Next* are used to sort waypoints in aiplans.

4.3.16.2. *Context*: identifies to which aiplan belongs each record.

4.3.16.3. *Status*: if this filed holds the string ‘enabled’ then the record corresponding to a given waypoint is still on the aiplan and its data has not been committed to the AIE for processing.

4.3.16.4. *Num*: key value that identifies each waypoint. This is not the same as the *Id* field that identifies each record, since waypoints from different aiplans could be loaded in different ways and hence, a sequential processing of each waypont based on the *Id* value becomes unreliable for that specifi purpose.

4.3.16.5. *Lat*: latitude of the target waypoint as projected over the surface of the RCB.

4.3.16.6. *Lon*: longitude of the target waypoint as projected over the surface of the RCB

4.3.16.7. *Alt*: altitude required at the waypoint over the mean or surface level of the RCB.

4.3.16.8. *Dist*: distance to move with the bearing expressed as the value of *Brg* field. This can be used to calculate target *Lat* and *Lon* values.

4.3.16.9. *Time*: time to move with the bearing expressed as the value of *Brg* field. This can be used to calculate target *Lat* and *Lon* values.

4.3.16.10. *Speed*: this value will be passed to the *Value* field of fact *tgt_speed*.

4.3.16.11. *Action*: if its value is zero, execution of the aiplan will stop at the related waypoint.

4.3.16.12. *Brg*: this value will be passed to the *Value* field of fact *tgt_speed*.

4.3.16.13. *MaxRadius*: the AIE will make corrections to reach the waypoint defined by *Lat*, *Lon* and *Alt*. *MaxRadius* determines the sphere of acceptable error around the waypoint, meaning that once the ship is within that sphere with regards to the waypoint, it will stop trying to navigate to the point and instead, jump to the next waypoint as determined by the *Next* field.

4.3.16.14. *Next*: describes the order of execution. After the current waypoint *k*, the AIE will always attempt to process a given record *n* given that $n >= k+1$, so that $Next_k \leftarrow Num_n$ after sorting the aiplan records ascending by *Num* value and eliminating those records with *Status != 'enabled'*. In other words, the records corresponding to an aiplan will be sorted by their *Num* values, then those with *Status = 'enabled'* will be SELECTed and each record listed on the SELECTion will be assigned a *Next* value equal to the *Num* value of the record immediately following it on the list obtained after executing the SELECT query.

Also, all records receive A *Value = 1* (*sde_wpt.Value* $\leftarrow$ *1*) except the last one, whose value gets *Next = 0*, *Action = 0* and therefore *sde_wpt.Value* $\leftarrow$ *0*, given that there is no waypoint registered after it. That last waypoint indicates the termination of the aiplan, since a *Value = 0* will deactivate the mode specified in the field *Item*.

4.3.16.15. *Item*: item of *sde_facts* to be manipulated – usually *mode_wpt*. This means that reading data from waypoints can trigger *per se* actions of the AIE. Since each time a waypoint is reached *mode_wpt* deactivates itself because it has no waypoint ot follow, if an aiplan has several legs to follow, *mode_wpt* needs to be restarted, or another mode that will in turn, reactivate *mode_wpt*.

4.3.16.16. *Value*: value to be assigned to the *Value* field of *Item* as per **[4.3.16.15.]**.

4.3.16.17. *sde_wpt* record example:

- Id = 3
- Context = ‘pln1.2’
- Status = ‘enabled’
- Num = 3
- Lat = 25
- Lon = 25
- Alt = 11000
- Dist = 100
- Time = 0
- Speed = 300
- Action = 1
- Brg = 25
- MaxRadius = 15
- Next = 4

- Item = ‘mode_wpt’

- Value = 1

Id = 3 means that this is the third record of the file.

Context = ‘pln1.2’ states that this record corresponds to aiplan 1, version 2.

Status = ‘enabled’ means that the waypoint is valid.

Num = 3 means that the record corresponds t the third waypoint in its aiplan.

