User's Manual

sEnglish Publisher

for

Eclipse 

London – Southampton, UK

September 2011

Contents

sEnglish Publisher is a MATLAB related natural language programming system under Eclipse that can be used

- to produce formal representations of procedural knowledge in engineering
- to write sEnglish sections in publications that can be executed by the reader
- to write sEnglish sections in publications that can be learned from by an agent
- to program the reasoning cycle and operational logic of intelligent agents
- to publish annotation free machine readable web pages.

and possibly more that we still need to discover.

Cognitve Agents Toolbox (CAT) is an extension of the sEnglish Publisher under Eclipse with a compiler to allow the creation of Jason code with data structures that are conceptually linked to human ways of modelling the world. A combination of a CAT, Jason, MCMAS, Latex plugins under Eclipse and the Agent Executive Toolbox for MATLAB/Simulink jointly make up a powerful programming environment for cognitive physical agents onboard robots.

Installation and Updates

Standard installation

Installation under MS Windows XP, Vista and 7:
Unzip **sEP_Installer.zip** into a new folder, double click on **setup.exe** and follow the installation window sequence.

Version maintenance

Updates of **sEP_Installer.zip** are announced to users by email and can be downloaded from www.senglish.org/promo or from system-english.com/promo .

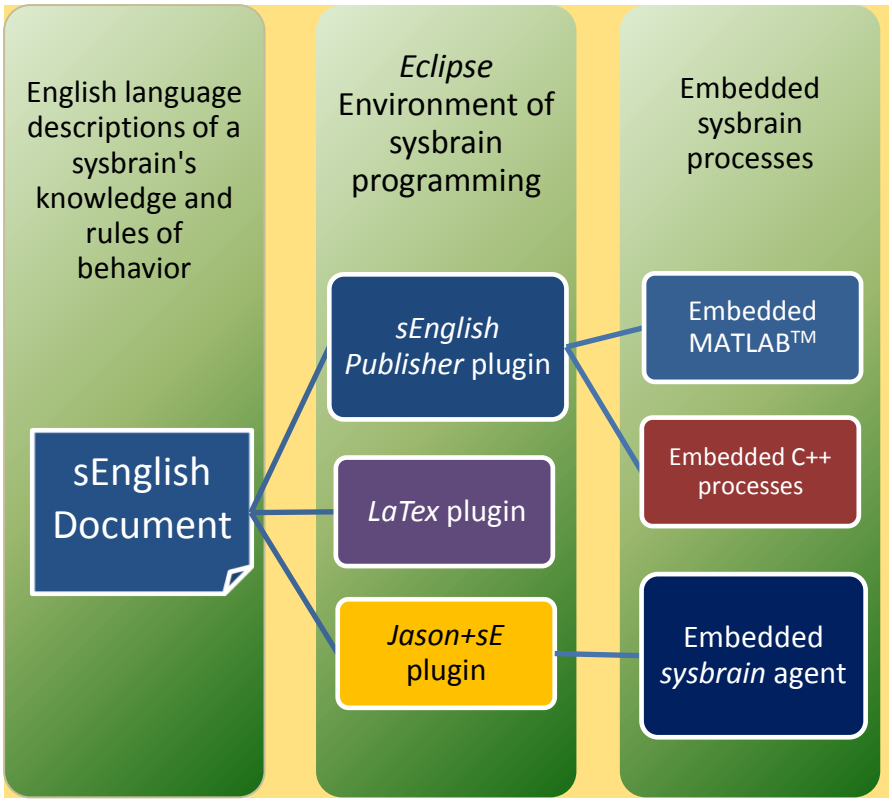
Usage of sEnglish Publisher is subject to licence agreement even if it is obtained for free.
Activation of various advanced functionalities are regulated by pass-codes.

Overview

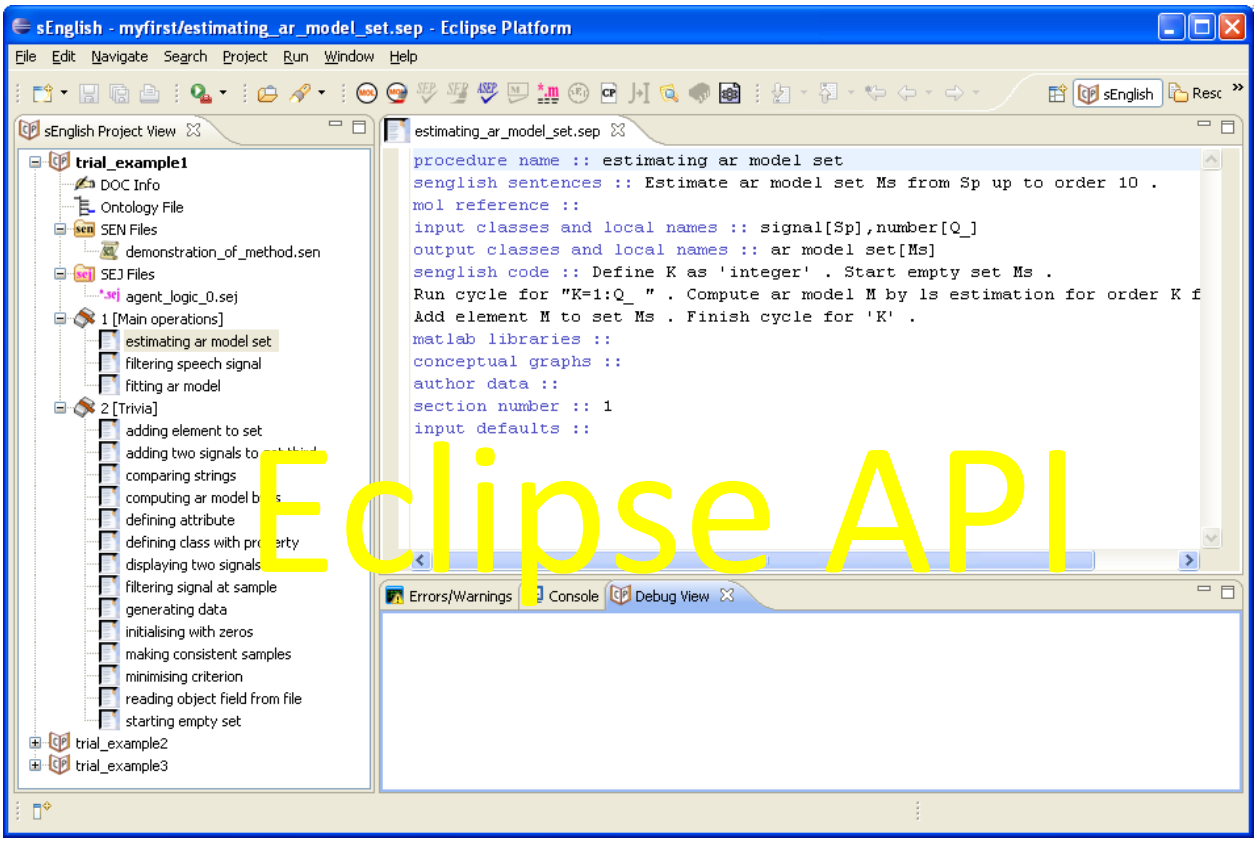
The Cognitive Agent Toolbox (CAT) is a software integration tool for automated or autonomous appliances. To provide intelligent behaviour in complex situations It enables its user to define and deploy a "sysbrain" rational agent architecture on an appliance's embedded processor. Sysbrains can be programmed to have unprecedented level of adaptivity and environmental awareness.

CAT is a complete system for creating, and maintaining machine intelligence throughout its lifetime while serving its human users.

CAT is based on the use of an English language document to encode the behaviour of intelligent agents controlling a machine. The documents is conceptually organised in such a manner that it supports easy understanding of the agents knowledge of the world, its reasoning and its behavioural constraints.



Technically a single *Eclipse application programming interface (API)* is used to do
(1) Create and author an sEnglish document describing and editing an agent's knowledge
(2) Publish this knowledge on the Internet or in print as HTML or LaTex/PDF based document
(3) Compile agent reasoning into Jason/Java processes into a .jar file
(4) Compile signal processing procedures of sensing, modelling and control into embedded processes in MATLAB/C/C++.
(5) Use the Agent Executive Toolbox to produce embedded applications for devices.

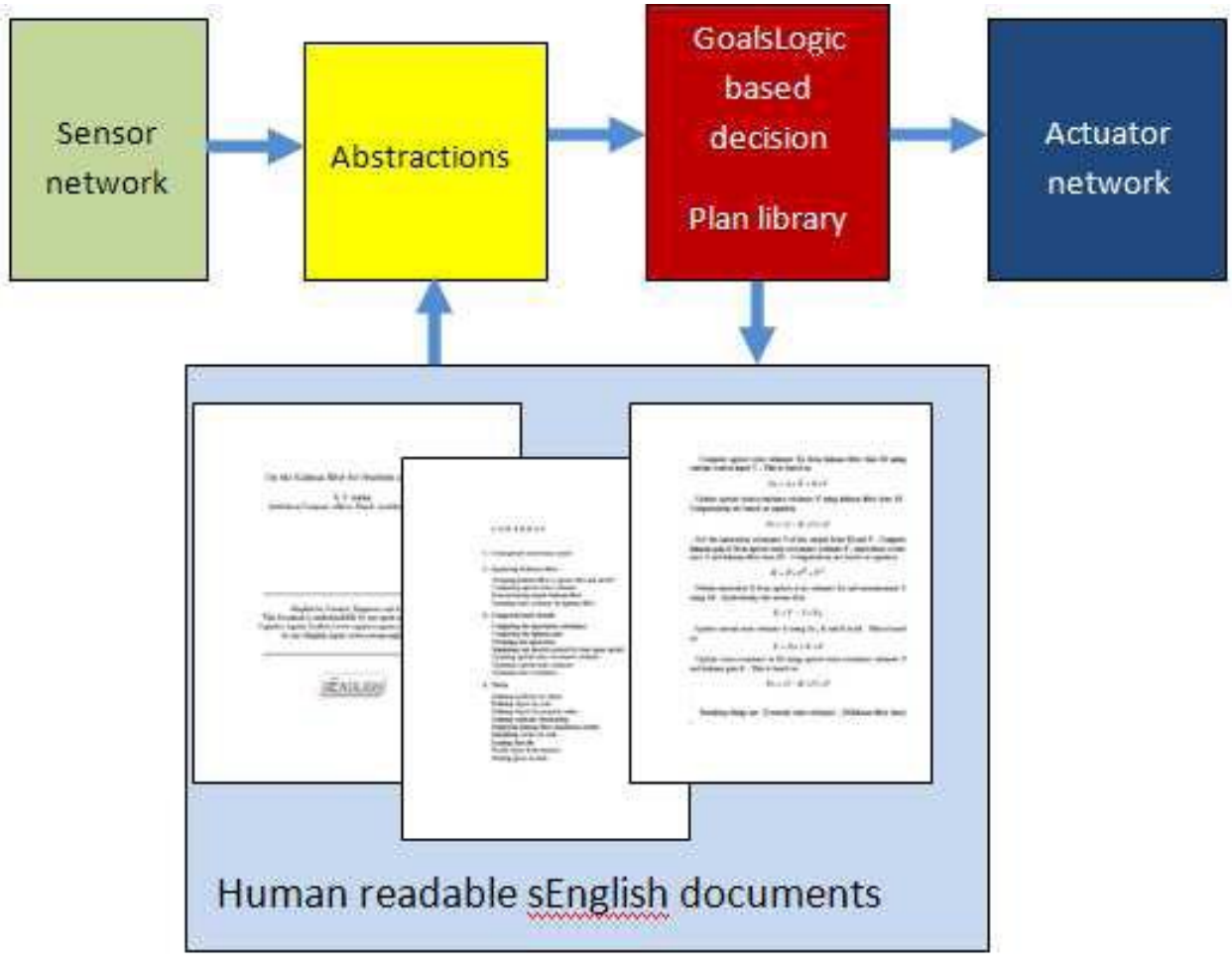


At the root of all agent knowledge lies its document that summarises its behaviour to both humans and the agent itself. The document is a "summary" that unambiguously compiles into the executable code of the agent,. And yet this document is indeed a summary as it is written at a very high abstraction level. High abstraction level objects and relationships in the world model shared by humans are explained by English sentences hierarchically until sentences are only explained by a piece of MATLAB code. MATLAB is universally capable to handle sensing, signal processing and control procedures and compiles into C/C++ for embedded code for small devices. It is only "Sensor networks" and "Actuator networks" that lie outside the abstraction framework. All of the goals, logic based reasoning, sensing are expressed and related in English sentences.

CAT benefits for system programmers

- You can formulate in English sentences the basic behaviour rules for safety and priorities, values of events and actions.
- Code development is easily shareable in a team as all code of your sysbrain is in natural English sentences of your choosing.
- You can program your sysbrain to enable it to solve complex problems across physical domains based on the principle "everything is possible that is not disallowed by the rules" .
- CAT enables you develop a sysbrain that makes logic based decisions on time as things happen in the world.

- CAT enables you to develop a sysbrain that, upon request, can also describe the reasoning behind its decisions, ongoing actions, reactions to events in English.
- upport and developer feedback is available via http://dev.sysbrain.com/bugs.



Benefits to users of robots

- The manual of your appliance is its program, some parts of it you may be able to redefine in English sentences to modify its behaviour.
- You can ask your appliance to explain what is going on, you can receive reasons for actions or lack of them.
- Intelligence updates are by the device reading publications on the web, that you can also read, not by replacing your device.
- Manufacturers can release new sensors/actuators to be added/removed for which software is reconfigured, better value for money.

Components of the Cognitive Agent Toolbox

The CAT Application Programming Interface (API) contains:

- sEnglish™ Publisher on Eclipse
- sEnglish to Jason+ Compiler
- Web/PDF Publisher
- Document Reader Tools
- Executive Toolbox for MATLAB/Simulink™
- Jason+sE to MCMAS compiler [1]
- Formal verification tool set[1]
- Java Virtual Reality Tools for complex environments[1]

Note that the Jason used in CAT is an upgrade of the official version at www.jason.org called *Jason+*. *Jason+* know all that the standard Jason plus more that can be summarised as follows:

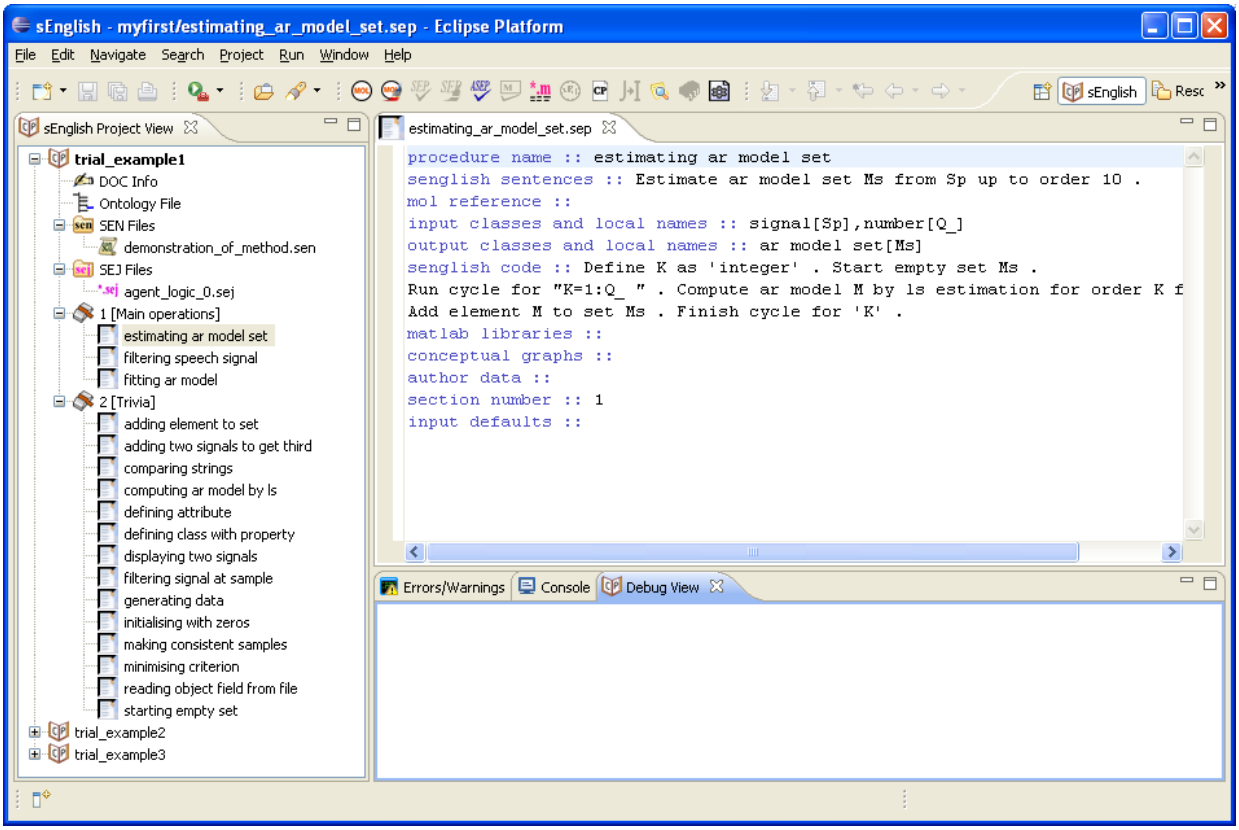
- Context formulae can also contain external calls using the *testBoolean* .
- Built-in external calls have been implemented to call m- functions symbolically. Predicates with variables in Jason correspond to MATLAB functions. When called in Jason some of the predicate arguments are first substituted from agent objects memory as inputs for the m-function. After the m-function is executed some of the symbolic predicate arguments receive values in the agents object memory. The MATLAB calls in Jason+ include *invoke(...)* for calling an m-function, *configureBoolean(...)* to set up cyclic updating of the value of a Boolean output m-function and its automatic transfer to the Jason agent's belief base, *configureObject*(...) for cyclic updating of object values that are not handed over to Jason+ but symbolically handled.
- The types *scalar* and *string* (which are subclasses of MATLAB types *double* and *char* , respectively) can be outputs of some MATLAB functions in which case the Jason+ variables corresponding to them can be updated with these values and use in Java based analysis that is allowed in Jason+. The sEnglish based agent reasoning program can contain Java based logic analysis as only sentences in square brackets [...] are compiled into invoke statements in the Jason+ executable plans..

[1] Under development

Getting Started

Main Editable Parts

A typical view of a session in sEnglish Publisher session is displayed below:



On the left hand side one can see the sEnglish Project View that can contain several projects with the icon (“CP” stands for conceptual programs/projects). The active project title is in bold-face (**trial_example1**) and its project tree is expanded. Each project includes the following items:

1. The Ontology File item that contains a single ontology ***.ont** file for conceptual hierarchies. Only one ontology file can be used per project.

2. The DOC Info item that contains the **DOCInfo.txt** file listing the project/document title, author data and section title. It is solely used to change these items for the whole project.

3. SEN Files folder to contain sEnglish scripts (**.sen** files) that can be executed. Beside icons .sen sEnglish script files are listed that can be edited upon a double-click.

4. SEJ Files folder that can contain one or several ***.sej** files. Beside icons .sej files are listed that can be edited to define an agent’s reasoning in terms of sEnglish/Jason statements.

5.  *Section number and titles in square brackets* [..] containing ***.sep** files for definitions of actions, relationships or questions. Beside icons  .sep files are listed that can be edited upon a double-click.

An *sEnglish project* is simply the collections of these files in a folder.

Editor controls

1. **Opening a new sEnglish project without an .exp file**. Use File >> New/Project >> sEnglish /sEnglish Project and click on [Next >] to get the sEnglish Project Wizard to create a new sEnglish project. Type in any suitable name into the Project name (this will not be displayed at the root of the project tree) and unclick the Use default location to define your own project folder on your hard drive by Browse. Type in or browse to the directory of the new project. Click on Finish. Wait until the new project appears in the tree of sEnglish Project View beside an icon  . Note that you should not use this method to open an existing sEnglish project from a folder as this will produce another .ont file that is not allowed. Use instead "Import" as described next.

2. **Opening a new sEnglish project from an .exp file**. Use File >> New/Project >> sEnglish /sEnglish Project from .exp file to be prompted for a file name and folder for the project to be created. Note that .exp files can be extracted from PDF or HTML browser images by copy pasting.

3. **Opening a new sEnglish project from an .pdf or .html file**. Use File >> New/Project >> sEnglish /sEnglish Project from .pdf/.html to be prompted for a file name and folder for the project to be created.[2]

4. **Import and existing sEnglish project.** This assumes that you have all the project files DocInfo.txt, a single *.ont file, some *.sep files, and possibly some *.sen and .sej files in a folder. Use File >> New/Project >> sEnglish>>Project to activate a wizard and define the folder name that contains the project files. You will be prompted for an internal project name for Eclipse and the tree of the new project will appear in the Project View window as an active project.

5. **Set active project.** If there are several projects in the tree then the active project is in bold face. To make an inactive project active: select it with left click and then right click on it to get a popup menu in which left click again to choose Set Active Project . The selected project should change to bold face and the project will be active. Active means that ontology, all sep-file compilations and all other global operations will refer to it.

6. **Refresh.** Right click on any item in the sEnglish Project View and select Refresh. Refresh is definitely needed if you have copied new files into the project folder, i.e. externally to the sEnglish Publisher.

7. **Open editable file.** There are two ways to do this: either right click on a file name and select Open by left click or *double click* on name of file.

8. **Delete a .sep file or a section.** Right click on name of file or section and select Delete by left click. Note that the deleted file will not appear in the Recycle Bin, hence extra care must be taken when deleting files.

[2] This feature is under preparation

9. **Create new .sep file.** Right click on section title and select New SEP-file.

10. **Remove a project from Project View..** Right click on project name and select Remove Project. This will not erase the content of the sEnglish project folder.

Project procedures

1. **Compile Ontology.** The button with icon can be clicked to compile the .ont file of the project. Note that a single ontology file is allowed for a project. No two *.ont files should be present in the project directory. Error messages or warnings will be displayed in the Problems tab.

2. **Sep Chek.** Click on the button with icon to check the syntax of a .sep file with respect to its sample sentence, available object and resulting object declarations. No checks are carried out about the sEnglish code. Errors are displayed in the Problems tab.

3. **All Sep Check.** Click on the button with icon to check the syntax of all .sep files with respect to their sample sentences, available object and resulting object declarations. No checks are carried out about the sEnglish codes. Errors are displayed in the Problems tab.

4. **Meaning check.** Click on checks the sEnglish code of the active .sep file for the interpretation of its sentences in terms of sentences defined within the project document.

5. **Compile all M-Files.** Having checked all .sep files for their syntax and meanings, the button can be clicked on the compile all .sep files into corresponding m-functions in MATLAB™. The produced m-functions are placed into the project directory.

6. **Compile Internet Papers.** To compile HTML and LaTex papers from the click on the button. .html and .tex files of a papers containing all project data will be produced and deposited into the project directory. These papers will have title page, contents, sections, subsections and appendix containing low level code.

7. **MATLAB™ Script Executor facility**. This button activates utility that is able to interpret MATLAB scripts via the use of the button that executes a selected **.sen** file in sEnglish . Without this utility being active the sEnglish script files in the SEN Files folder will not be executed and error message will be given. The MATLAB script translated from the .sen file can use any m-function that is present in the project directory or built in m-files in the *2007a release* of MATLAB by MathWorks Inc. exclusive any toolboxes.

8. **Saving executable sections**. A debugged and compiled sEnglish project can be saved into a **.txt** file in self-contained format that enables its recovery upon extraction from a publication in PDF format. Authors of publications can select this feature to *export their algorithmic results in an sEnglish project* and include it in publication between **sE-** and **–sE** delimiters. The exported section can be split into paragraphs and included into the published paper in an arbitrary order, each section placed between sE- and –sE. The sEnglish executable sections in any PDF publication can be extracted using a utility of the publisher on their website or copy-

pasted by the reader of the publication and assembled into a **.exp** ASCII file. The reader can then create a new sEnglish project from the .exp-file using the File >> New/Project >> sEnglish/sEnglish Project from Executable Paper (.exp).

9. **Compile to Jason**. Having compiled all m-files from .sep files successfully, compilation of an active **.sej** file can be carried out by clicking on the button. *This feature is only available for specific licensing types of sEnglish Publisher. Please contact support@system-english.com for information on availability of this feature.*

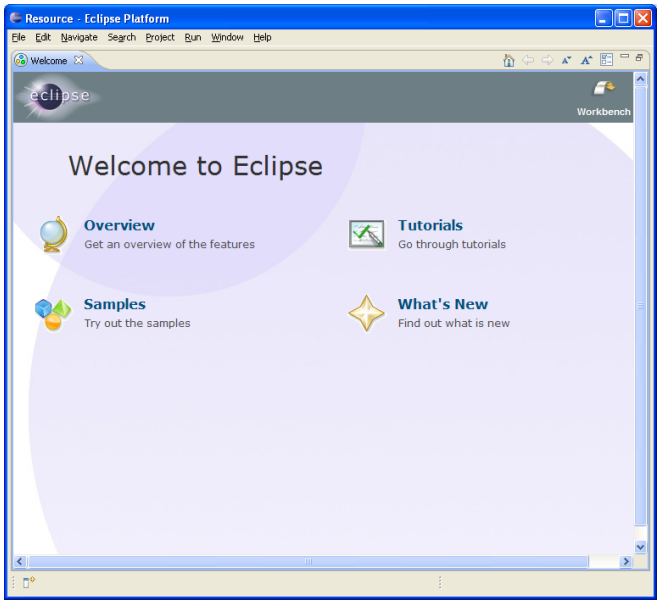
10. **Jason to ISPL.** The agent code of .sej files (and the equivalent .asl files) can be abstracted out to obtain discrete event system described using the interpreted system programming language (ISPL) using the button. For this to be feasible special "//x" comments must be placed after each plan in the sEj code of the .sej file. See the details of this in the Declaring Agent Logic section of this help facility. *This feature is only available specific licensing types of sEnglish Publisher. Please contact support@system-english.com for information on availability of this feature.*

11. **Quick Reference** is provided for fast lookup up of sEnglish syntax, semantics, SEP Editor and CAT features .

An Example

A small project has been created to document the algorithms of filtering a speech signal using AR model estimation and its application to a noisy speech signal to demonstrate an adaptive method of speech signal enhancement.

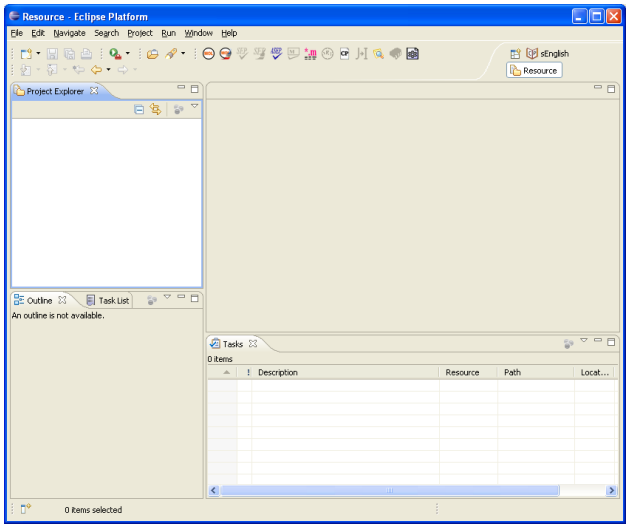
When installing and first using sEnglish Publisher, the standard Eclipse Welcome window

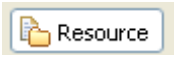


This contains numerous interesting information about the Eclipse such as an Workbench basics under Overview that explains how the *Eclipse Features, Resources, Perspectives, Views and Editors*

operate. These are very useful to know about while using the sEnglish Publisher. Eclipse is the most flashy and powerful editor system available today for programming tasks such as natural language programming.

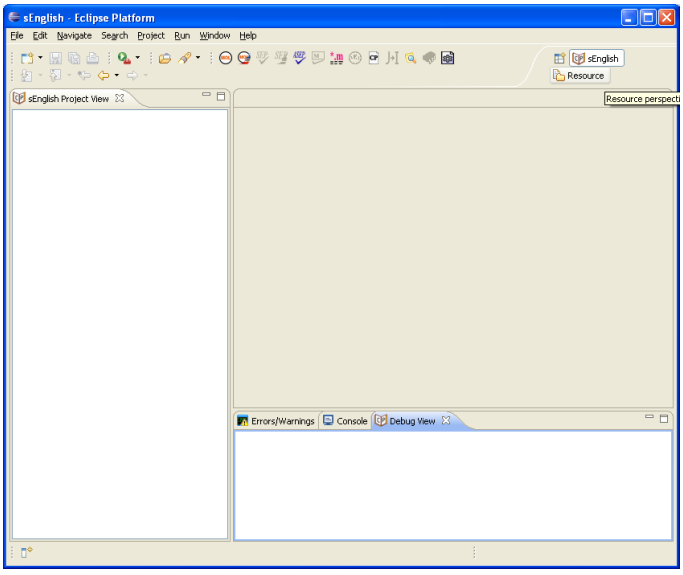
For the purposes of this example, close the Welcome window by clicking on the **x** in its tab. The next window revealed can look similar to



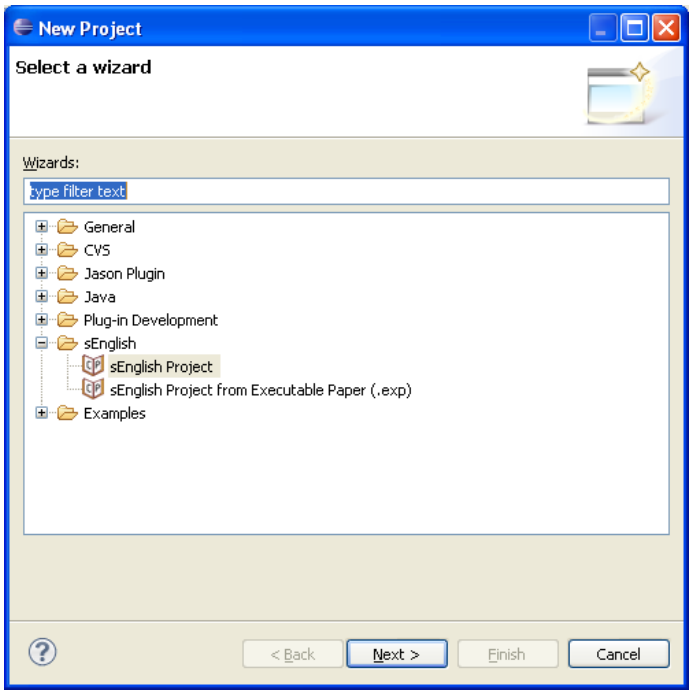
that shows Eclipse in the Resource 

perspective that is not what we want to use under the sEnglish Publisher. Click on the 

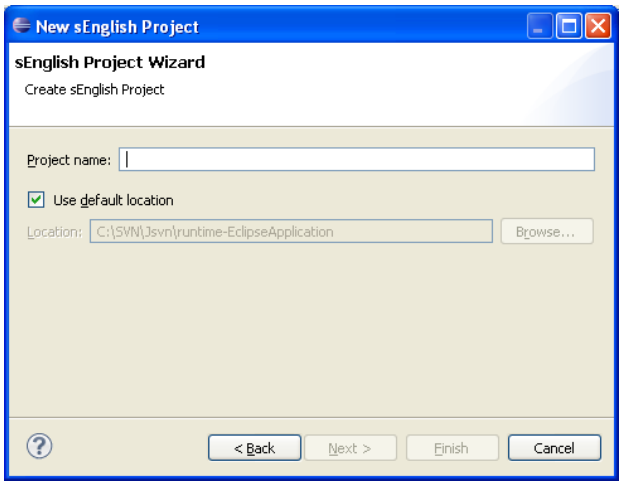
button (top right part of the window pane) to switch to the *sEnglish Perspective* for the sEnglish Publisher facilities to become available. If it's not in the list you can find it under Window->Open Perspective->Other.. and choose sEnglish Perspective in the list. Having switched to the sEnglish perspective a window similar to the one below should appear:



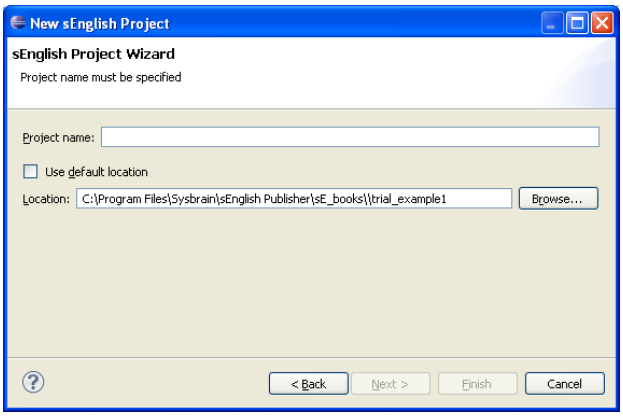
This window is now switched to the sEnglish perspective but does not contain any sEnglish project yet. Use File/Project to invoke the



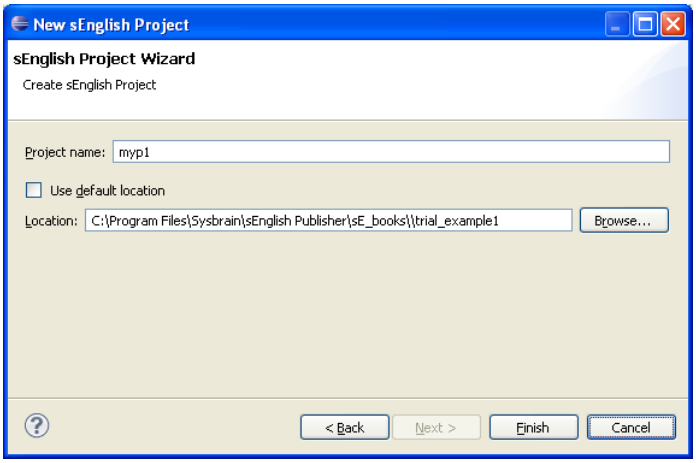
wizard selection window that contains several wizard folders. The one that interests us is the sEnglish wizard folder that contains two items: sEnglish Project and sEnglish Project from Executable Paper(.exp) . Select sEnglish Project so that the New sEnglish Project Wizard window



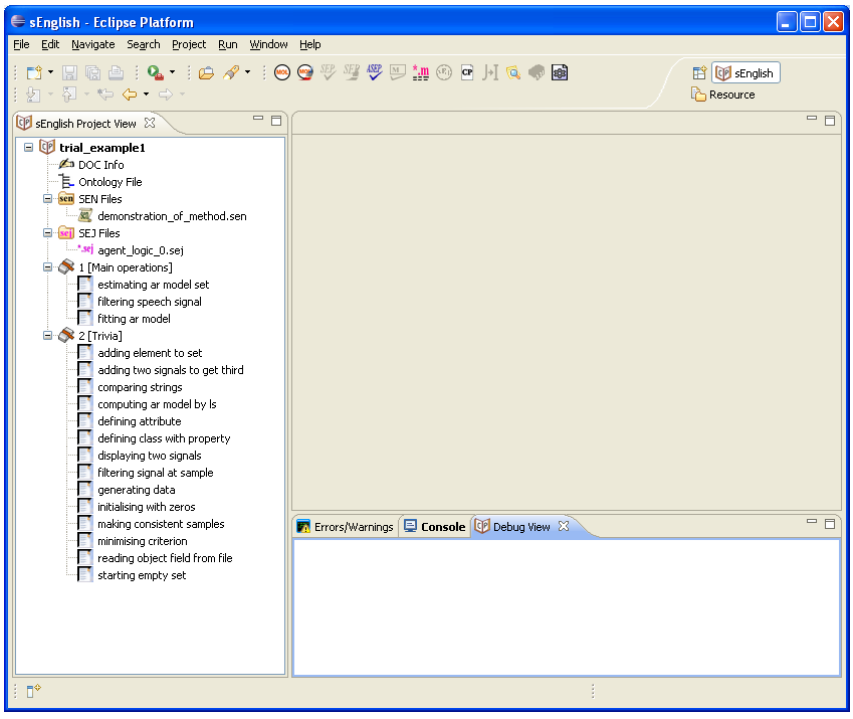
appears. This requires typing in a Project name that will be recorded in the `.project` meta file of the project directory but will not be displayed at the root of the project tree (the folder name will be displayed instead without its path). To define the right folder for our example, un-tick the “Use default location” and browse to a copy of the ...\trial_example1 folder.



When Finish is available (this may not happen in case the folder is being used by another Eclipse project already!) then click on it :

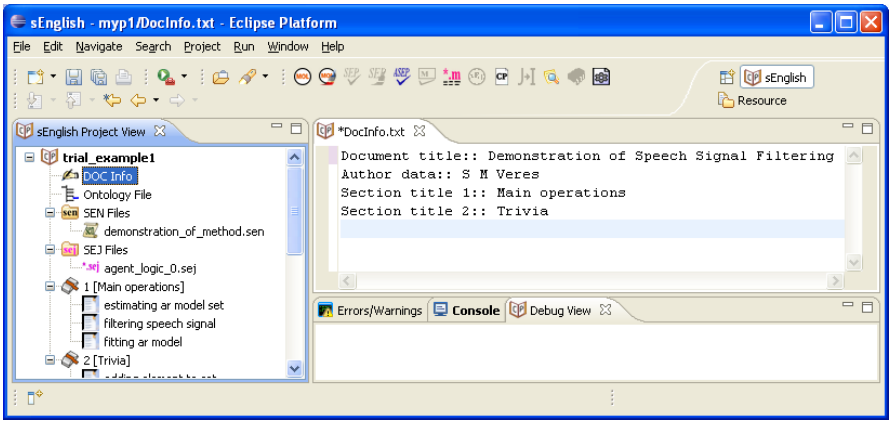


As a result the following window or similar should appear:

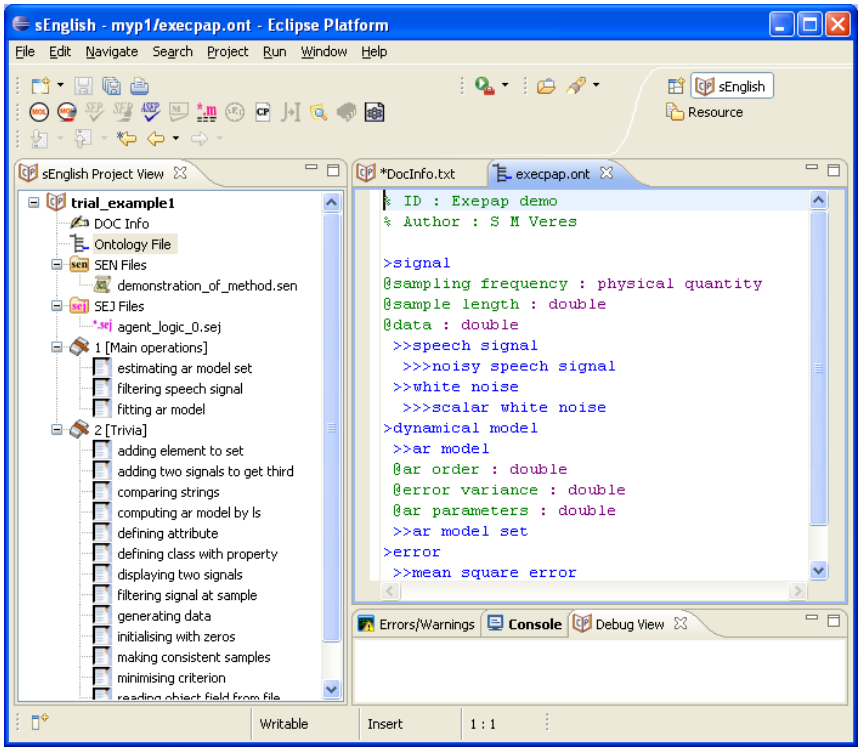


that now shows all the files found in the project directory that we will now explore. Before doing so it is worthwhile noting that by right clicking on a project one also has access to Properties (apart from Set Active Project, Remove Project and Refresh) that can be used at present to look up the path of the project folder.

By double clicking on DOC Info an editor appears for the `DOCInfo.txt` file of the project directory `trial_example1` :



that can be edited for project/document title, author and section titles of a machine readable document publishable from this project. After changes made to the file the button can be used to save the file as with all files worked on in the Eclipse system. Note also that to save all files that have been worked on the button can be used. Double click now on Ontology File to open an editor window for the single (only one is allowed!) ontology file of the project as:



This simple ontology looking extracted as

```
>signal
```

```
@sampling frequency : physical quantity
@sample length : double
@data : double
  >>speech signal
     @@data: size(..,2)<=3
    >>>noisy speech signal
  >>white noise
    >>>scalar white noise
>dynamical model
  >>ar model
   @ar order : double
   @error variance : double
   @ar parameters : double
  >>ar model set

>error : double
  >>mean square error
```

illustrates most of features of a MOL (machine ontology language) ontology. The blue items you can think of as classes of data objects . The green ones as attributes of classes and the purples as attribute classes or constraints .