Lat = 25 indicates a latitude of 25 degrees north; southern values are negative.

Lon = 25 indicates a longitude of 25 degrees east; western values are negative.

Alt = 11000 states that the waypoint is located at 11000 m above mean surface altitude. This will be passed to *sde_facts* so that *tgt_alt = 11000* .

Dist = 100 means that the ship will follow a leg of 100km with the heading stated in the field *Brg*. *Lat* and *Lon* will be recalculated.

Time = 0 means that time will not be used as a measure for the leg to follow.

Speed = 300 states that the indicated airspeed will be 300. This value will be passed to *sde_facts* so that *tgt_speed* = 300 .

Action = 1 means that the waypoint will be processed and a next waypoint will be followed after that.

Brg = 25 means that the ship will turn to 25 degrees. This value will be passed to *sde_facts* so that *tgt_hdg* = 25 .

MaxRadius = 15 means that once the ship approaches the waypoint at a distance of 15 km or less, then the AIE will consider that it has reached the waypoint and afterwards, proceed according to the values of *Action* and *Next*. A generous value for *MaxRadius* is required to ease navigation through non – terminal waypoints so that the ship can perform gentle turns to acquire a new heading for the following leg. This value could be reduced to increase precision in navigation, but in that case speed should be probably adjusted in order to diminish the radius of the turns required to alter course.

Next = 4 means that the waypoint to be processed after the current waypoint is reaches is that one identified by its field Num, so that Num = 4.

Item = 'mode_wpt' means that the fact identified as *'mode_wpt'* will receive the value specified in the field *sde_wpt.Value*.

Value = 1 means that in *sde_facts*, a fact identified by the same string as contained in the record that corresponds to the current waypoint in s*de_wpt.Item* will receive *sde_wpt.Value* as its own *Value*.

In principle, the ship can navigate both on land – for example, taxiing to a a runway or landing pad – or flying. Legs are calculated as great circle route segments. In order to navigate following an aiplan, *mode_wpt = 1* is required. This implies that waypoint properties are recalculated and the corresponding values are passed as new facts to *sde_facts*. Then, *mode_crs* will make the ship turn, change speed and acquire the required altitude.

4.3.17. General description of *sde_prg_wpt*

This table has the same structure as *sde_wpt*, but contains aiplan details that are not necessarily loaded into *sde_wpt* for immediate processing. It is a long - term storage of plans and in relation to *sde_wpt* works in a way analogous to that of *sde_prg_rules* with regards to *sde_rules*.

Aiplans are not expressed as SQL code snippets, but data sorted in rows and columns, each row corresponding to a record, and each column to a data field.

Aiplans stored in this table should note be deleted; instead, they should be rendered inactive by *sde_prg_wpt.Status ← 'disabled'*.

4.3.18. General description of *sde_astro_cat*

This file is a celestial catalog that at this point has no special function but in the future it will be used for navigational planning.

4.3.18.1. *Id*: primary key. Differently from the case of *sde_wp*t (see **[4.3.16.]**) a logical structure dependency tree is constructed using the Id field. In *sde_wpt* the *Id* fields plays a lesser role and fields *Num* and *Next* are used to establish dependencies between records. But *sde_astro_cat.Id* is vital. Therefore it is important to remember not to delete records. Instead, if a record should not be used then remember to set *Status = 'disabled'*.

4.3.18.2. *Context*: normally set to *'ship'*.

4.3.18.3. *Status*: only *'enabled'* records will be processed.

4.3.18.4. *Item*: CB or celestial structure name.

4.3.18.5. *Description*: a brief description of the CB.

4.3.18.6. *Parent*: *Id* number of the CB or structure from which the given CB depends or forms part of.

4.3.18.7. *Type*: CB type.

4.3.18.8. *Keywords*: keywords that might be used to search and find this particular CB.

4.3.18.9. *sde_astro_cat* example:

- Id = 8
- Context = ‘ship’
- Status = ‘enabled’
- Item = ‘Sun’
- Description = ‘Our sun.’
- Parent = 7
- Type = ‘star’
- Keywords = ‘NA’

Id = 8 is the primary key of this table with an eight as value. This means that this is the eight record in this table.