The reason this is a MOL because all classes are derivatives if one of the four basic MATLAB types:

`struct, char, cell, double, logical, int8,...` etc.

You can think of an ontology as descriptions of the structure of objects used in a program. Each blue class defines a format for class of objects that can be used in sentences during natural language programming. The colon ":" is always followed by a class declaration or a constraint formula in MATLAB that needs to evaluate to Boolean `true`.

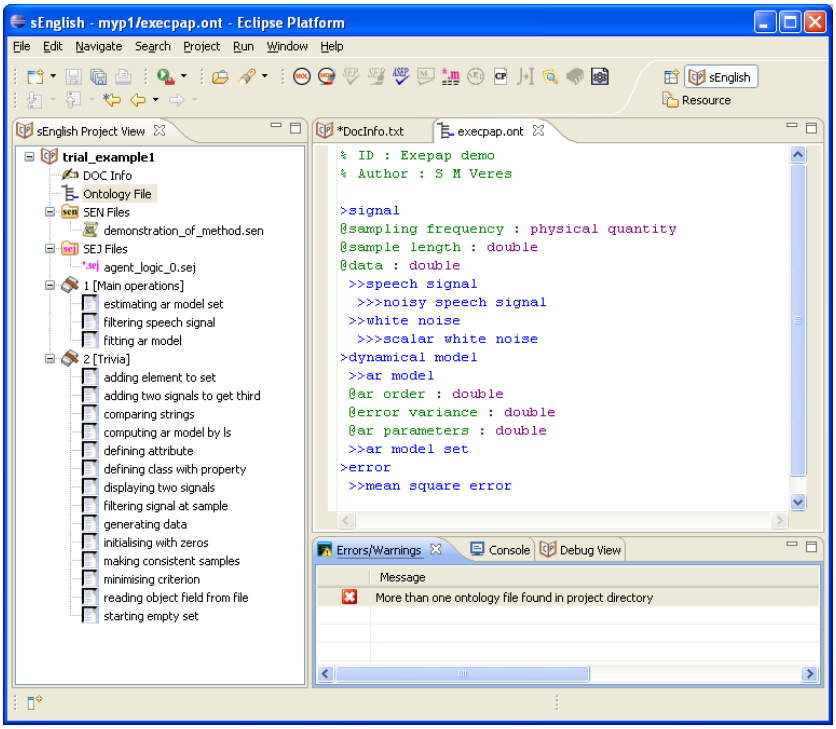
The classes with green attributes will have MATLAB type `struct` when using in a program. Classes or attributes declared with "`: char`" are of MATLAB type `char` . Similarly , declarations with "`: cell`" or "`: double`" mean respective MATLAB classes .

Note the following simple rules of ontology notations:

- Attribute classes can also be declared by classes defined in the ontology elsewhere, for instance `@sampling frequency : physical quantity .`
- Some classes are not structures and can be declared by using a colon after the class name, for instance `>error : double.`
- Subclasses are listed after class with larger number of >, for instance `noisy speech signal` is a *subclass* of `speech signal` and `scalar white noise` is subclass of `scalar white`. Subclasses inherit attributes from superior classes. For example and `speech signal` will have all the attributes of a `signal` .
- To make constraint declarations more concise, the ".." is used to indicate that the constraint or formula is to be applied to the whole object. These are the places where the object name would be substituted to evaluate whether a particular object satisfies a constraint. For instance `@@data: size(..,2)<=3` means that the data attribute of class `speech signal` must be maximum 3 columns .
- The options for some attributes can be listed in braces, for instance `@type: {'archive', 'current'}` means that the attribute type can be only one of two strings: `'archive'`, or `'current'`.

- The double dash "--" after a class can be followed by a MATLAB expression that can create a sample of that class . For instance `>list : cell -- {'mik';'mak'}` declares a MATLAB code to generate a sample of list . This is especially useful for structure d objects where it is difficult to write an algorithms for a good quality sample generator.

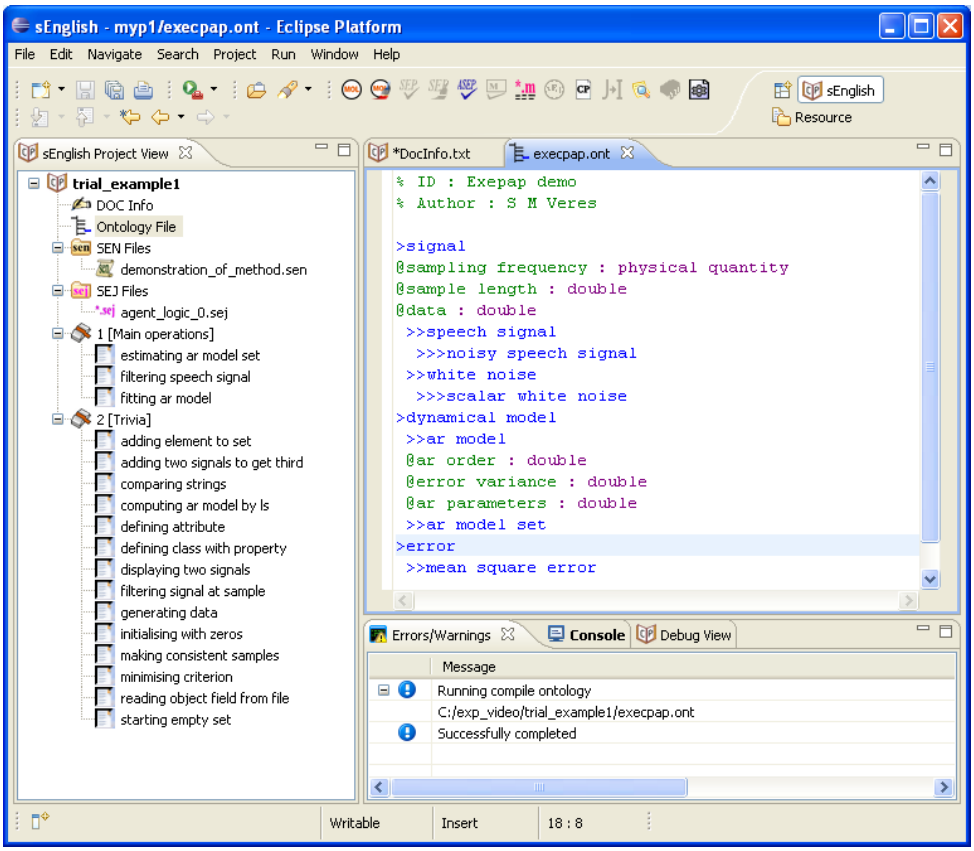
The ontology file can be compiled by clicking on button MOL . Before doing that let's click on Errors/Warnings that is the first tab in the bottom right area of the Eclipse window:



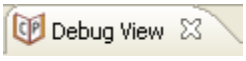
An error message appears in the display window as "More than one ontology file found in project directory". This happend because while preparing this manual a new peoject was opened in the existing demo directory that automatically created a `.ont` file with the name of the. To remedy this situation and to inform the Eclipse system about the correct .ont file, one must erase from the project folder all `.ont` files except `exepap.ont`.

General note: user of this system are allowed to add, erase, alter files in the project directory to repair the project (for instance erasing redundant or empty .sep files) with the rule that after the changes one must Refresh the project using the right click popup of the project name.

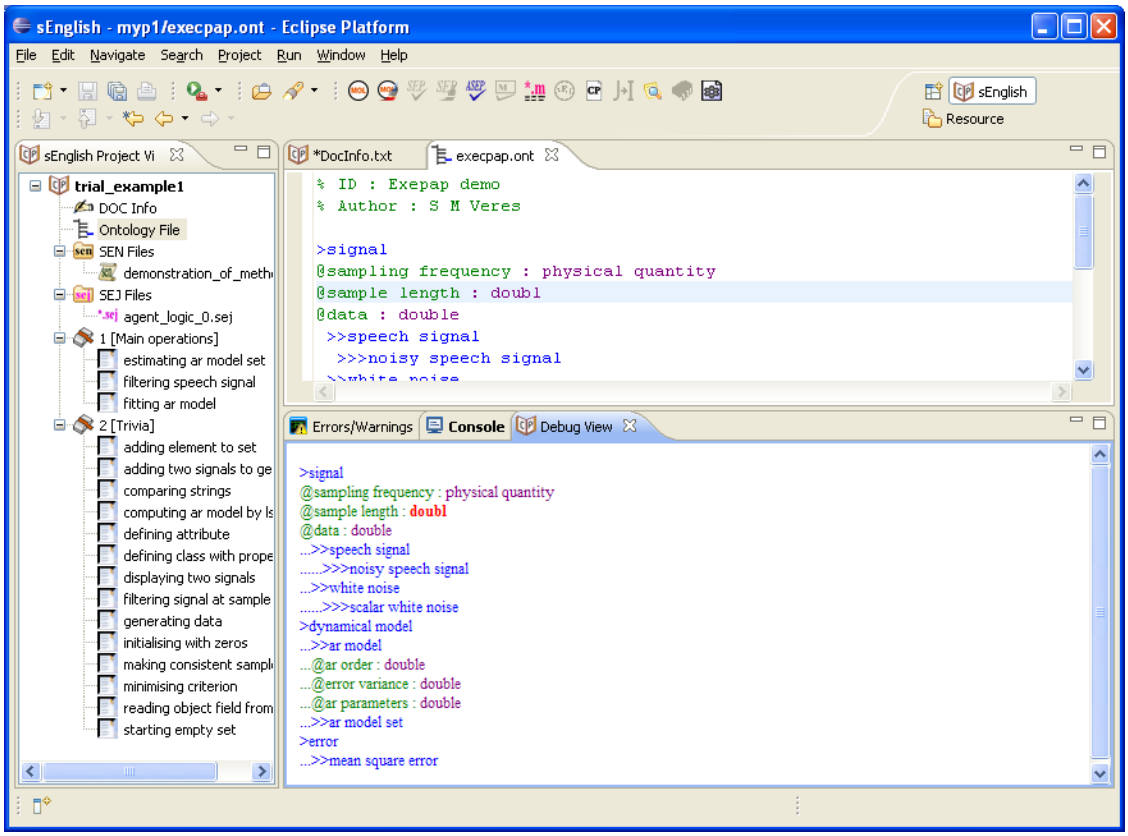
Having eliminated the reduncant .ont files, the ontology file can now be compiled by clicking on button MOL to obtain:



that now clealy states that ontology compilation was run and succedfully completed with name of the .ont file used shown with full path. By introducing an intentional error in the ontology by declaring the sample length of signal to be “doubl” (instead of “double”), on gets an error message as shown on the next page. For this to work one needs first to save the ontology file by  as there is no automatic save before compilation.

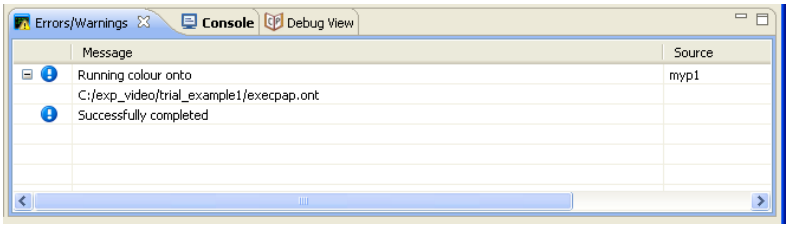
There is however an other way to show up mistakes in an ontology: by using the button  and inspecting the 

tab window as shown consecutively on thenext page.



that now clearly highlights the source of error in red.

By re-correcting and compiling the ontology now successfully the Errors/Warnings tab will again

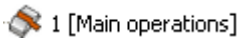


contain no errors. The error content of Debug View will not however disappear before another  or  is again activated but one can right click with cursor over the Debug View window and select Refresh to empty the window.

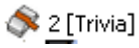
The Console window is only for the advanced user and is can normally be ignored, ultimately it can be used to detect the very infrequent breakdown of the sEnglish processor that is running in the background of the Eclipse system. This is however automatically restarted after the next use of  . Any persistent breakdown should be reported to support@system-english.com .

For the time being we step over any discussion of the use of the SEN Files and SEJ files as their use logically follows the writing and debugging of all `.sep` file definitions in the project/document sections under icons  and  .

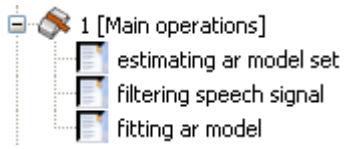
This example document contains two sections with titles

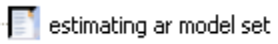


and



as displayed on the project tree in the sEnglish Project View . The first contains three activity definitions:



Each of these names the respective activity, provides sample sentences (can be several) for invocation in NLP and other data such as assumed known objects before use of the sentence and what it results in and not the least also its meaning is defined in terms of other sentences. All the data relating to `estimating ar model` set is displayed in an editor window when double clicking on the `.sep` file leaf/label 

:

```
procedure name :: estimating ar model set
senglish sentences :: Estimate ar model set Ms from Sp up to order 10 .
process, repeat mode ::
input classes and local names :: signal[Sp],number[Q_]
output classes and local names :: ar model set[Ms]
senglish code :: Define K as 'integer' . Start empty set Ms .
Run cycle for "K=1:Q_ " . Compute ar model M by ls estimation for order K
from Sp . Add element M to set Ms . Finish cycle for 'K' .
matlab routines url ::
conceptual graph ::
testing formula ::
section number :: 1
input defaults ::
```

The following table summarises the labels in SEP files:

Label	**Comment**
`procedure name`	All lower case tag to name the activity or relationship.
`senglish sentences`	Sample (template) sentences of how to invoke the procedure in sEnglish code.
`process, repeat mode`	The name of the executive process that is capable of evaluating/ executing the meaning of the sentence followed by comma separated 'runOnce', 'runRepeated', 'stopRepeated'. If there is no comma separated second entry then the default of 'runOnce' is used for the execution of the sentence.
`input classes and local names`	Comma separated lists of available objects *class_name*[*Object_Name*] before the sentence is . For some sentences this field can be empty, for instance when inputs are obtained from sensors.

`output classes and local names`	Comma separated lists of resulting objects *class_name*[*Object_Name*] before the sentence is . For some sentences this field can be empty, for instance when the outputs are settings of actuators.
`senglish code`	List of sEnglish sentences to defne the "meaning" of the sample sentences. The sentence Execute code " <*matlab code*> ". or Execute code " <*matlab code*> ". can be used to include pure MATLAB code at any point in the sEnglish code. Otherwise only sentences can be used that fit to sentence formats declared somewhere within the same sEnglish document.
`matlab routines url`	URL in the Internet where MATLAB routines or DLL libraries used in the meaning definition of sentences can be found.
`conceptual graph`	Detailed single conceptual graph description of the main sample sentences. (Must be the same graph for all sample sentences.
`testing formula`	A MATLAB formula that tests whether the resulting objects are what should have been obtained from available objects.
`section number`	A positive integer that should be maximum the number of sections in the document. It is used to indicate which section the sentence should belong to.
`input defaults`	Comma separated list of matlab expressions (possibly calling user defined m-functions) that can be used to define available objects for sentence testing.

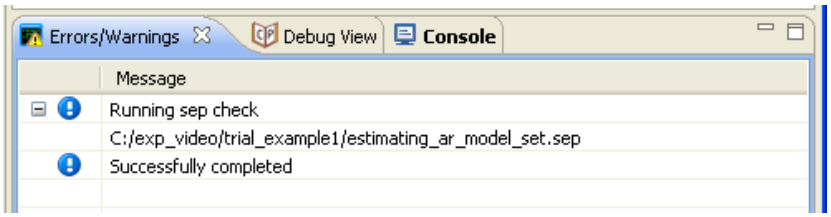
Informally a SEP definition is often called a "sentence" that is a bit of misuse of terminology as there can be several sentence formats declared to have exactly the same meaning: a SEP declaration allows several template sentences after the label senglish sentences .

Each of the sections of the document can contain a set of `*.sep` files (sEp stands for sEnglish procedure). SEP declarations can be three kinds:

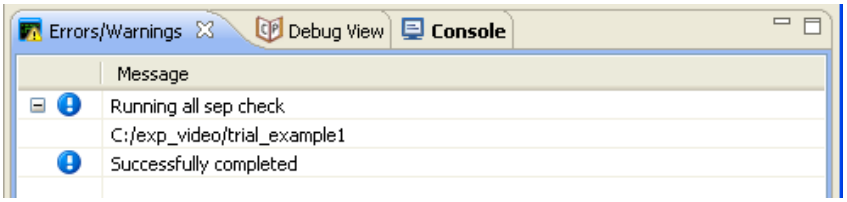
1. Definition of an *action procedure* that is invoked by a sentence. Such a procedure have available and resulting objects named in the sentence and each belonging to a class in the ontology.
2. *Statement of a relationship* using a sentence. Such procedures have available objects and their result is a Boolean value of `true` or `false` in a variable of type `relation_Boolean` that is to be declared in the list of resulting objects..
3. *Formulating a question using* a sentence ending with "?" instead of a period "." Questions have available objects named in the sentence and the "output" is an object with a class in the ontology that is declared as `answer_variable` . Note that questions are rarely used in agent programming but are possible in the sEnglish framework.

Debugging of .sep files is only possible after compilation of the ontology is successfully completed. For the Sep Check and All Sep Check buttons to be available one has to double clock on any of the .sep files items at the icons  .

Initially the `.sep` file definitions can be checked one-by-one using the Sep Check  button, for instance

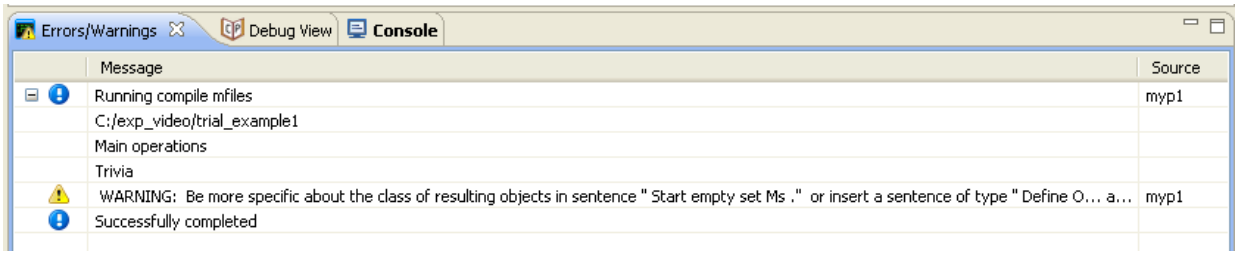


All `.sep` files in the project folder can be tested by the All Sep Check button in one procedure but the list of error messages can be potentially longer to inspect, a successful testing looks as



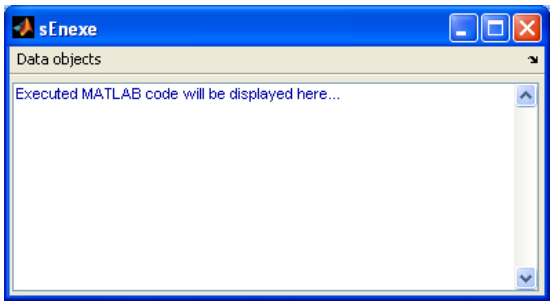
For use to execute sentences and sequences of sentences (called sEnglish scripts that can be stored in .sen files), all .sep files need to be compiled into corresponding *m-functions* in MATLAB™ in the project directory. Each subsection's meaning, for instance estimating ar model set, is to be compiled into a corresponding m-function, for instance `estimating_ar_model_set.m`.

To compile all .sep files in all sections into m-functions to represent the meaning of the associated sentences that invoke the activity or relationship, the Compile M-Files button can be used. When completed successfully, a message similar to

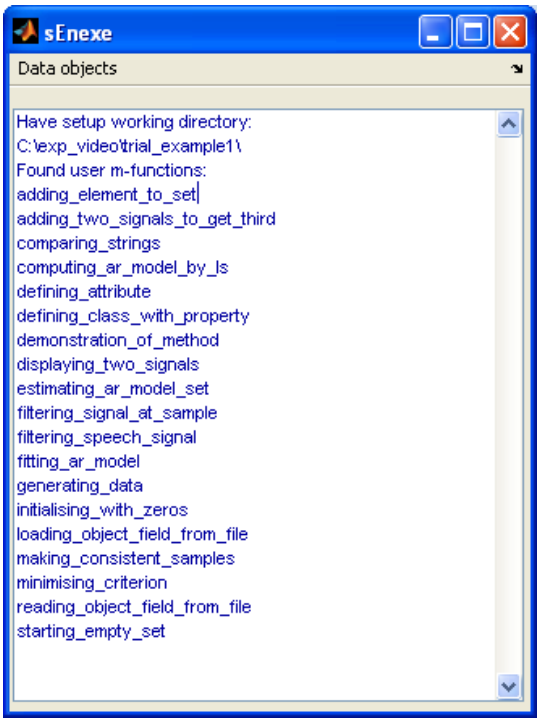


appears that on this instance provides a warning for the sEnglish programmer to check.