Context = ‘ship’ indicates that the record is universal for use by the ship and its sub modules. There might be records with a different context.

Status = ‘enabled’ means that the record is enabled and will be processed upon request.

Item = ‘Sun’ indicates that this record represents our sun.

Description = ‘Our sun.’ describes briefly the record.

Parent = 7 contains the Id value of the CB’s record to which the current record is subordinated as a child. In this case, record 8 – our Sun – is part of what record 7 represents – our solar system. In the case of binary systems we will have to star records with the same star system parent record.

Type = ‘star’ describes the kind of CB that this record represents.

Keywords = ‘NA’ means that this record does not have yet any associated keywords for catalog searching.

This table has been implemented only recently, but in the future it will be used to generate aiplans and even scenario and CB files for OSFS, and as an interface to external celestial catalogs.

4.3.19. General description of Prg*sql files

These are intermediate files used to write .sql files with programs to be stored in *sde_prg_rules*. The reason behind writing them in flat text files before is only a matter of convenience. You can write programs to be stored on *sde_prg_rules* entirely without the steps involving the Prg*sql files. However, it might be much more easier to see what you are doing in this fashion.

You will find these files inside the databases/DGIIIAI folder. They do not work in conjunction with the AIE as they are. You must copy each SQL snippet to *sde_prg_rules* first.

4.3.19.1. Structure of a Prg*sql. file:

4.3.19.1.1. ‘--’ Indicates a comment. A commented line should not be copied into *sde_prg_rules* except if it is a description for the Description field, which are also commented out. The reason for this is that in the text fields only pure SQL code is left uncommented so that editors capable of parsing SQL code can help you analyze the syntax and semantics of the code being written. Comments start with a double minus sign in SQL.

4.3.19.1.2. A line of minus signs divides two records.

4.3.19.1.3. N, being N an integer number, indicates the sequence order of the snippets of code that follow and correspond to a single record in *sde_prg_rules*.

4.3.19.1.4. The first SQL query after the line and the ordinal, usually a SELECT, corresponds to the *Condition* field, meaning that you should copy that query into the *Condition* field of a new record in *sde_prg_rules*.

4.3.19.1.5. The second snippet corresponds to the *Action* field. Usually consists of a fairly complex UPDATE, but can be a DELETE or INSERT snippet as well.

4.3.19.1.6. Following these two snippets and before a new line of minus signs comes a commented out string that describes what the rule does. This comment goes into the Description field of *sde_prg_rules*.

4.3.19.1.7. Aside from what was described above, other fields need to be taken into account when adding a Prg*sql file to *sde_prg _rules*:

4.3.19.1.7.1. *Id* is set incrementally by the RDBMS. Users must not interfere with the auto increment set by default for this field.

4.3.19.1.7.2. *Context* should receive the name of the Prg*sql file with no capital letters, just as it is in the first line of the example below(see **[4.3.19.2.]**).

4.3.19.1.7.3. *Prob* should be set to 1, unless the rule is a probabilistic one (not yet fully implemented at the time of this writing, so set it to 1 always, for now).

4.3.19.2. Prg*sql example:

-- prg1.0

-- Test program. Concept development. Not used in actual flight operations.

1

SELECT Value FROM sde_facts WHERE Item = 'submode_prg_counter' AND Value >= 2

UPDATE sde_facts SET Value = ((SELECT Value FROM sde_facts WHERE Item = 'test_fact_1') + 1) WHERE Status = 'applykbrules' AND Item = 'test_fact_1'

-- Prg1.0 definition. Test program that increases the value of test fact 1.

2

SELECT Value FROM sde_facts WHERE Item = 'submode_prg_counter' AND Value >= 100

UPDATE sde_facts SET Value = 1 WHERE Status = 'applykbrules' AND Item = 'submode_prg_rst'

-- Prg1.0 definition. Resets submode prg when submode prg counter reaches 100.