The button can be used to turn on a MATLAB™ based runtime facility that can execute m-scripts translated from .sen files. This facility needs to register the available user written m-functions (all m-functions that can be found in the project directory) in prior to executing sEnglish code in a .sen file. If there is the intention to run the results of a .sen file then should be run before is used to compile all the m-files. The opening window of the MATLAB runtime program is

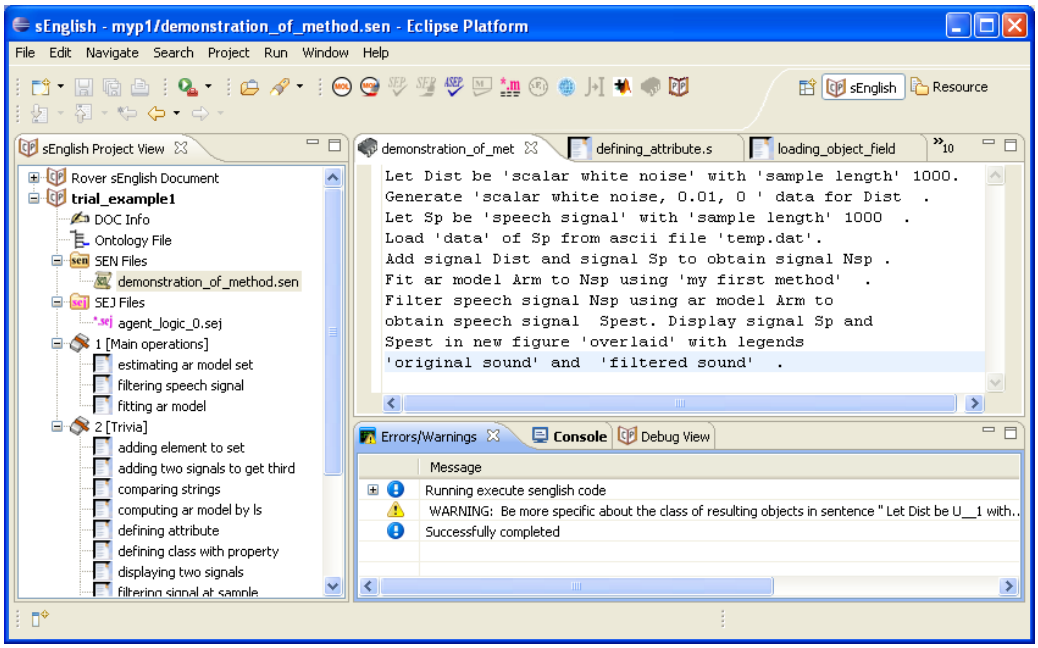


After compiling all .sep files into .m files (m-functions) by this runtime console displays

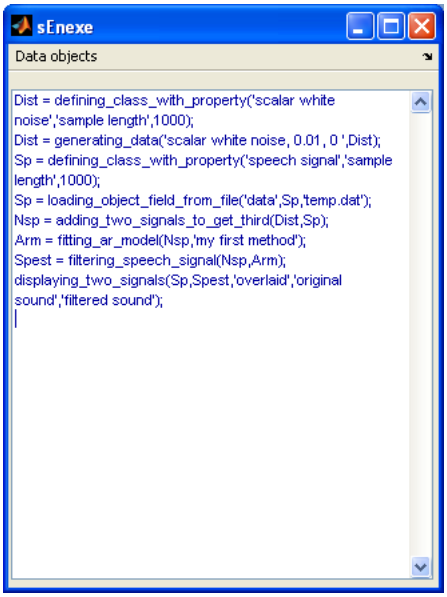


for our example project. Observe that this window now lists accepted m-functions for use with names that are underscore connected versions of the activity names at icons in the project tree.

Having successfully compiled the whole document, executable .sen files can be written from the sentences defined. New .sen files at icons can be opened in the SEN Files folder and sentences copy-pasted into to describe a procedure. Our example already has one such .sen file `demonstration_of_method.sen` the editor display of which can be activated by a double click on the file name as displayed on the next page. Note that the Editor tab of this file now has the icon for the simple reason that this sEnglish script can be executed using the button.

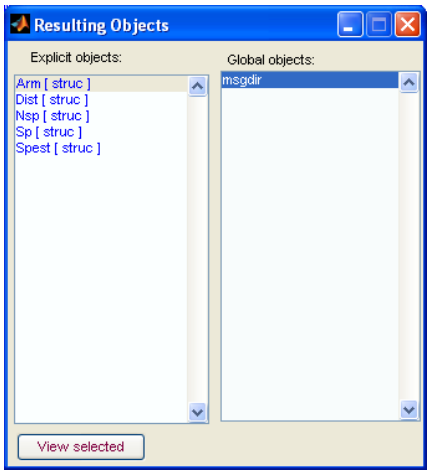


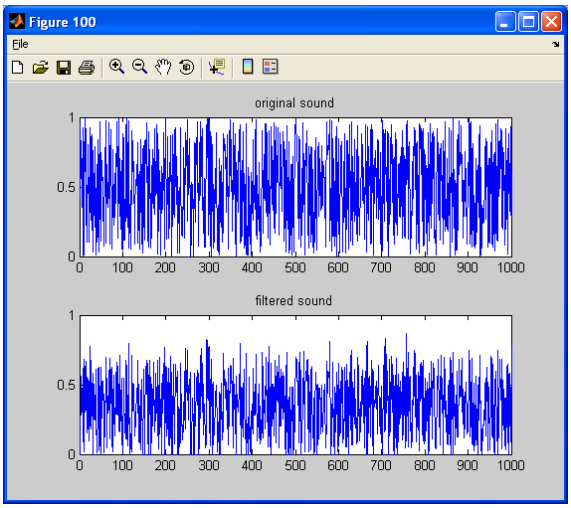
The sentences of this *executable file* (hence the use of the chip icon) can normally be copy-pasted from .sep definitions in the other editor windows with icons . After saving this file with (important as there is no automatic save before any of the button operations in this system) it can now be executed using the button that first compiles it into a sequence of m-function calls. Clicking on this button will start the procedure of compilation from sEnglish to matlab code. The code is sent over from the Eclipse based system to the utility that displays it as follows for the example being presented:



This code is the compilation of `demonstration_of_method.sen` into MATLAB.

Simultaneously this MATLAB code is interpreted and executed. During this run some objects may be created that are now displayed in Resulting Objects window as shown on the next page. The run may display some figures as well, for instance in this case MATLAB figure labelled Figure 100 on the next page.

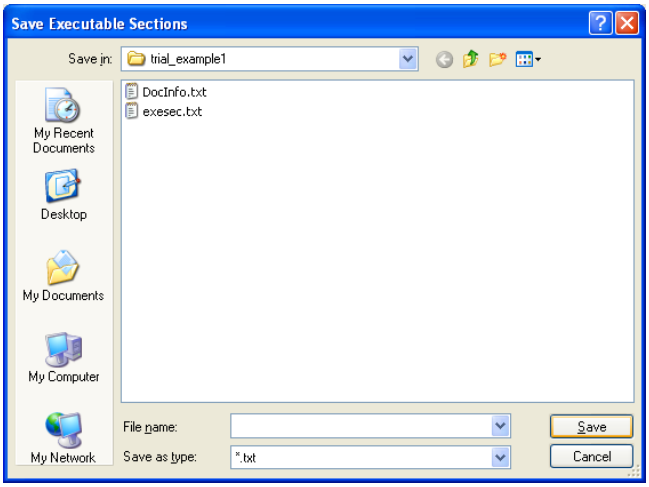




The example demonstrates that a technical procedure can be *formally represented* in terms of basic concepts, data and algorithms described in terms of English sentences. The emphasis here is on that this is not only a user friendly presentation of technical results but precise and unambiguous that can be analysed by computers. The daunting task of needing to browse through hundreds of publications can be analysed by digital assistants in the near future. The next section will describe how journal publications can be populated by sections exported from an sEnglish projects so that readers can extract them in their own projects and run the demonstrations. Though this process could be done by the author making his ordinary (i.e. non sEnglish) code downloadable on the Internet, but the advantage of using sEnglish is that it reveals to the reader that what is being demonstrated in the code is indeed what the theory presents. The next section describes on our example how a project can be exported into text for a paper to be published; this is followed by another section that shows how to extract an sEnglish project for experimentation from PDF publications containing sEnglish executable sections.

Exporting to a publication

The Create Executable Section button  can be used to compile a well formatted text for inclusion in a conference or journal publication or report that contains all the essential information of the project and is reversible in the sense that the whole project can be recovered and used by the reader of the sections.



The file saved for our example project is `exesec.txt`. By convention executable paper sections are exported into ACII files with .txt extension (this distinguishes them from the .exp files that are executable text extracted from published PDF papers and convertible to a project).

There is a general rule what constitutes the essence of an sEnglish project for publication in terms of what will be used for creating executable paper sections for publication :

1. all `.sen` files,
2. all `.sep` file definition in Section 1,
3. the single `.ont` file .

These and only these will be used to create executable text by the use of Create Executable Section button  . For our example the `exesec.txt` file exported by  is as follows:

```
Data classes: Main classes are dynamical model, error, signal. Attributes of signal
are sampling frequency (physical quantity), sample length (double), data (double).
Subclasses of dynamical
model are ar model, ar model set. Subclasses of error are mean square error.
Subclasses of signal are speech signal, noisy speech signal, scalar white noise,
white noise. Subclasses of speech signal are noisy speech signal. Subclasses of white
noise are scalar white noise. Attributes of ar model are ar order (double),  error
variance (double), ar parameters (double).

Executable script : Demonstration of method : Let Dist  be 'scalar white noise' with
'sample length'  1000 .  Generate 'scalar white noise, 0.01, 0 '  data  for  Dist.
Let  Sp be 'speech signal'  with 'sample length'  1000 . Load  'data' of Sp from
ascii  file  'temp.dat'. Add  signal  Dist  and signal Sp to obtain signal Nsp . Fit
ar model Arm to Nsp  using  'my first method'  . Filter  speech signal  Nsp  using ar
model Arm to obtain speech signal Spest. Display signal Sp and Spest in new figure
'overlaid' with legends 'original sound' and 'filtered sound'.
```

```
Estimating ar model set : Estimate ar model set Ms(r) from Sp(a) up to order Q_(10) .
Define K as 'integer' . Start empty set Ms .
Run cycle for "K=1:Q_ " . Compute ar model M by ls estimation for
order K from Sp . Add element M to set Ms . Finish cycle for 'K'.

Filtering speech signal : Filter speech signal Nsp(a) using ar
model Arm(a) to obtain speech signal Spest(r) . Let K be the 'ar order' of Arm . Let
L be the 'sample length' of Nsp . Initialise Spest as 'speech signal' with K initial
zeros for its 'data'. Run cycle for "S=K+1:L" . Filter Spest from Nsp at sample S
using Arm . Finish cycle for 'S' . Make sample length of Spest  consistent .

Fitting ar model : Fit ar model Arm(r) to signal Sp(a) using M_('my method') . If M_
is 'my first method' , then do the following . Estimate ar model set Ms from Sp up to
order 10.   Minimise "'(M.ar_order)*M.error_variance'" over all entries of Ms to
obtain best ar model Arm . Finish conditional actions.
```

The can now be placed into a paper to be published in an arbitrary order as the executable paper sections extractor will recognise them. In addition to this the author needs to place all remainder .sep files and .m files that are called from meanings into a single .zip file and place it into a *resources URL*. The resources URL is provided by the following sections from a PDF paper that contains the sections above for our example:

liability verification, will consider the following aspects. The following section is executable:

sE-
Executable Script: Demonstration of method. Let Dist be 'scalar white noise' with 'sample length' 1000 . Generate 'scalar white noise, 0.01, 0 ' data for Dist. Let Sp be 'speech signal' with 'sample length' 1000 . Get data from url 'http://www.speechsamples.com/sample34521.zip'. Load 'data' of Sp from ascii file 'temp.dat'. Add signal Dist and signal Sp to obtain signal Nsp. Fit ar model Arm to Nsp using 'my first method'. Filter signal Nsp using ar model Arm to obtain speech signal Spest. Display signal Sp and Spest in new figure 'overlaid' with legends 'original sound' and 'filtered sound'.
-sE

This paper sections declares the demonstration of the method developed. At another part, of the paper, possibly after a series of theoretical developments, the

The sentences in the demo can only be understood if some of the sentences are explained as in these sEnglish sections.

Some of the sentences in the above sEnglish code are defined as follows:

sE-
Fitting ar model : Fit ar model Arm(r) to speech signal Sp(a) using 'my method'. If M_a is 'my method' , then do the following. Estimate ar model set Ms from Sp up to order 10 . Minimise "(M.ar_order)*M.error_variance" over all entries of Ms to obtain best ar model M(r). Finish conditional actions.

Filtering speech signal : Filter signal Nsp(a) using ar model Arm(a) to obtain speech signal Spest(r). Let K be the 'ar order' of Arm. Let L be the 'sample length' of Nsp. Initialise Spest as 'speech signal' with K initial zeros for its 'data'. Run cycle for "S=K+1:L". Filter Spest from Nsp at sample S using Arm. Finish cycle for 'S'. Make sample length of Spest consistent.

Estimating ar model set: Estimate ar model set Ms(r) from Sp(a) up to orders 10 . Define K as 'integer'. Start empty set Ms. Run cycle for "K=1:Q_". Compute ar model M by ls estimation for order K from Sp. Add element M to set Ms. Finish cycle for 'K'.

References: Trivial m-files can be found at 'http://localhost:8080/ExecutablePaper/Resources/example1.zip'.
-sE

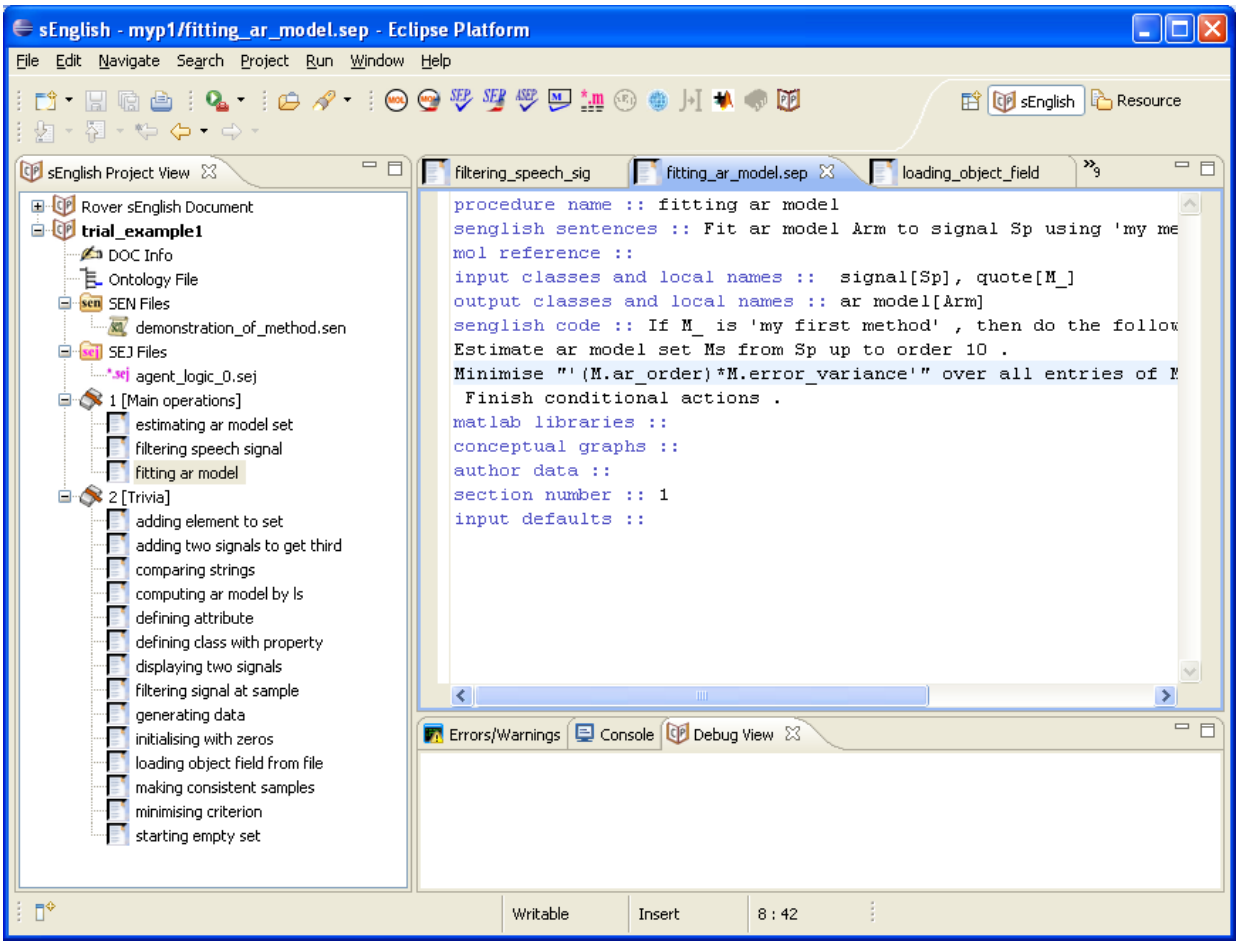
The latter section contains the URL declaration for the non-essential .sep file and .m file storage the content of which is automatically loaded by the executable section sEnglish project extractor utility (discussed later).

Finally, the whole algorithmic description of the paper becomes much more clear if conceptual classes used are also described. In our example paper these have been placed into the appendix of the paper as

sE-
Data classes: Main classes are signal, dynamical model, error, model set (cell). Subclass of signal is speech signal, white noise. Subclass of dynamical model is ar model, ar model set. Subclass of error is mean square error. Subclass of speech signal is noisy speech signal. Subclass of white noise is scalar white noise. Attributes of signal are sampling frequency(physical quantity), sample length(double), data(double). Attributes of ar model are ar order(double), error variance(double), ar parameters(double).
-sE

Note that the sEnglish sections have been placed between sE- and –sE signs that makes them unambiguously identifiable by the executable paper PDF reader. Extra care must also be taken to sue the right LaTex or MS Word (or other word processor) techniques so that the PDF file contains the right characters in its appearance as it has been in the `exesec.txt` file exported.

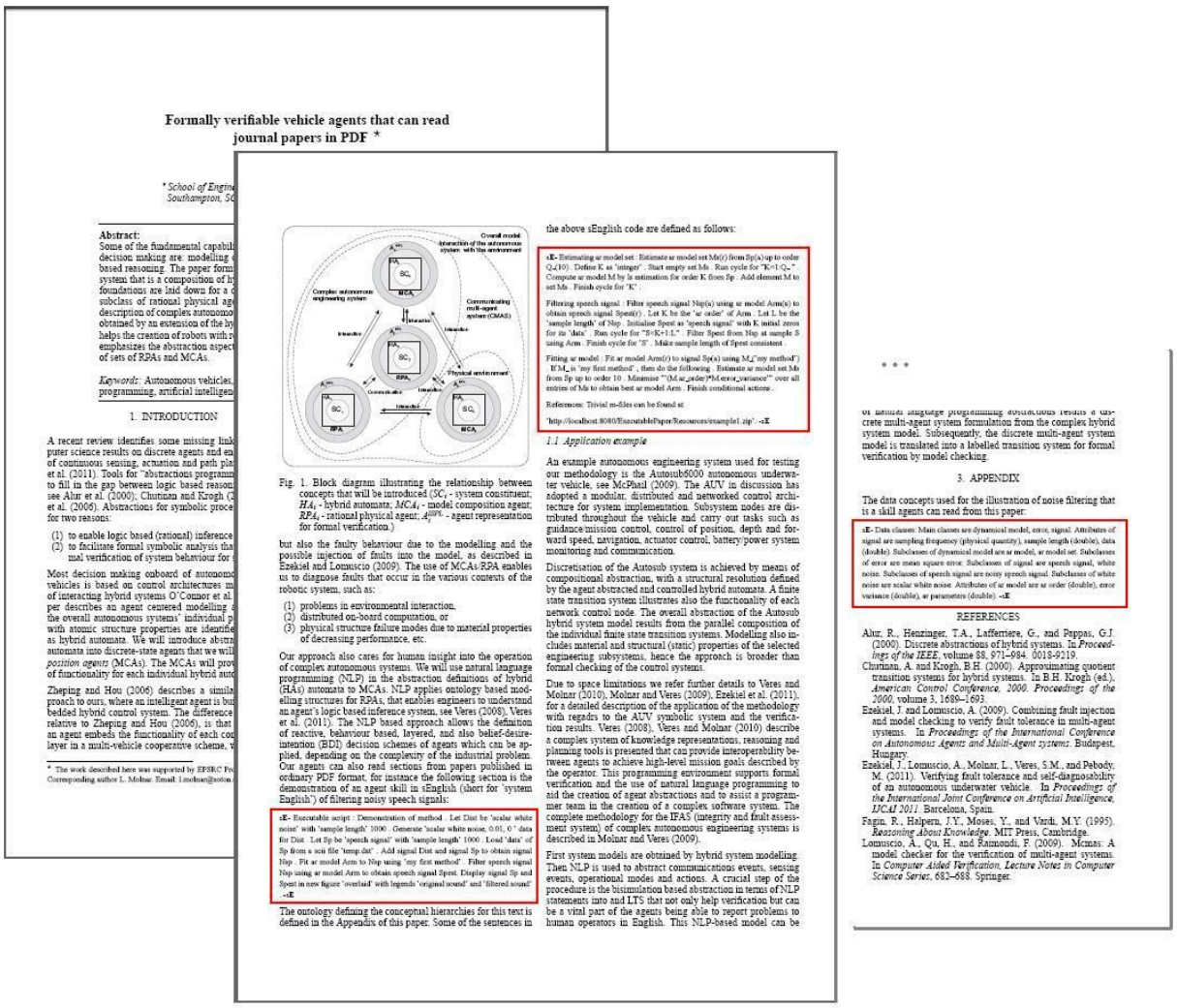
These sections are 100% equivalent with the original sEnglish project as displayed below.



and the project is essentially just embedded into a PDF paper, it can be extracted as described in the next section.