3

SELECT Value FROM sde_facts WHERE Item = 'submode_prg_counter' AND Value >= 100

DELETE FROM sde_rules WHERE Context = (SELECT ('prg' || CAST((SELECT Value FROM sde_facts WHERE Item = 'cur_ver_prg1') AS TEXT)))

-- Prg1.0 definition. Delete from sde facts records belonging to cur ver prg1.

It is possible to copy these Prg*sql files into *sde_prg_rules* in many ways. So far I have copied and pasted the data using a visual DB editor..

4.3.20. General description of pln*sql files

These are intermediate files that facilitate the edition if aiplans and their data.

You will find these files inside the databases/DGIIIAI folder. They do not work in conjunction with the AIE as they are. You must copy each SQL snippet to *sde_prg_wpt* first.

4.3.20.1. Structure of a pln*sql. file:

4.3.20.1.1. '--' Indicates a comment. Lines that are commented out in this way should not be copied into *sde_prg_wpt* (See **[4.3.19.1.1.]**).

4.3.20.1.2. Aside from what is specified in **[4.3.20.1.1.]**, each line in a pln*sql represents a whole *sde_prg_wpt* record and data must be copied according to :

4.3.20.1.2.1. *Context* ← Col 1.

4.3.20.1.2.2. *Status* ← Col 2.

4.3.20.1.2.3. *Num* ← Col 3.

4.3.20.1.2.4. *Lat* ← Col 4.

4.3.20.1.2.5. *Lon* ← Col 5.

4.3.20.1.2.6. *Alt* ← Col 6.

4.3.20.1.2.7. *Dist* ← Col 7.

4.3.20.1.2.8. *Time* ← Col 8.

4.3.20.1.2.9. *Speed* ← Col 9.

4.3.20.1.2.10. *Action* ← Col 10.

4.3.20.1.2.11. *Brg* ← Col 11.

4.3.20.1.2.12. *MaxRadius* ← Col 12.

4.3.20.1.2.13. *Next* ← Col 13.

4.3.20.1.2.14. *Item* ← Col 14.

4.3.20.1.2.15. *Value* ← Col 15.

4.3.20.2. pln*sql example:

```
--------------------------------------------------------------------------------
-- pln2.0
-- Test flight plan for scenario Test2.
--------------------------------------------------------------------------------

pln2.0 enabled    1  0   0    11000 100 0 300 1 35 10 2 mode_wpt 1

pln2.0 enabled    2 30  20   11000 100 0 300 1 30 10 3 mode_wpt 1
```

pln2.0 enabled 3 20 30 12000 100 0 300 0 25 10 0 mode_wpt 0

Both in the case of Prg*sql and pln*sql files it is recommendable to be very tidy and systematic. Plus, whenever a new version of the program is created, a specific, new file should be created for that programs as well.

4.4. Modes and submodes

In **[4.3.13.]** the existence of modes is described as an extra layer of abstraction on top of programs or SQL query strings that have a common context. The AIE itself works at the highest level with modes, activating and deactivating them by setting *Value = 1* or *Value = 0* respectively in the fact record corresponding to each one of them.

Modes allow the AIE to identify easily differences in the status of the spacecraft and its components. For example, certain conditions lead to the interpretation that the ship has a *Value = 1* for *Item = 'mode_crs'* in *sde_facts* meaning that the fact is that the ship is in cruise flight in an atmosphere.

Once *mode_crs* has *Value = 1* a lot of things star happening, certain programs are loaded, while others are unladed from *sde_rules*. Several programs loaded according to *Item = 'load_order*'* and *Value != 0* or included in any given mode, in turn, load other programs. In this fashion, situational awareness is managed at one level, modes are enabled or disabled accordingly, and then at deeper levels programs are loaded and rules enforced.

Therefore, situational awareness and the rules for mode activation or deactivation becomes detached from rules operating deeper. In this way, new modes can be defined, others can be redefined, or implemented as parts of other modes regardless of the actual implementation of other rules.