Importing from a publication

Readers of journal papers can come across sEnglish executable sections that can be extracted into a



Formally verifiable vehicle agents that can read journal papers in PDF *

1. INTRODUCTION

Fig. 1. Block diagram illustrating the relationship between concepts that will be introduced (SC_i - system constituent; HA_i - hybrid automata; MCA_i - model composition agent; RPA_i - rational physical agent; A_i^{REPR} - agent representation for formal verification.)

1.1 Application example

3. APPENDIX

REFERENCES

an sEnglish project. In the above example the red framed text , i.e. text in a paper between sE- and –sE delimiters, can jointly be extracted into an sEnglish project. The steps of this are as follows:

Step 1. All the section between sE- and –sE delimiters are extracted into a `<paper_name>.exp` ASCII file, in the same or arbitrary order.

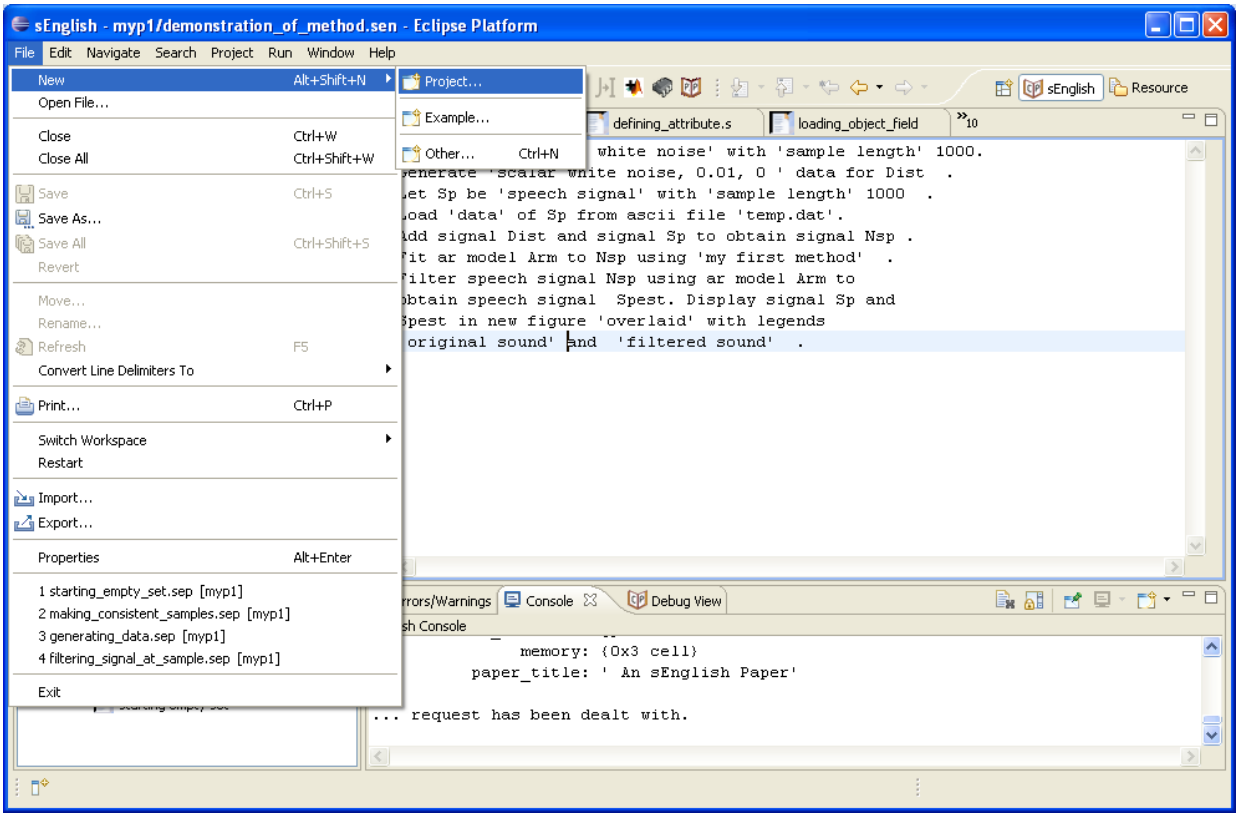
Step 2. A statement such as

References: Trivial m-files can be found at

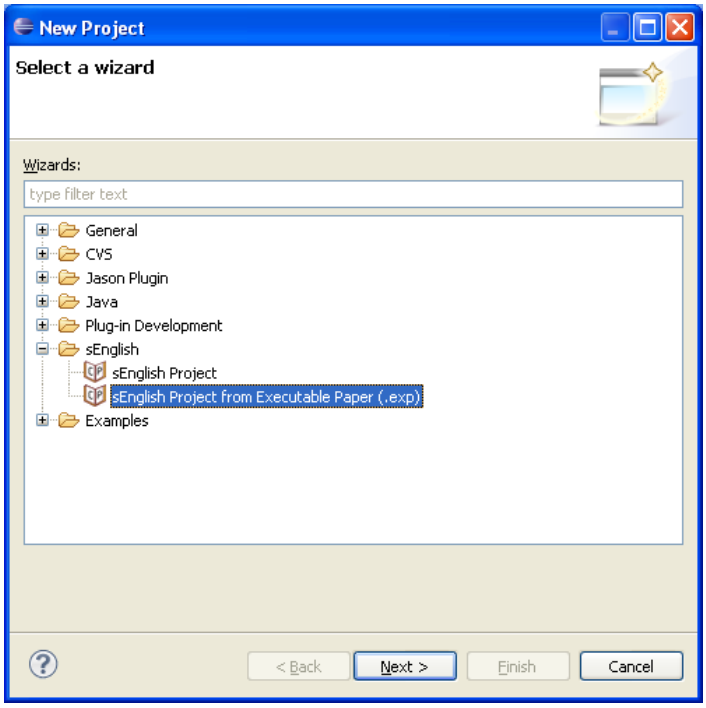
'http://localhost:8080/ExecutablePaper/Resources/example1.zip'. -sE

is sought that points to the URL of a single . zip file containing all the simple or standard operations that are not part of the journal papers contribution. These files are extracted into a user defined folder `...\<se_project>` for the sEnglish project to be created.

Step 3. Finally the `<paper_name>.exp` file is converted into an sEnglish project by starting a new project in the Eclipse environment using New >> Project to select the extractor wizard.

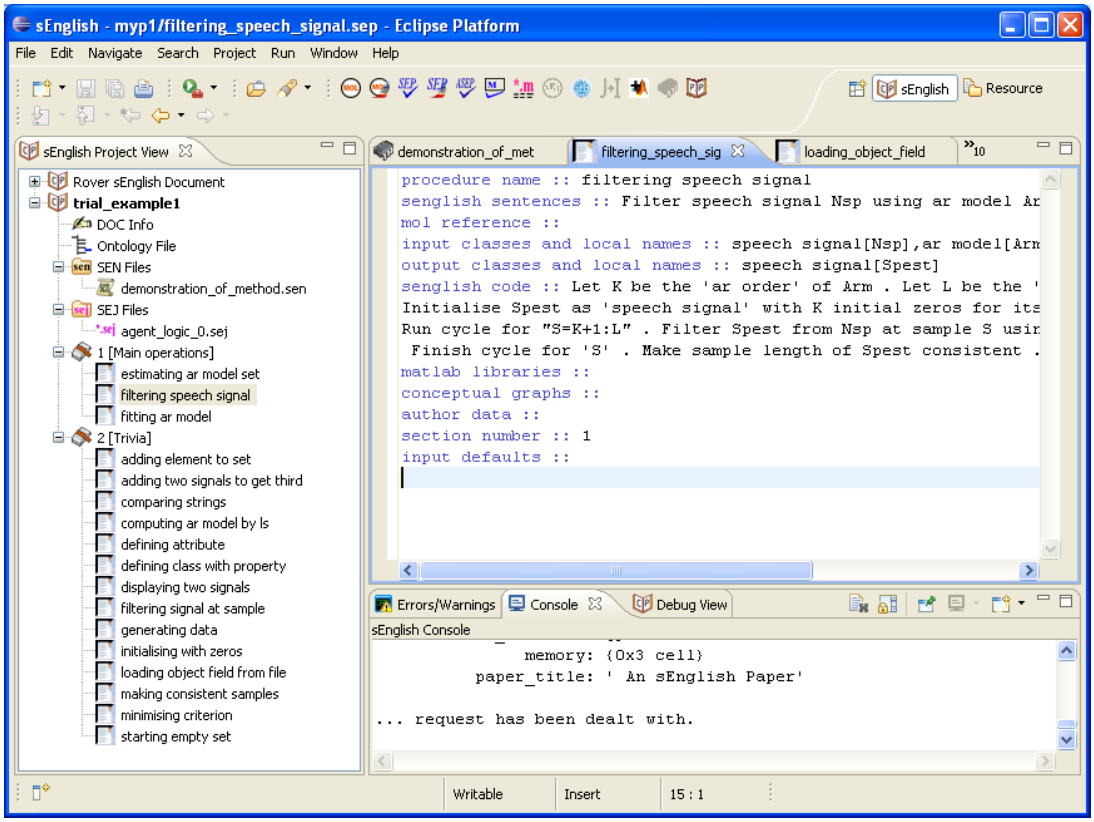


The *extractor wizard* is in the sEnglish folder of the New Project window.

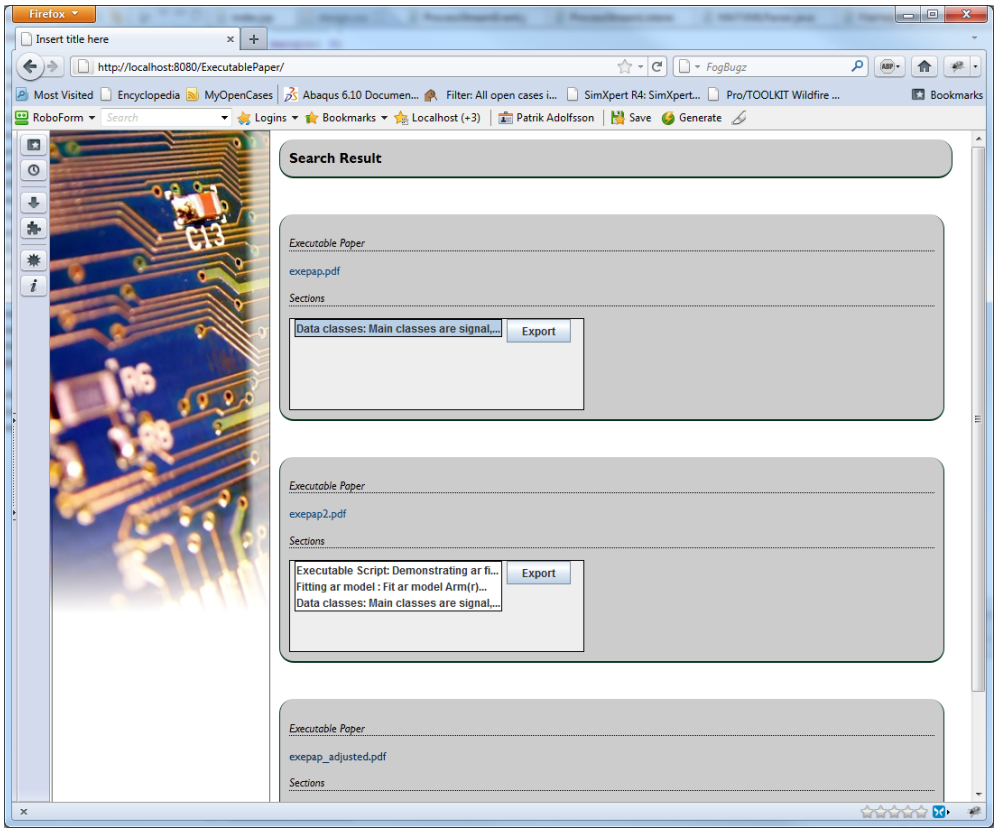


Choosing this wizard it will ask for a new Project Name and the location of the `<paper_name>.exp` .exp File file as in Step 1 above and finally it requires the definition of Location that of course needs to be same as `...\<se_project>` under (2) where the supporting `.sep` and `.m` files were extracted from the `.zip` file.

Having completed these steps on the example paper `exepap2.pdf`, for our example the resulting sEnglish project will be (as the original) where `<se_project>` = `trial_example1` :



There is however another and simpler way to extract sEnglish projects from journal papers. Publishers of journal papers can provide a facility such as displayed below:



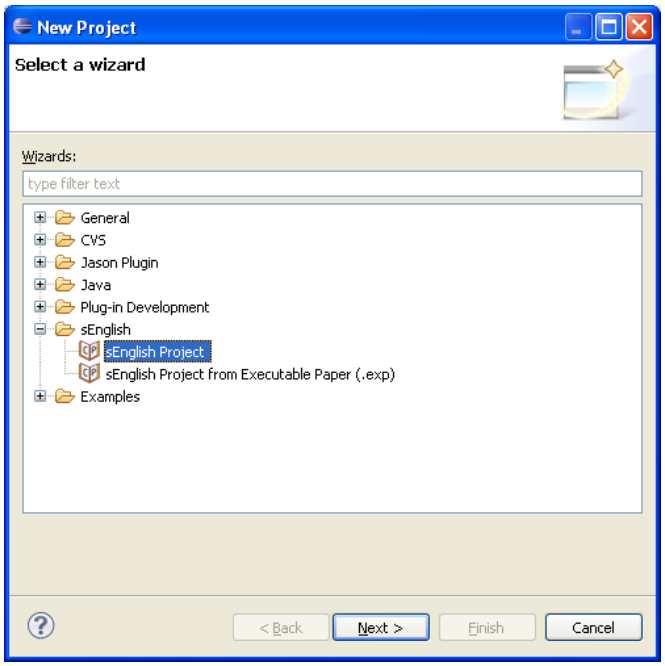
Such a facility can automatically extract an sEnglish project from a journal paper that is being looked up by a *subscribing reader* using a browser.

The steps of this are much simpler for the reader:

Step 1. Subscriber to the journal finds the paper on the publisher's website. The publisher's website automatically displays the extraction options as show (as an example) above.

Step 2. The Subscriber clicks on Export that brings up a window to define a folder where he or she would like to place the extracted sEnglish project.

Step 3. The Subscriber opens a new project using File >> Project in the newly defined folder using the new project wizard but this time selecting sEnglish Project .



Having created a new project by either method from a journal or a conference paper, the .sen scripts (and also newly written or modified ones by the subscriber) can be executed using after the due procedures of ontology compilation using and compilation of all .sep files into .m files using #

Exporting to HTML and Latex

Using the button, one can create HTML (/html) and LaTex (.tex) files within seconds from any project (that as a by-product also produces a .exp file of the project in the current directory.

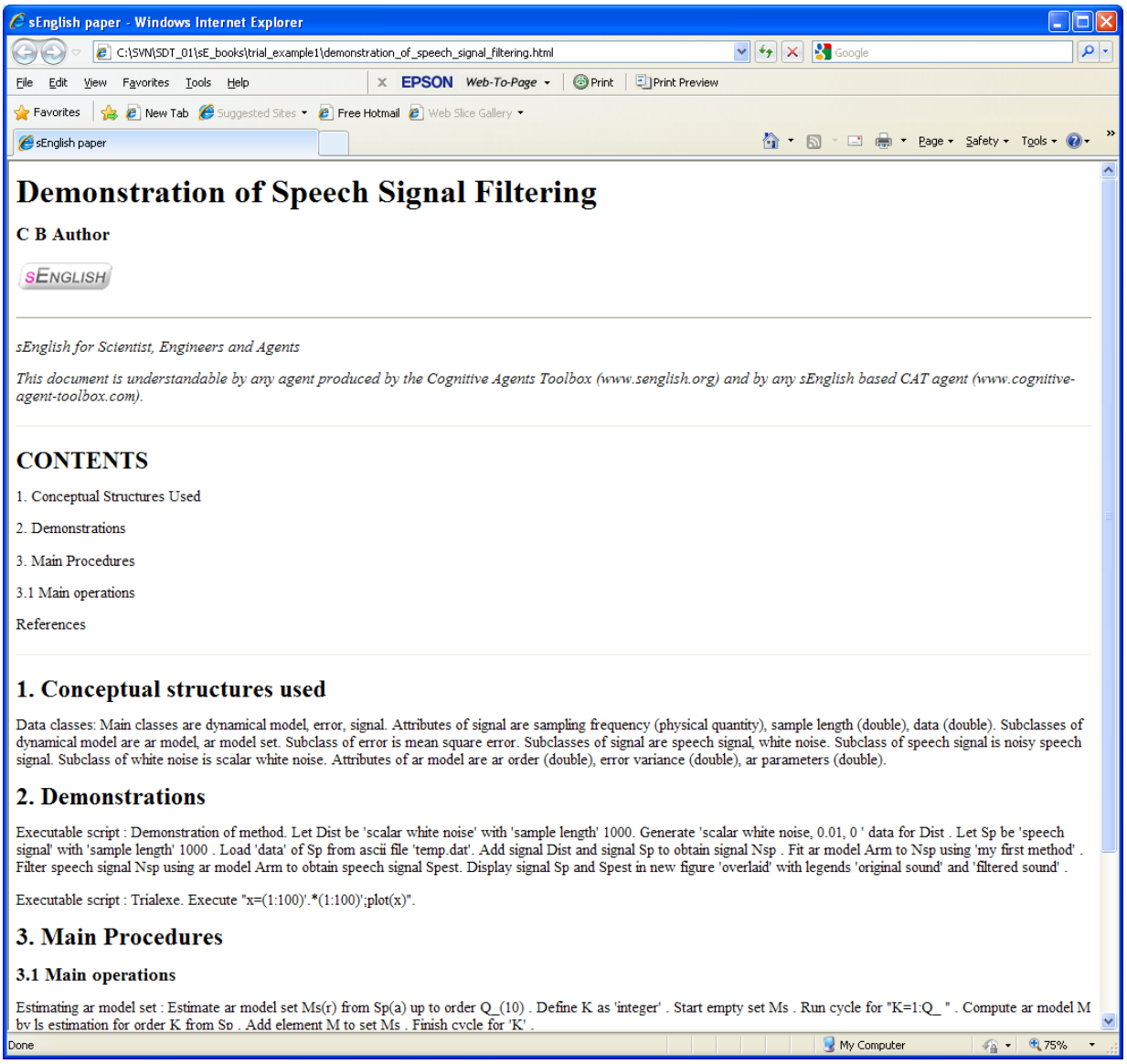
Step 1. The `DOCInfo.txt` file must contain comments about which sections of the project are to be detailed in their meaning in the exported .tex and .html files. For instance the `DOCInfo.txt` file

```
Document title::  Classical Control design
Author data:: C B Author
Section title 1:: Demos section (include in exe paper)
Section title 2:: Controller design (include in exe paper)
Section title 3:: Trivia
```

defines that only the first two sections are to be included with their procedures in the textual document in the HTML and LaTex files exported. The rest of the procedures in "Trivia" will be placed into a `.zip` file and referenced by a sentence of the format:

```
References: Trivial m-files can be found at
'http://localhost:8080/ExecutablePaper/Resources/example1.zip'.
```