For example, *mode_hato* (horizontal, atmospheric take off) which implies taking off from a runway, and *mode_vato* (vertical, atmospheric take off), which VTOL operations, can be established as part of a departure procedure either by the user or the AIE, and both share some common operations such as getting the landing gear up at some point, or changing mode to *mode_crs* on route can share

several rules or programs, but are not bound to be the same and admit entire redefinitions of what they do in detail without having to redefine the modes themselves.

Submodes are modes within modes. One such submode is *mode_prg*: when it has *Value = 0*, it does nothing, but when *Value != 0* it calls a prg manager set of rules, which has several submodes.

Once the program is loaded into *sde_rules*, *mode_load_prg* deactivates itself. Since modes often require loading programs, *mode_load_prg* is defined as a submode that can operate within any mode. For example, *mode_hato* requires some programs to load, then *mode_crs* also needs loading several programs. In both cases, *mode_load_prg* is invoked as a submode of *mode_hato* and *mode_crs* respectively.

Think of modes and submodes as analogies of nested functions in any imperative programming language.

You can review all current modes in the project by performing the adequate SELECT operation on *sde_mem_facts* using any compatible database editor. As the project advances, usually more modes are defined.

4.5. Reasons for an expert system instead of neural networks

Expert systems seem to have fallen from grace within the realm of AI research, but they remain suitable for use in specific realms. For general AI neural networks seem to offer more flexibility, but specialized usage require clear rules, and expert systems can help in those cases and provide easier development and use than neural networks.

Both AI paradigms have their advantages and disadvantages, and both could have been used in the case of DGIIIAI. However, the ES paradigm is easier to implement because the ES itself could be developed separated from DGIIIAI, while a comparable NN could not. Exsys can conceivably be adapted more easily to other ships, but any NN system would have to be developed from scratch in such case.

This does not mean that NN cannot have a place within DGIIIAI: several NN modules could be used to modify or create new rules, to refine values associated to facts, etc. In fact, the AIE could be supplemented with any sort of ML algorithm or system.

In order to simplify compilation, linking and installation for users of DGIIIAI, this version only makes use of OpenMP and Sqlite3 libraries; however, past versions used a full, server – based MySQL database with record level locking in the KB files, Bayesian networks, SVM and other algorithms provided by the Dlib[6.12.][6.13.] library. Results using these tools were very good but required significantly more effort to install and manage the ship.

A DB based in MySQL proved to be slower and more complicated to manage than one based on Sqlite3; however, a server – based RDBMS has its advantages, especially concerning connectivity: while each module such as an MFD is considered as a different user by the RDBMS and requires a definition of password and access privileges, a server – based database can interconnect computers running modules that can be literally anywhere. Sqlite3 is a compromise solution that works very well, but has more limited connectivity.

And aside from all that, since the KB of this particular AIE is a relational database and the language used to write the rules is SQL, it is easy to extend the functionality of the system to provide more services for the ship such - as was mentioned before – data exchange between modules such as various MFD or even ships using the RDB included as an interface between them. As SQLC demonstrates, it is possible to connect to the KB from any module by just adding some C++ code.

5. Use

5.1. Installation

Refer to the README.md file contained in the original compressed file that corresponds to DGIIIAI.

5.2. Quick startup

There is no way to way to quickly grasp everything about this project, but two scenarios are provided as examples. Load the first one and you will find your ship on the runway at KSC. You should just let the ship perform its start up procedures as the initial programs are loaded into *sde_rules*.

It will close hatches, test control surfaces, set adequate trims, etc. This will take a while. Them the ship will start accelerating and it will take of, reach a safe altitude and start turning to acquire a certain heading. It is a simple example, but essentially shows what the ship does.

5.3. Realistic startup:

This will require you to become acquainted by the ship and the AIE first. You will need to set the ship to prepare an aiplan or manage modes on your own.

6. Sources

6.1. Edronkin, P. (2019). g2q - Guile to QASM compiler. [online] g2q. Available at: https://peschoenberg.github.io/g2q/ [Accessed 26 Aug. 2019].