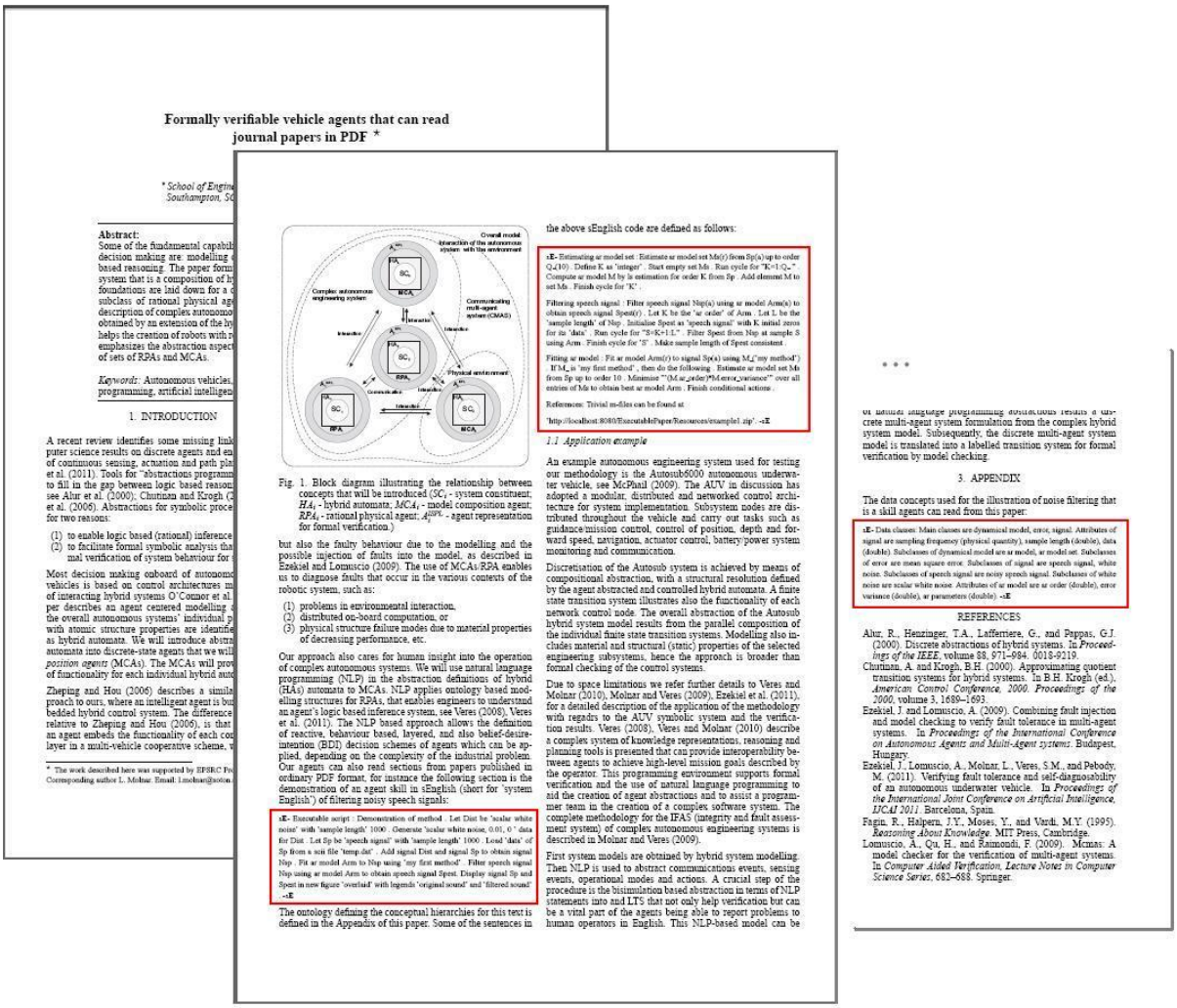
Through a URL on the Internet where the resource will be available for automated inclusion into a project imported from the HTML or PDF file of the reader of the publication. The trial_example1 project can be exported into a document/report/manual- like webpage :



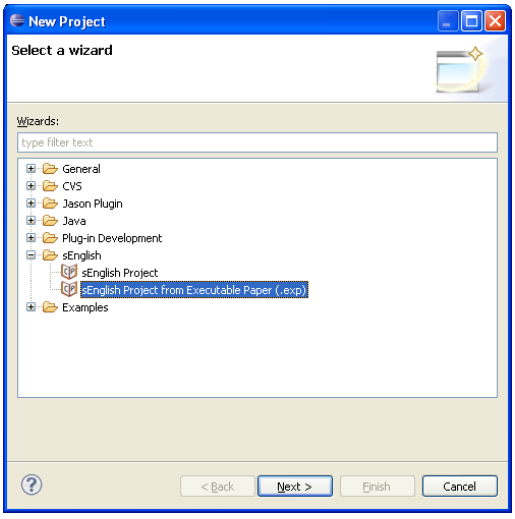
and into a .tex file that can be compiled into a PDF file. Note that the PDF file will also contain a .zip file reference for the library of trivial computations in chapters not commented by

Importing from HTML and PDF by copy/paste

PDF paper sEnglish sections as shown in the red frames below:



can be copy pasted into an ASCII file `<your file name>.exp` and new project can be started from that using File>>New>> New Project >> sEnglish wizard for "sEnglish Project from executable paper (.exp).".



Similarly, sEnglish sections can be copy/pasted from a web page into a .exp file and made into a project in the same way.

The methods presented here for creating executable papers and convert them into projects can equally well be applied to an entire book. Indeed a whole text book or monograph can be written with a lot of executable sections for the reader that can greatly enhance the value of the book as sEnglish projects can be experimented with in demonstrations to accompany the reading experience.

A more important benefit of Natural Language Programming (NLP) through sEnglish is the it can be used as *formal representation of knowledge* ***as well as*** *computer programming .*

Since computers are capable of sensing and feedback control, sEnglish provides the missing link between computer programming and *machines with knowledge*.

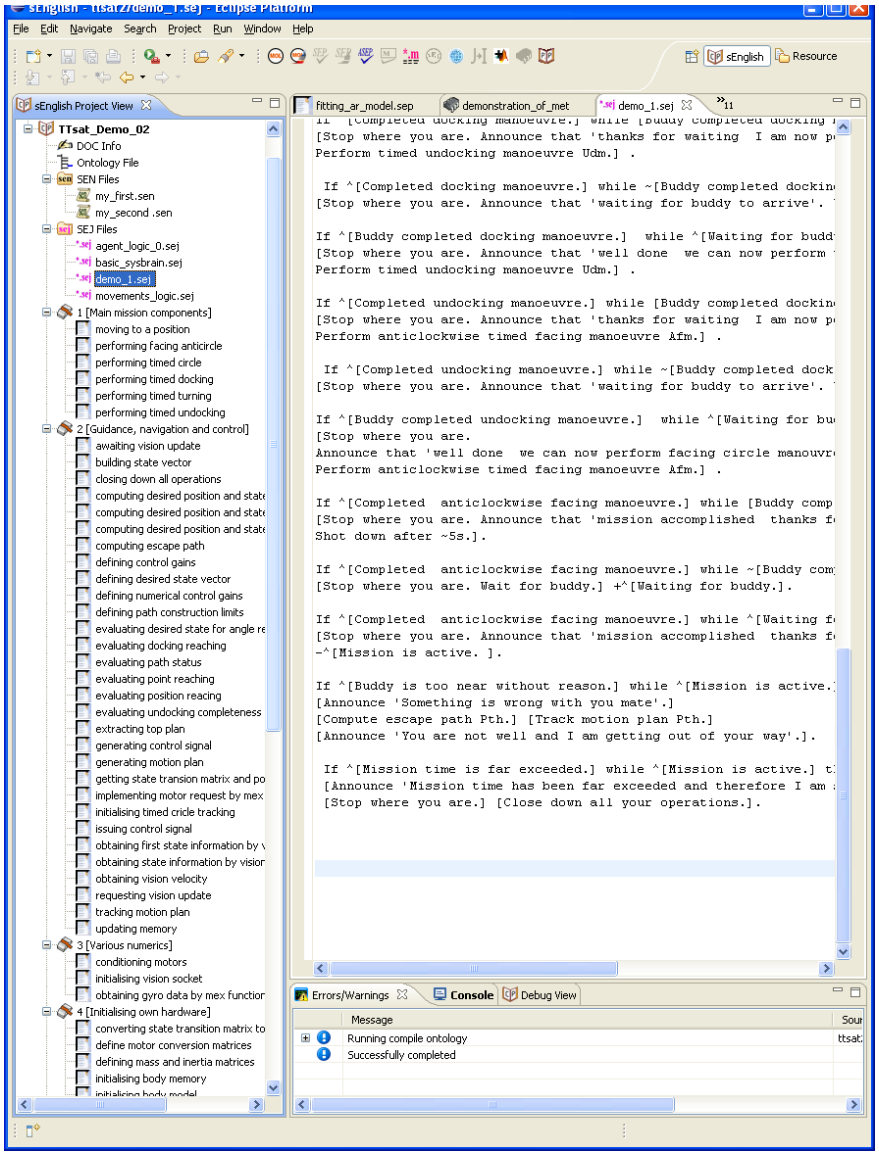
The software that can make machines to have the rudimentary forms of shared understanding with humans is the Cognitive Agent Toolbox that is introduced next.

Examples of Agent Reasoning

This chapter first provides a systematic overview of the most important concepts an agent to control a safe and useful autonomous vehicle or robot. This conceptual structuring is followed by generic ontology and sentence definitions to serve the agent's reasoning cycle in terms of initial actions, perception processes, reasoning processes that lead to formally analysable and provably correct behaviours. For simplicity, autonomous vehicles (UAVs, AUVs, AGVs, autonomous spacecraft, etc.), manufacturing, pick and place, healthcare, household, gardening and other robots will simply be called *machines* in this manual. The agents, the programming of which this manual is about, are those controlling machines operating in some structured, semi-structured or unstructured environment.

Machine reasoning

The `demo01.sej` file of the `TTsat_demo_02` project has a project window as follows:



Below is s printout of the `demo_1.sej` file that illustrates how the reasoning of an agent can be programmed. In the rest of this chapter we will look at this code in more detail and explain its various

parts. A consecutive section provides a systematic overview of the agent programming solutions that is followed by a second example on the programming of a rover.

```
INITIAL BELIEFS AND GOALS

~System is initialised.
Mission is pending.
~Mission is active.
~Buddy reached his starting point.
~Completed first circle manoeuvre.
~Buddy completed first circle manoeuvre.
~Completed rotation.
~Buddy completed rotation.
~Completed docking manoeuvre.
~Buddy completed docking manoeuvre.
~Completed  anticlockwise facing manoeuvre.
~Buddy completed anticlockwise facing manoeuvre.
~Being at position P0.
!Take initial actions.

INITIAL ACTIONS

Initialize system.
Read timed circle manoeuvre Tc from file 'c:\tcm1.txt'.
Get starting point P0 from Tc.
Read timed turning manoeuvre Tm from file 'c:\trn1.txt'.
Read timed docking manoeuvre Dm from file 'c:\doc1.txt'.
Read timed undocking manoeuvre Udm from file 'c:\und1.txt'.
Read timed facing manoeuvre Afm from file 'c:\afm1.txt'.


PERCEPTION PROCESSES

Monitor the following Booleans  :

Buddy is too near without reason.
Buddy appears out of control.
Buddy reached starting point.
Completed first circle manoeuvre.
Buddy completed first circle manoeuvre.
Completed rotation.
Buddy completed rotation.
Completed docking manoeuvre.
Buddy completed docking manoeuvre.
Completed anticlockwise facing manoeuvre.
Buddy completed anticlockwise facing manoeuvre.
Being at position P0.

Monitor the following objects  :

REASONING

^[Waiting for buddy.] implies that  ^[Mission is active.] .
^[Mission time is far exceeded.]implies that ~^[Mission is active.] .
^[Completed  anticlockwise facing manoeuvre.] implies that ^[Mission is active.] .

EXECUTABLE PLANS

If ^[System is initialised.] under the condition of ^[Mission is pending.] &
~^[Mission is active.] then do the following: [Announce that 'I am moving towards my
initial position'. Move to position P0.] -^[Mission is pending.].

If  ^[Being at position P0.] appears while ~^[Buddy reached his starting point.] then
[Wait for buddy.] +^[Waiting for buddy.].

If  ^[Being at position P0.] appears while ^[Buddy reached his starting point.] then
[Stop where you are. Announce that 'I am ready to go around'.] +^[Mission is active.]
[Perform timed circle manoeuvre Tc.].
```

```
If ^[Buddy reached his starting point.] while ^[Waiting for buddy.] then [Stop where
you are.] -^[Waiting for buddy.]  [Announce that 'I am ready to go around'. Perform
timed circle manoeuvre Tc. ].

 .... etc (see attached CD for the rest of the code)
```

A closer look at the example

It is split into initial beliefs and goals, perception processes, reasoning and executable plans:

```
INITIAL BELIEFS AND GOALS
...
...
INITIAL ACTIONS
...
...
PERCEPTION PROCESSES
Monitor the following Booleans
...
...
Monitor the following objects
...
...
REASONING
...
...
EXECUTABLE PLANS
...
...
```

Each of these sections and subsections need to be filled with sEnglish code of suitable syntax that is illustrated next.

This reasoning code in the `demo_1.sej` uses sentence definitions from the project `TTsat_Demo_02` the Eclipse window of which shown above. Using the buttons  and  one can compile all the sentences into m-files. The `demo_1.sej` file can be compiled into a `demo_1.asl` file by the use of the  button . The result of the compilation is shown later in this manual. In the next few section the work of the compiler is explained as it converts the `.sej` file into a `.asl` file part-by-part.

The section "INITIAL BELIEFS AND GOALS"

Each sentence in this section is converted into a predicate name full lower case and underscore between words. For instance

```
~System is initialised.
Mission is pending.
```

Compile into the Jason statements

```
~system_is_initialised .
mission_is_pending .
```

If there is proper name (words starting with capital) in the sentences than those are extracted, the rest of the sentence converted into a predicate and placed as an argument to the end in round brackets (...). For instance

```
~Being at position P0.
```

compiles into

```
~being_at_position(P0) .
```

There are two important points to note about initial beliefs and goals section:

(1) The predicates obtained for Jason are *mental notes* and not calls to any executive process. This implies that the sentences defined here may not have been defined in the associated sEnglish document, that is the project that contains the .sej file in question.

(2) These are statements are used to define what is "true" at the start of a Jason program (see use of Jason for agent programming in the next section).

The purpose of these mental notes is to provide *initial states* and context for triggering events that can activate applicable plans of the agent that are described later. For instance

```
Mission is pending.
~Mission is active.
```

are initial conditions. Also initial goals can be defined that are sentences starting with ! sign.

The initial actions and goals section must be closed by

```
!Take initial actions.
```

that instructs the use of the next section and compiles into

```
!take_initial_actions .
```

The section "INITIAL ACTIONS"

This section invokes sEnglish sentences that are defined in the associated sEnglish project. For instance sentences

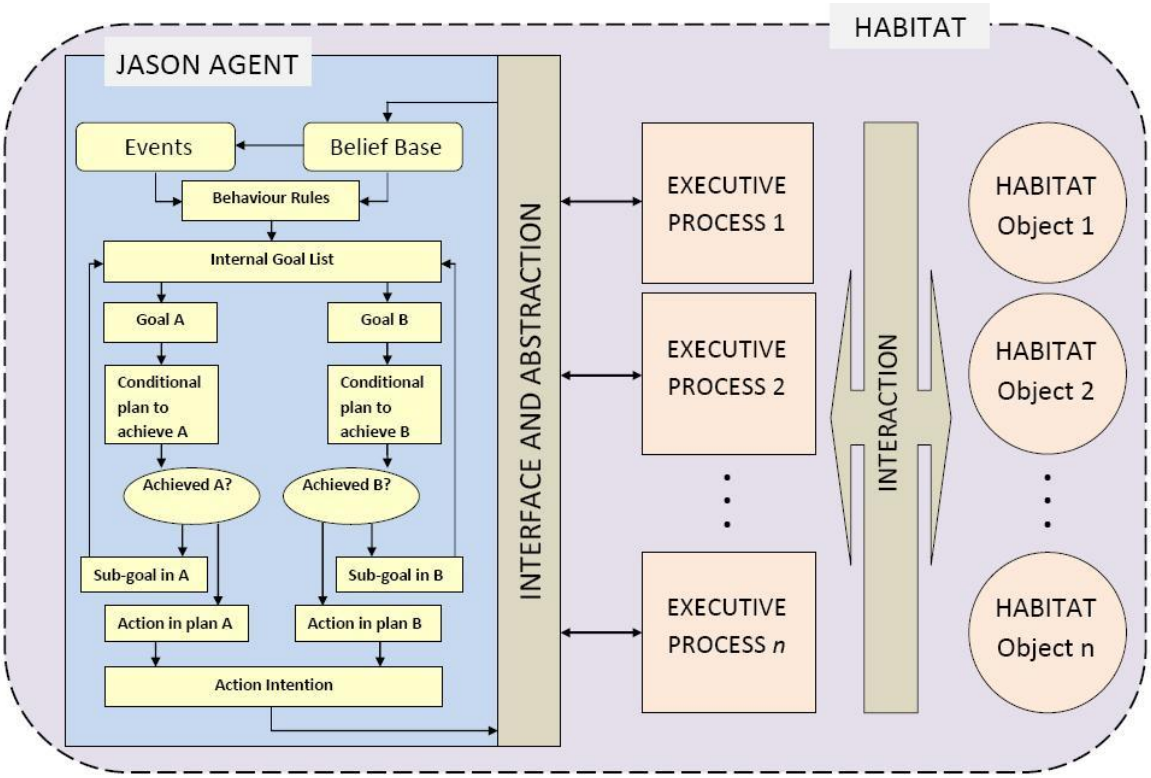
```
Initialize system.
Read timed circle manoeuvre Tc from file 'c:\tcm1.txt'.
Get starting point P0 from Tc.
```

compile into

```
+!take_initial_actions <-
invoke('ttsat01',runOnce,initializing_system,[],[]);
invoke('ttsat01',runOnce,reading_timed_path_from_file,['c:\tcm1.txt'],["Tc"]);
invoke('ttsat01',runOnce,getting_starting_point_from_path,["Tc"],["P0"]);
```

in Jason. Here the `ttsat01` refers to the MATLAB based executable project `ttsat01.exe` that is compiled as a library of sentence meanings that in MATLAB code can do such tasks as operating and actuator by turning it on and off or for instance running a feedback loop until some task is achieved. The following block diagram show the parallel operation of the JASON AGENT process that is in constant communication with the *executive processes* (based on embedded MATLAB code compiled into C) that are indicated as EXECUTIVE PROCESSes. The JASON AGENT is a multi-threaded BDI agent

program that does the "reasoning" with the symbols of actions carried out by the EXECUTIVE PROCESSes that are independent processes.



All these processes can either run on a single micro-processor or on a set of networked micro-processors. The computational demand of the executive processes dictates what number of microprocessors are used in an embedded application. It is up to the system designer to identify the computational bottleneck and eliminate them by parallel running processors so that realtime speed of processing sensor signals such as computer vision speech recognition and processing are sufficiently fast.

The *name of the executive process* that is associated with a sentence is defined in the meaning of the sentence (i.e. the "sEnglish code::" field in its .sep definition file) by a declaration such as

```
This sentence is interpreted by executive process 'ttsat01'.
```

or the "`Process, repeat mode::`" field in its `.sep` definition file contains the name of the associate `.exe` process as the first entry of a comma separated list. For instance

```
Process, repeat mode:: ttsat01, repeat
```

or

```
Process, repeat mode:: ttsat01
```

Are possible process declarations in the .sep file. Such a declaration needs to be inserted into all sentence meanings or .sep files process fields that will be executed by `'ttsat01'` .

The section "PERCEPTION PROCESSES"

Compilation of the sEJ-file (sEj is short for sEnglish/Jason code) section on perception processes starts with a standard part

```
// PERCEPTION PROCESSES

+!configureSystem:true <-
linkSystems(6253,'ttsat01');
linkSystems(6253,'ttsat02');
       +wide_awake;!monitorSystem.
+!monitorSystem:true <-
updateSystems;
!monitorSystem.
```

The Boolean monitoring sEJ section contains the following three example sentences to define continuous, i.e. periodic with the Jason agent's reasoning cycle, monitoring of some Boolean predicates for the reasoning of the agent:

```
Monitor the following Booleans  :

...
Completed first circle manoeuvre.
Buddy completed first circle manoeuvre.
Completed rotation.
...
```

This compiles into

```
+!monitorSystem <-
...
configureBoolean('ttsat01',completed_first_circle_manoeuvre);
configureBoolean('ttsat01',buddy_completed_first_circle_manoeuvre);
configureBoolean('ttsat01',completed_rotation);
...
```

These are calls to the 'ttsat01' process to check the validity of the respective statements. The ttsat01.exe can return the following three predicates into the belief base of the agent which are

```
completed_first_circle_manoeuvre
buddy_completed_first_circle_manoeuvre
completed_rotation
```

in case the Boolean evaluates to true, otherwise nothing is added to the belief base.

It should be noted at this stage that of you look ahead to the section EXECUTABLE PLANS of the example demo_1.sej file, then notice that the conditions:

```
^[Completed first circle manoeuvre.]
^[Buddy completed first circle manoeuvre.]
^[Completed rotation.]
```

occur in several plans. These compile precisely to the above three predicates and potentially trigger an action plan of the agent, depending on the context defined after the semicolon in plan definitions.

The section "REASONING"

This section is about logic inference, i.e. some combination of mental notes implies another. In `demo_1.sej` there are only 3 logic rules

```
^[Waiting for buddy.] implies that  ^[Mission is active.] .
^[Mission time is far exceeded.] implies that ~^[ Mission is active.] .
^[Completed  anticlockwise facing manoeuvre.] implies that ^[Mission is active.] .
```

which compile into the Prolog rules.

```
mission_is_active :- waiting_for_buddy.
mission_is_active :- ~mission_time_is_far_exceeded.
mission_is_active :- completed_anticlockwise_facing_manoeuvre.
```

The section "EXECUTABLE PLANS"

The first three plans of this section provide ample of opportunity to explain the standard parts of a plan:

```
If ^[System is initialised.] under the condition of ^[Mission is pending.] &
~^[Mission is active.] then do the following: [Announce that 'I am moving towards my
initial position'. Move to position P0.] -^[Mission is pending.].

If  ^[Being at position P0.] appears while ~^[Buddy reached his starting point.] then
[Wait for buddy.] +^[Waiting for buddy.].

If  ^[Being at position P0.] appears while ^[Buddy reached his starting point.] then
[Stop where you are. Announce that 'I am ready to go around'.] +^[Mission is active.]
[Perform timed circle manoeuvre Tc.].

If ^[Buddy reached his starting point.] while ^[Waiting for buddy.] then [Stop where
you are.] -^[Waiting for buddy.]  [Announce that 'I am ready to go around'. Perform
timed circle manoeuvre Tc. ].
```

Here the `^[System is initialised.]` , `^[Being at position P0.]` and `^[Buddy reached his starting point.]` are triggering events that can be mental notes appearing in the belief base. These can appear as a result of Boolean value monitoring in perception processes or can be valid by initial assumptions in INITIAL BELIEFS AND GOALS . These compile into `system_is_initialised`, `being_at_position(P0)` and `buddy_reached_his_starting_point` , respectively, by the `.sej` to `.asl` file compiler activated by the  button. A general rule is that

`^[Word1 word2 word3..]` compiles into word1_word2_word3…

as that may have been suspected from the previous examples. If ~ for "not" appears in front of [...] or ^[...] then that is carried over by the compiler. If there is no ^ in front of a bracketed sentence then the sEnglish project is looked up whether there is a sentence to match it.

If no ^ appears in front of [...] such as for instance

```
[Announce that 'I am moving towards my initial position'. Move to position P0.]
```

then the sentence definitions are sought in the document to match them (if there are no such sentences then the sEJ compiler gives an error message). The two sentences compile however differently as:

```
invoke('ttsat01',runOnce,announcing_something,['I am moving towards my initial
position'],[]);

invoke('ttsat01',runRepeated,moving_to_a_position,["P0"],[]);
```

The main difference appears in the use of `runOnce` and the `runRepeated` arguments of the invoke predicate. The first indicates single "open loop" action, the second means repeated execution of some action until something is achieved and the action is topped. The rest of the arguments of invoke are : <name of the logic predicate> , [<list of MATLAB call input arguments>],[<list of MATLAB call output arguments>].

Whether `runOnce`, `runRepeated` or `stopRepeated` is used is determined by the sEnglish to Jason compiler from the meaning of the sentence. If the meaning of the sentence contains

```
Repeat this activity until stopped.
```

or the sentence definition .sep file contain "repeat" in its "Process, repeat mode::" field then the type of the invoke command becomes `runRepeated` . To stop a repeated process by another sentence, its meaning must contain

```
This is a description of stopping this activity.
```

or the sentence definition `.sep` file contain "`stop`" in its "`Process, repeat mode::`" in the second entry of a comma separated list of its `.sep` file definition . If no declaration occurs about repeating or stopping in the .sep file, or its sEnglish code , then the invoke statement in Jason becomes of type `runOnce`.

Some of the possible plan formats are as follows:

```
If ^[A] appears under the condition of [B] then do the following:  ^[C].
```

that compiles to

```
+ A : B <-  C .
```

Several [...] or ^[...] conditions can be listed for the context after the "appears under the condition of". Similarly, several +^[...] or [...] can be listed after the "then do the following:" for the body of the plan.

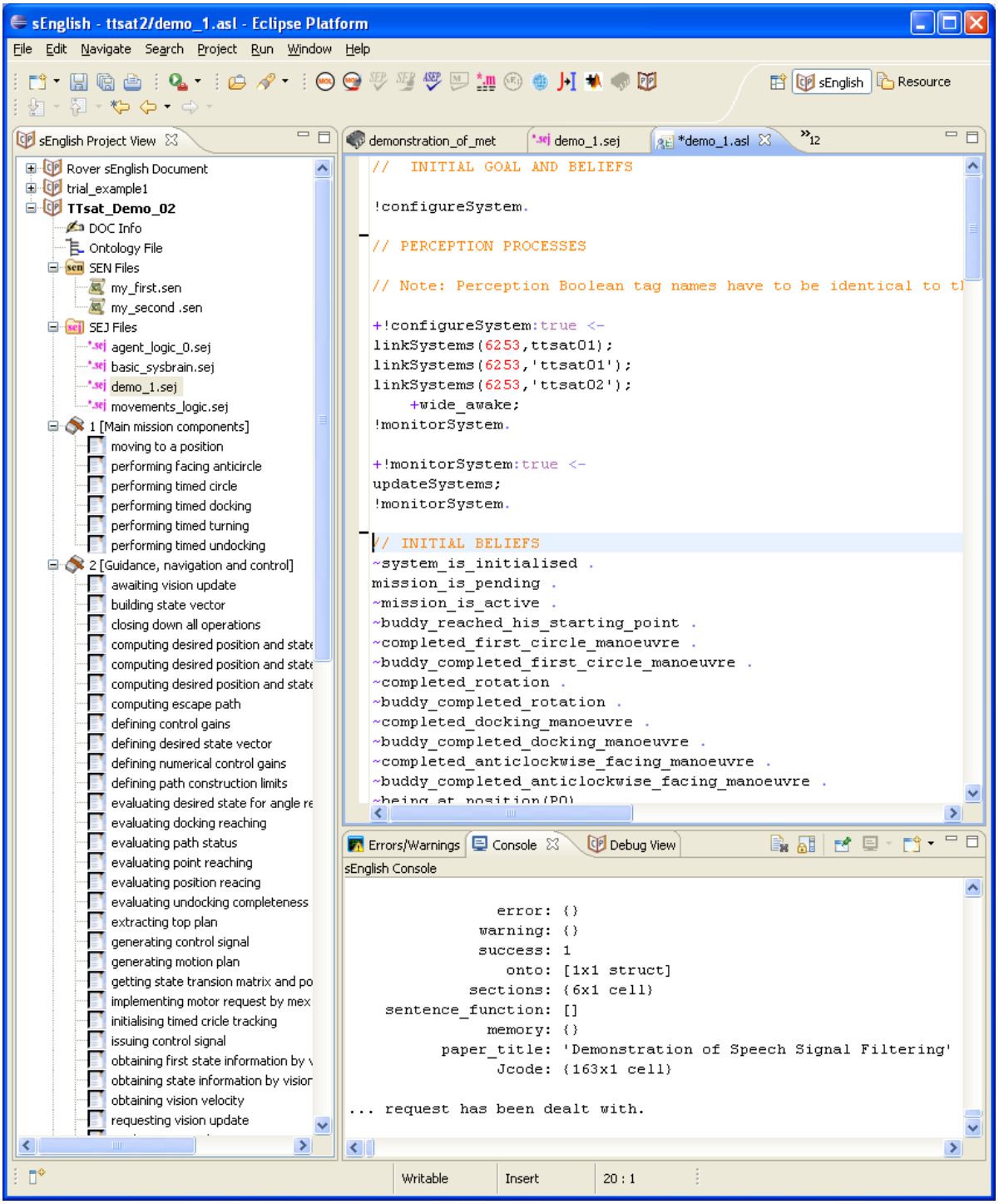
```
For the goal of ^[A] try to achieve [B] in repeated steps of [C] and finish with [D]
.

+! A : ~B <- C .
+B <- D.
```

The primary reason for using a construct like this is to detach a continuous MATLAB based feedback loop with the environment from the Jason code run until something what is stated in the goal is achieved. The constraint is that the goal must be a single statement combining both the success and failure of the attempt and derived by reasoning. Then next schema facilitates the separation of success or failure in one big statement.

Having compiled the *sEj-code* into a Jason agent demo_1.asl file can be loaded and inspected using

File >> Open File that can be used in the Jason plugin of Eclipse. Its detailed code is as follows:

```
//  INITIAL GOAL AND BELIEFS
!configureSystem.
// PERCEPTION PROCESSES
// Note: Perception Boolean tag names have to be identical to their sentence in lower
case.
+!configureSystem:true <-
linkSystems(6253,ttsat01);
linkSystems(6253,'ttsat02');
      +wide_awake;
!monitorSystem.
```




```
+!monitorSystem:true <-
```

```
updateSystems;
!monitorSystem.

// INITIAL BELIEFS
~system_is_initialised .
mission_is_pending .
~mission_is_active .
~buddy_reached_his_starting_point .
~completed_first_circle_manoeuvre .
~buddy_completed_first_circle_manoeuvre .
~completed_rotation .
~buddy_completed_rotation .
~completed_docking_manoeuvre .
~buddy_completed_docking_manoeuvre .
~completed_anticlockwise_facing_manoeuvre .
~buddy_completed_anticlockwise_facing_manoeuvre .
~being_at_position(P0) .
!take_initial_actions .
// INITIAL ACTIONS
+!take_initial_actions <-
invoke('ttsat01',runOnce,initializing_system,[],[]);
invoke('ttsat01',runOnce,reading_timed_path_from_file,['c:\tcm1.txt'],["Tc"]);
invoke('ttsat01',runOnce,getting_starting_point_from_path,["Tc"],["P0"]);
invoke('ttsat01',runOnce,reading_timed_path_from_file,['c:\trn1.txt'],["Tm"]);
invoke('ttsat01',runOnce,reading_timed_path_from_file,['c:\doc1.txt'],["Dm"]);
invoke('ttsat01',runOnce,reading_timed_path_from_file,['c:\und1.txt'],["Udm"]);
invoke('ttsat01',runOnce,reading_timed_path_from_file,['c:\afm1.txt'],["Afm"]).
// PERCEPTION PROCESSES
+!monitorSystem <-
configureBoolean('ttsat01',buddy_being_too_near_without_reason);
configureBoolean('ttsat01',buddy_appearing_our_of_control);
configureBoolean('ttsat01',buddy_reached_starting_point);
configureBoolean('ttsat01',completed_first_circle_manoeuvre);
configureBoolean('ttsat01',buddy_completed_first_circle_manoeuvre);
configureBoolean('ttsat01',completed_rotation);
configureBoolean('ttsat01',buddy_completed_rotation);
configureBoolean('ttsat01',completed_docking_manoeuvre);
configureBoolean('ttsat01',buddy_completed_docking_manoeuvre);
configureBoolean('ttsat01',completed_anticlockwise_facing_manoeuvre);
configureBoolean('ttsat01',buddy_completed_anticlockwise_facing_manoeuvre);
configureBoolean('ttsat01',being_at_position(P0)).
// REASONING
// EXECUTABLE PLANS
// *** new plan ***
+system_is_initialised :
mission_is_pending & ~mission_is_active <-
invoke('ttsat01',runOnce,announcing_something,['I am moving towards my initial
position'],[]);
invoke('ttsat01',runRepeated,moving_to_a_position,["P0"],[]);
-mission_is_pending .


// *** new plan ***
+being_at_position(P0) :
buddy_reached_his_starting_point <-
invoke('ttsat01', runRepeated,waiting_for_buddy,[],[]);
+waiting_for_buddy .
// *** new plan ***
+being_at_position(P0) :
buddy_reached_his_starting_point <-
invoke('ttsat02', runRepeated,waiting_for_buddy,[],[]); invoke('ttsat02',runOnce,
announcing_something,['I am ready to go around'],[]);
+mission_is_active;
invoke('ttsat01',runRepeated,performing_timed_circle,["Tc"],[]) .
// *** new plan ***
+buddy_reached_his_starting_point :
waiting_for_buddy <-
invoke('ttsat01', runRepeated,waiting_for_buddy,[],[]);
-waiting_for_buddy;
```

```
invoke('ttsat01',runOnce,announcing_something,['I am ready to go around'],[]);
invoke('ttsat01', runRepeated,
performing_timed_circle,["Tc"],[]) .
  ...
```

where we have truncated this Jason code as it is too long for this manual and the above illustrates the kind of Jason code one can obtain by the sEj compiler.

This piece of code very much follows a finite state machine logic by executing one movement after th other. The challenge is to involve more reasoning in Jason codes instead of prescribing what steps each individual goal achievement requires. The avoid the pitfalls of rigid finite state machine like, i.e. FSS-like programming, the following sections first describe the useful conceptual base of machines followed by a quantised logic that enables a richer set of transitions than for instance StateFlow™ based definition of hybrid control systems.

A more mature version of the agent program can be found in `demo_2.sej` that puts more emphasis on logic based reasoning to decide what to do next and has a small number of plans associated with physical skills.

```
Listing of demo_2.sej for agent reasoning.

INITIAL BELIEFS AND GOALS

~System is initialised.
Mission is pending.
~Mission is active.
~Buddy reached his starting point.
~Completed first circle manoeuvre.
~Buddy completed first circle manoeuvre.
~Completed rotation.
~Buddy completed rotation.
~Completed docking manoeuvre.
~Buddy completed docking manoeuvre.
~Completed  anticlockwise facing manoeuvre.
~Buddy completed anticlockwise facing manoeuvre.
~Being at position P0.
!Take initial actions.

INITIAL ACTIONS

Initialize system.
Read timed circle manoeuvre Tc from file 'c:\tcm1.txt'.
Get starting point P0 from Tc.
Read timed turning manoeuvre Tm from file 'c:\trn1.txt'.
Read timed docking manoeuvre Dm from file 'c:\doc1.txt'.
Read timed undocking manoeuvre Udm from file 'c:\und1.txt'.
Read timed facing manoeuvre Afm from file 'c:\afm1.txt'.


PERCEPTION PROCESSES

Monitor the following Booleans  :

Arrived at position P0.
Buddy is too near.
Buddy reached starting point.
Completed first circle manoeuvre.
Buddy completed first circle manoeuvre.
Completed rotation.
```

Buddy completed rotation.
Completed docking manoeuvre.
Buddy completed docking manoeuvre.
Completed anticlockwise facing manoeuvre.
Buddy completed anticlockwise facing manoeuvre.

Monitor the following objects :

REASONING

If ^[Mission is pending.] & ~^[Mission is active.] then ^[Ready to move to initial position P0.].
The fact ^[Arrived to position P0.] implies ~^[Mission is pending.] .
When ^[Arrived to position P0.] & ^[Buddy reached starting point.] & ~^[Completed first circle manoeuvre.] then ^[Ready for first circle manouevre.].
When ^[Completed first circle manoeuvre.] & ^[Buddy completed first circle manoeuvre.] then ^[Ready for turning around.].
When ^[Completed rotation.] & ^[Buddy completed rotation.] then ^[Ready for docking manouevre.].
When ^[Completed docking manoeuvre.] & ^[Buddy completed docking manoeuvre.] then ^[Ready for undocking manouevre.].
When ^[Completed undocking manoeuvre.] & ^[Buddy completed undocking manoeuvre.] then ^[Ready for facing circle manouevre.].
When ^[Completed facing circle manoeuvre.] & ^[Buddy completed facing circle manoeuvre.] then ^[Completed mission.].
If ^[Completed mission.] then ~^[Mission is active.].
Fact ^[Buddy is too near.] & ~^[Performing docking.] requires ^[Ready to modify manouevre to avoid buddy.].

EXECUTABLE PLANS

If ^[Ready to move to initial position P0.] while ~^[Mission is active.] then
[Announce that 'I am moving towards my initial position'.] [Move to position P0.]
[Stop where you are.] +^[Mission is active.].

If ^[Ready for first circle manouevre.] while ^[Mission is active.] then
[Announce that 'I am ready to go around'.] [Perform timed circle manoeuvre Tc.]
[Stop where you are.].

If ^[Ready for turning around.] while ^[Mission is active.] then
[Announce that 'thanks for waiting I am now turning around'.]
[Perform timed turning manoeuvre Trn1.] [Stop where you are.].

If ^[Ready for docking manouevre.]. while ^[Mission is active.] then
+^[Performing docking.]
[Announce that 'thanks for waiting now performing docking manouvre'.]
[Perform timed docking manoeuvre Dm.] [Stop where you are.].

If ^[Ready for undocking manouevre.]. while ^[Mission is active.] then
[Announce that 'well done we can now perform undocking manouvre'.] [Perform timed undocking manoeuvre Udm.] [Stop where you are.] .