6.2. Edronkin, P. (2019). gexsys - Guile Expert System. [online] gexsys. Available at: https://peschoenberg.github.io/gexsys/ [Accessed 26 Aug. 2019].

6.3. Edronkin, P. (2019). SQLC - Multi function display for the DGIIIAI Delta Glider. [online] SQLC. Available at: https://peschoenberg.github.io/SQLC/ [Accessed 26 Aug. 2019].

6.4. En.wikipedia.org. (2019). Expert system. [online] Available at: https://en.wikipedia.org/wiki/Expert_system [Accessed 31 Aug. 2019].

6.5. En.wikipedia.org. (2019). *Homoiconicity*. [online] Available at: https://en.wikipedia.org/wiki/Homoiconicity [Accessed 31 Aug. 2019].

6.6. En.wikipedia.org. (2019). Rule-based machine learning. [online] Available at: https://en.wikipedia.org/wiki/Rule-based_machine_learning [Accessed 31 Aug. 2019].

6.7. En.wikipedia.org. (2019). *Self-modifying code*. [online] Available at: https://en.wikipedia.org/wiki/Self-modifying_code [Accessed 1 Sep. 2019].

6.8. Friedman, R. et al (2019). Home - OpenMP. [online] OpenMP. Available at: https://www.openmp.org/ [Accessed 31 Aug. 2019].

6.9. Kelechava, B. (2018). The SQL Standard - ISO/IEC 9075:2016 - ANSI Blog. [online] The ANSI Blog. Available at: https://blog.ansi.org/2018/10/sql-standard-iso-iec-9075-2016-ansi-x3-135/ [Accessed 31 Aug. 2019].

6.10. Celestia Development Team (2001). Celestia: Home. [online] Celestia.space. Available at: https://celestia.space/ [Accessed 4 Sep. 2019].

6.11. Cosmos.esa.int. (1997). Catalogues - Hipparcos - Cosmos. [online] Available at: https://www.cosmos.esa.int/web/hipparcos/catalogues [Accessed 4 Sep. 2019].

6.12. King, D. (2019). *dlib C++ Library*. [online] Dlib.net. Available at: http://dlib.net/ [Accessed 1 Sep. 2019].

6.13. King, D. (2009). *Dlib-ml: A Machine Learning Toolkit*. [ebook] Journal of Machine Learning Research 10, pp.1755 - 1758, 2009. Available at: http://jmlr.csail.mit.edu/papers/volume10/king09a/king09a.pdf [Accessed 1 Sep. 2019].

6.14. Mysql.com. (2019). MySQL. [online] Available at: https://www.mysql.com/ [Accessed 26 Aug. 2019].

6.15. Postgresql.org. (1996). PostgreSQL: The world's most advanced open source database. [online] Available at: https://www.postgresql.org/ [Accessed 26 Aug. 2019].

6.16. Riley, G. (2019). CLIPS: A Tool for Building Expert Systems. [online] Clipsrules.net. Available at: http://www.clipsrules.net/ [Accessed 31 Aug. 2019].

6.17. Schweiger, M. (2000). Orbiter 2016 Space Flight Simulator. [online] Orbit.medphys.ucl.ac.uk. Available at: http://orbit.medphys.ucl.ac.uk/ [Accessed 26 Aug. 2019].

6.18. Sqlite.org. (2000). SQLite Home Page. [online] Available at: https://www.sqlite.org/index.html [Accessed 26 Aug. 2019].

6.19. Weiss, S. and Indurkhya, N. (1995). Rule-based Machine Learning Methods for Functional Prediction. [ebook] AI Access Foundation, Inc. Available at: https://jair.org/index.php/jair/article/view/10150/24055 [Accessed 31 Aug. 2019].

6.20. Widenius, M. (2019). MariaDB.org. [online] MariaDB.org. Available at: https://mariadb.org/ [Accessed 26 Aug. 2019].

7. Alphabetical Index