If ^[Ready for facing circle manouevre.]. while ^[Mission is active.] then
[Stop where you are.] [Announce that 'thanks for waiting I am now performing facing circle manouvre'.]
[Perform anticlockwise timed facing manoeuvre Afm.] [Stop where you are.] .

```
If ^[Ready to modify  manouevre to avoid buddy.] while ^[Mission is active.]
then
[Announce 'Something is wrong with you mate'.] [Compute escape path Pth.]
[Track motion plan Pth.]
[Announce 'You are not well and I am getting out of your way'.].

 If ^[Mission time is far exceeded.] while ^[Mission is active.] then
 [Announce 'Mission time has been far exceeded and therefore I am
stopping'.] [Stop where you are.] [Close down all your operations.].
```

The corresponding compiled .asl file Jason code is as follows:

```
//  INITIAL GOAL AND BELIEFS

!configureSystem.


// PERCEPTION PROCESSES

// Note: Perception Boolean tag names have to be identical to their sentence
in lower case.

+!configureSystem:true <-
linkSystems(6253,ttsat01);
linkSystems(6253,ttsat02);
configureBoolean(ttsat01,being_at_position(P0));
configureBoolean(ttsat01,buddy_being_too_near_without_reason);
configureBoolean(ttsat01,buddy_reached_starting_point);
configureBoolean(ttsat01,completed_first_circle_manoeuvre);
configureBoolean(ttsat01,buddy_completed_first_circle_manoeuvre);
configureBoolean(ttsat01,completed_rotation);
configureBoolean(ttsat01,buddy_completed_rotation);
configureBoolean(ttsat01,completed_docking_manoeuvre);
configureBoolean(ttsat01,buddy_completed_docking_manoeuvre);
configureBoolean(ttsat01,completed_anticlockwise_facing_manoeuvre);
configureBoolean(ttsat01,buddy_completed_anticlockwise_facing_manoeuvre).
.print("Called all configure booleans...");
!take_initial_actions.
// INITIAL BELIEFS
~system_is_initialised .
mission_is_pending .
~mission_is_active .
~buddy_reached_his_starting_point .
~completed_first_circle_manoeuvre .
~buddy_completed_first_circle_manoeuvre .
~completed_rotation .
~buddy_completed_rotation .
~completed_docking_manoeuvre .
~buddy_completed_docking_manoeuvre .
~completed_anticlockwise_facing_manoeuvre .
~buddy_completed_anticlockwise_facing_manoeuvre .
~being_at_position(P0) .
!take_initial_actions .
// INITIAL ACTIONS
+!take_initial_actions <-
invoke(ttsat01,runOnce,initializing_system,[],[]);
invoke(ttsat01,runOnce,reading_timed_path_from_file,['c:\tcm1.txt'],["Tc"]);
invoke(ttsat01,runOnce,getting_starting_point_from_path,["Tc"],["P0"]);
invoke(ttsat01,runOnce,reading_timed_path_from_file,['c:\trn1.txt'],["Tm"]);
invoke(ttsat01,runOnce,reading_timed_path_from_file,['c:\doc1.txt'],["Dm"]);
```

```
invoke(ttsat01,runOnce,reading_timed_path_from_file,['c:\und1.txt'],["Udm"])
;
invoke(ttsat01,runOnce,reading_timed_path_from_file,['c:\afm1.txt'],["Afm"])
.
// REASONING
ready_to_move_initial_position(P0) :-  mission_is_pending &
~mission_is_active .
~mission_is_pending :-  arrived_to_position(P0) .
ready_for_first_circle_manouevre :-  arrived_to_position(P0) &
buddy_reached_starting_point & ~completed_first_circle_manoeuvre .
ready_for_turning_around :-  completed_first_circle_manoeuvre &
buddy_completed_first_circle_manoeuvre .
ready_for_docking_manouevre :-  completed_rotation &
buddy_completed_rotation .
ready_for_undocking_manouevre :-  completed_docking_manoeuvre &
buddy_completed_docking_manoeuvre .
ready_for_facing_circle_manouevre :-  completed_undocking_manoeuvre &
buddy_completed_undocking_manoeuvre .
completed_mission :-  completed_facing_circle_manoeuvre &
buddy_completed_facing_circle_manoeuvre .
~mission_is_active :-  completed_mission .
ready_to_modify_manouevre_to_avoid_buddy :-  buddy_is_too_near &
~performing_docking .
// EXECUTABLE PLANS
// executable plan :
+ready_to_move_initial_position(P0) :
mission_is_active <-
invoke(ttsat02,runOnce,announcing_something,['I am moving towards my initial
position'],[]);
invoke(ttsat01,runOnce,moving_to_a_position,["P0"],[]);
invoke(ttsat01,runOnce,waiting_for_buddy,[],[]);
+mission_is_active .
// executable plan :
+ready_for_first_circle_manouevre :
mission_is_active <-
invoke(ttsat02,runOnce,announcing_something,['I am ready to go around'],[]);
invoke(ttsat01,runOnce,performing_timed_circle,["Tc"],[]);
invoke(ttsat01,runOnce,waiting_for_buddy,[],[]) .
// executable plan :
+ready_for_turning_around :
mission_is_active <-
invoke(ttsat02,runOnce,announcing_something,['thanks for waiting I am now
turning around'],[]);
invoke(ttsat01,runOnce,performing_timed_turning_manoeuvre_trn,["Trn1"],[]);
invoke(ttsat01,runOnce,waiting_for_buddy,[],[]) .
// executable plan :
+ready_for_docking_manouevre :
mission_is_active <-
performing_docking;
invoke(ttsat02,runOnce,announcing_something,['thanks for waiting now
performing docking manouvre'],[]);
invoke(ttsat01,runOnce,performing_timed_docking,["Dm"],[]);
invoke(ttsat01,runOnce,waiting_for_buddy,[],[]) .
// executable plan :
+ready_for_undocking_manouevre :
mission_is_active <-
invoke(ttsat02,runOnce,announcing_something,['well done we can now perform
undocking manouvre'],[]);
invoke(ttsat01,runOnce,performing_timed_undocking,["Udm"],[]);
invoke(ttsat01,runOnce,waiting_for_buddy,[],[]) .
// executable plan :
+ready_for_facing_circle_manouevre : mission_is_active <-
invoke(ttsat01,runOnce,waiting_for_buddy,[],[]);
```

```
invoke(ttsat02,runOnce,announcing_something,['thanks for waiting I am now
performing facing circle manouvre'],[]);
invoke(ttsat01,runOnce,performing_facing_anticircle,["Afm"],[]);
invoke(ttsat01,runOnce,waiting_for_buddy,[],[]) .
// executable plan :
+ready_to_modify_manouevre_to_avoid_buddy : mission_is_active <-
invoke(ttsat02,runOnce,announcing_something,['Something is wrong with you
mate'],[]);
invoke(ttsat01,runOnce,computing_escape_path,[],["Pth"]);
invoke(ttsat01,runRepeated,tracking_motion_plan,[]);
invoke(ttsat02,runOnce,announcing_something,['You are not well and I am
getting out of your way'],[]) .
// executable plan :
+mission_time_is_far_exceeded : mission_is_active <-
invoke(ttsat02,runOnce,announcing_something,['Mission time has been far
exceeded and therefore I am stopping'],[]);
invoke(ttsat01,runOnce,waiting_for_buddy,[],[]);
invoke(ttsat01,runOnce,closing_down_all_operations,[],[]) .
```

This Jason code has special invoke calls, context handling and reception of new beliefs from parallel executive processes written in MATLAB/Simulink/C++ and hence can be run under a generic Jason environment. The connection between this type of Jason code and the executive processes can be established using the Agent Executive Toolbox as described in a follow on section. Next a general introduction is provided into non-specialised, generic language and interpretation of Jason code.

Jason programming

After the sEnglish based *.sej file is compile into a *.asl file all debugging can be carried out by running the Jason agent defined by .asl file with MATLAB/Simulink. Hence in this section we direct our attention to the syntax and semantics of pure Jason.

Jason is a well developed BDI (beliefs-desires-intentions) agent-oriented development language derived from Agentspeak by NASA, and is suitable for programming intelligent agents that are capable of reasoning. A rational agent can consist of a combination of Jason code presented using sEnglish sentences to form human readable intelligent agent rationale. This section provides an introduction into the principles of programming agents in Jason and the use of an Eclipse based *Jason plug-in* that can be conveniently used in conjunction with the *sEditor* .

Jason and AgentSpeak

Jason is a Java based interpreter for an *extended version of AgentSpeak(L)*[3]. It provides platform independent development for multi-agent systems. The most notable extensions of AgentSpeak(L) are the replacement of atoms with literals as described in formal syntax rules later in this section. Improvements over AgentSpeak are the permission for terms to contain variables, lists or strings and that bound variables may be used as literals. Additionally, atomic formulae may be annotated to encapsulate more information to be used within the reasoning cycle and a substantial subset of Prolog[4] is supported within the belief base. Operationally, Jason is attractive since it permits distribution of operational agents with inter-agent communication provided by SACI[5] or JADE[6]. Jason is available as Open Source, distributed under the GNU LGPL, and is well documented both online in the original Jason Manual[7] and off-line in the much more in-depth book[8] authored by Bordini, Hubner and Wooldridge. Due to the existence this documentation, only a cursory overview of Jason and its capabilities will be entered upon: the interested reader is referred to the original publishing that were previously mentioned.

At its most basic level, an agent prescribed within AgentSpeak(L) is a set of beliefs, perception updates. rules and a set of plans that can also take actions in the physical world. Quite intuitively, the 'beliefs' are what the agent perceives the state of the inhabited environment to be, described as a *first order predicate*[9] or its negation. Plans, within the AgentSpeak(L) syntax, represent basic actions that the agent may perform on its environment (or on a model of the environment) in order to achieve a particular goal: a plan may be initiated upon perception of a particular triggering event, which may be

[3] Information on AgentSpeak can be found at URL: ... (Jan 2012)
[4] Prolog is a logic programming language (see tutorial introduction at URL .... (Jan 2012)
[5] SACI is ...
[6] JADE is ....
[7] URL of Jason manual: http://jason.sourceforge.net/Jason.pdf (Jan 2012)
[8] URL of book on Jason: http://jason.sourceforge.net/jBook/jBookWebSite/Home.php (Jan 2012)
[9] Predicates in First Order Logic (FOL) , see for instance text books:....

internal (from a subgoal) or external (through a perception update). For a plan to run, it must be deemed relevant by the agent: here relevance is determined by the agent through logical consequence of the agent's current beliefs. Goals themselves may be an achievement goal, or a test goal: achievement goals are those where the agent wishes to achieve a particular state of the environment (or its model) ; test goals return a unification for the declared predicate with one of the agent's beliefs. A core item relating to the BDI notation is that of an intention: an intention is a particular course of action to which an agent has committed itself to as a result of perceived events and may itself be considered a stack of partially instantiated plans.

Now let us consider a reasoning cycle that the AgentSpeak interpreter completes for a particular agent program: at each cycle, a list of events is updated. These events may be generated as a consequence of environment perception, or reaching a subgoal within a particular plan body when executing a current intention. The set of triggering events is NOT the same as the set of beliefs. There are also internal mental notes that can be subject for logical operations that can imply some triggering events when in combination with other events. Mental notes obtained by logical derivation are however only kept in the belief base for one reasoning cycle. Those obtained by perception are kept until taken out by a – operator in a plan (placed in front of a square bracket as –[...] when an sEnglish sentence is used for the mental note) .

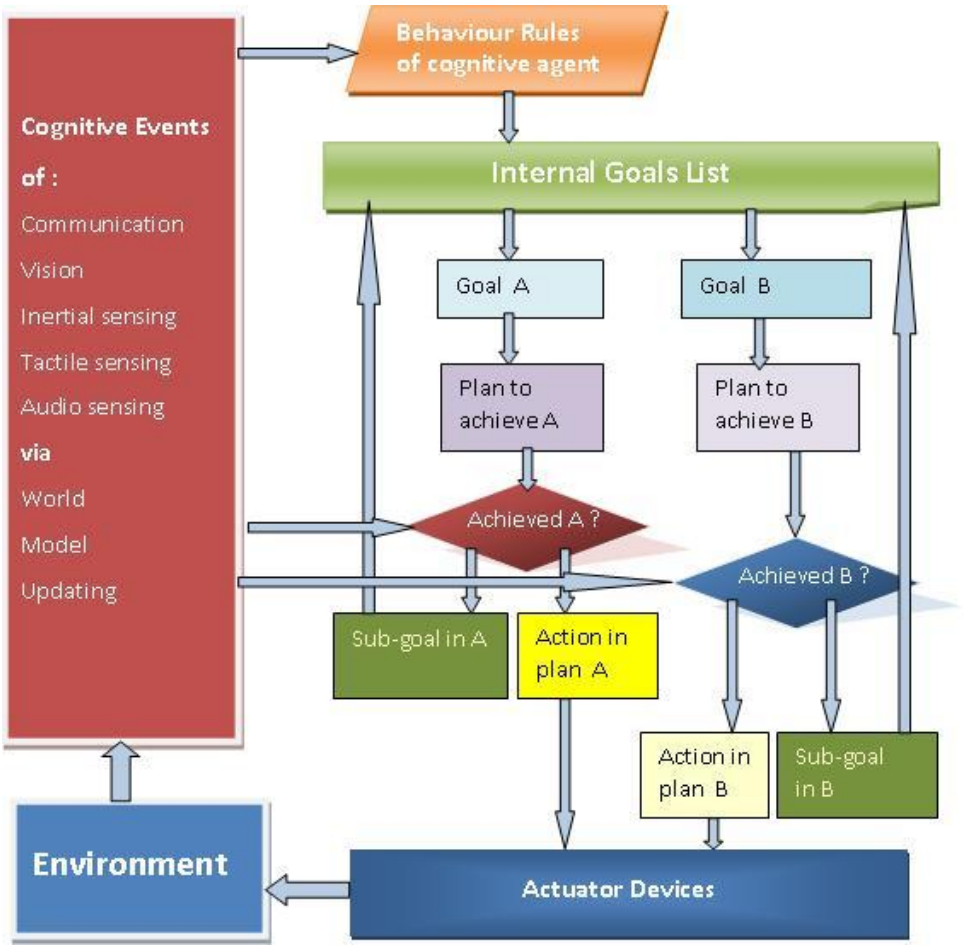
It is assumed that beliefs are updated via some form of environmental perception and whenever there is a change in the agent's belief set, an event may be generated by the agents reasoning rules. Upon selection of an event to be dealt with, the interpreter must unify the event with a particular triggering event to determine a set of relevant plans. This unification may involve substitution of scalar or string values to Jason variables. Applicable plans are extracted from the set of relevant plans by checking if the context of the plans follow from the agents belief base. A *single applicable plan is then selected* from the set and is placed conditionally into the set of intentions: internally generated events result in pushing the plan onto an existing intention; externally generated events result in the creation of a new intention. Intentions have previously been described as courses of action to which the agent is committed, as the intentions are executed through the implementation of instantiated plans, the list of intentions is modified to reflect this. The AgentSpeak(L) cycle then repeats.

Agent capability is obviously determined by the wealth of plans availed to the agent by the programmer(s). These plans, their trigger events and guard conditions must all flow through logical consequence; a key point is that neither Jason nor AgentSpeak(L) can guarantee logical consistency. Consequently it is the responsibility of the programmer(s) involved in the agent development to ensure the logical flow of the agent program and its correct operation. This will involve looking through the Jason code and manually checking all possible outcomes; there are no formal verification tools available with standard Jason. Note, however, that formal verification tools are available within the extended version of CAT that requires a specific format of Jason programming and annotations, without limiting agent capabilities. Extended CAT contains a compiler of this specific Jason+sE code into ISPL (interpreted system programming language), the language of the multi-agent model checker (MCMAS). Sysbrains are special agent architectures that are verifiable by design and their sEnglish/Jason code abstracts into a finite state multi-agent representation that can be formally verified for interesting CTL (computational tree logic) formulae.

Here we are concerned primarily with applied and cognitive agent systems, rather than the abstract workings of theoretical agent. We wish to develop a Jason agent that may operate within a *real habitat* (here it is intended to avoid using the reference of 'environment' to define the (physical) operational arena of an agent, since within Jason the term 'environment' is used to define a Java based world), with real requirements of action and where this action is carried out in real time. Moreover, we wish to develop a system that is intrinsically understandable and may be formally verified upon its completion. Prior to entering this, we shall first discuss the manner one could conceive a cognitive agent operating in a physical habitat with an underlying Jason architecture.
Regardless of the actual physical construction of an agent, at its base level any cognitive agent must be capable interpreting and acting within a habitat. Interpretation of sensor data will result in agent perceptions; such sensors may be inertial, tactile, audio or computer-vision based. In addition to locally generated perceptions, an agent may use communicative acts to enquire about pertinent facts to which it is unable to determine, or indeed to verify locally generated information. These perceptions and communicative acts may create events that the agent must respond to, by selection of an applicable plan extracted from a set of behavioural rules and filtered through to a set of intentions by the AgentSpeak(L) mechanisms. Intentions for a physical agent correspond directly to interaction with physical hardware or further communicative acts. Any actuator device, locomotive or otherwise, will have a direct consequence within the habitat of the agent, altering its state and necessitating further interpretation of sensor data with subsequent agent action.

An informal treatment of the agent cycle, both within AgentSpeak(L) and for a generic cognitive agent using Jason, has been given. The consideration of a generic physical agent operating within a real habitat (not a toy Java environment) presents issues with the ability of an agent to interact successfully with real hardware, in addition to the implicit requirement of logically correct operation. Within the development of an applied agent, the plan libraries must correspond directly to the agent capabilities that are themselves determined by the particular agents hardware configuration. Here we are directly linking the agent to a particular high level command that initiates a particular control routine or device (de)activation.

The agent itself is not necessarily concerned with the inner workings of a control routine, but more with the existence and performance of a particular item within its repertoire of skills. It is here where there is a gap between the development of the agent's 'mind' and the agent skills, with a computer scientist being involved in the former and a control engineer the latter. This is likely to result in a partitioned system rather than one that should theoretically be a single coherent entity: by using sEnglish it is possible to concurrently develop both the agent 'mind' and the agent skills to result in a unified agent development process. Use of sEnglish with Jason will be extensively discussed in later sections, first we provide a more precise, formal definition of the language of Jason. This is needed as the Jason syntax and semantics is kept despite the use of sEnglish to link it with actions in the real bitat.



Standard syntax of Jason Code

The complete *Backus-Naur Form (BNF)* grammar for writing a Jason agent is provided by the original authors of Jason as in the table below. As with the presentation in the original documentation, <ATOM> is an identifier starting with a lowercase letter, <VAR> is an identifier starting with an uppercase letter, <NUMBER> is any integer or floating point number and <STRING> is any string enclosed within double quotes. " (...)* " means possible finite number of repetition of the format . Parts enclosed in [ ] are optional but ( ) is used for logical grouping of components where | stands for "or" . No "&" is used in these definitions except as part of a logical expression, and simple white space " " is used to express that *language items* on either side are needed:

Language Item	Definition of Language Item
agent	(initial_beliefs | initial_goals)* plans
initial_beliefs	beliefs rules
beliefs	(literal " . ")*
initial_goals	(" ! " literal " . ")*
literal	["~ "] atomic_formula
atomic_formula	(<ATOM> | <VAR>) [" (" list_of_terms ") "] [" [" list_of_terms "] "]

list_of_terms	term (" , " term)*
term	literal | list | arithmetic_expression | <VAR> | <STRING>
list	" [" [term (" , " term) [" | " (list | <VAR>)]] "] "
rules	(literal " :- " logical_expression " . ")*
plans	(plan)*
plan	[" @ " atomic_formula] trigger_event [" : " context] [" <- " body] " . "
trigger_event	(" + " | " - ") [" ! " | " ?"] literal
context	logical_expression | "true"
logical_expression	simple_logical_expression | " not " logical expression | logical_expression " & " logical_expression | logical_expression " | " logical_expression | " (" logical_expression ") "
simple_logical_expression	(literal | relational_expression | <VAR>)
body	body_formula (" body_formula")* | " true "
body_formula	(" ! " | " ? " | " + " | " - " | " -+ ") literal | atomic_formula | <VAR> | relational_expression
relational_expression	relational_term (" < " | " > " | " <=" | " >= " | " == " | " \\== " | " = ") relational_term
relational_term	(literal | arithmetic_expression)
arithmetic_expression	arithmetic_term [("+ " | " - ") arithmetic_expression]
arithmetic_term	arithmetic_factor [(" * " | " / " | "div" | "mod") arithmetic_term]
arithmetic_factor	arithmetic_simple [" ** " arithmetic_factor]
arithmetic_simple	<NUMBER> | <VAR> | "- " arithmetic_simple | " (" arithmetic_expression ") "

Fig. 2 The formal grammar definitions of Jason. Note the use of literals and atomic formulae that permit the use of variables for arguments as well as in place of function and predicate names.

Though this is easy to read, it is a bit criptic for the enjoyment of a learner. For sake of helping a learner of Jason to absorb this dry list, a verbal interpretation is provided here. The first item of this syntactic declaration states that a Jason agent consist simply from a list of *initial goals, initial beliefs* and *plans*. There is nothing more to the agent as it is abstracted away from sensing, world modelling, prediction and skills execution. At the highest representation level, it is only a se of goals and plans.
The *initial beliefs* part of the agent can be split into two parts: *beliefs* and *rules* . Now beliefs are simply a sequence of literals but literals are not that simple in Jason. In the bove definition, one can trace them down to consist of an *atomic formula* that is possibly *strongly negated* by ~ . However, atomic formulae can be quite complex here, as they can be started by either an *atom* (i.e. <ATOM>) or a *variable* (i.e. <VAR>) . This is optionally followed by a *list of terms* in rounded ( ) brackets and a list of annotations in square [ ] brackets. A list of terms is a comma separated list of *terms* . A term can be either of a literal, *list, arithmetic expression,* variable or *string (i.e.* <STRING>) . Note that this means the literals can have nested literals by these syntax rules.

The formulation of a Jason agent follows the logical format described above to create a set of initial conditions, plans and prolog-like rules. Whilst events for handling plan *failures* are available within Jason, resulting in the generation of an internal event of -!g if +!g failed, a plan matching the triggering event of -!g must be provided by the programmer.

Some example code fragments will now be given to indicate the construction of Jason plans based upon the given syntax. Let us consider an autonomous rover that is equipped with a visual odemetry

system and ultrasonic sensors that are used for navigation within an unmapped environment. During movement it is entirely possible that one or more of these sensor may pick up an obstruction that the rover must avoid; this is a reaction to a percept generated by one of the onboard measurement systems. Upon generation of this percept there is a series of events that should take place in order to deal with the detection and may consist of: stopping current motion, identifying exact location and type of hazard, adding this hazard to a self generated map, planning a path around the hazard, negotiating the hazard and continuing with the original mission. All of these tasks represent the body of the plan and are triggered upon hazard perception and satisfaction of particular beliefs: for instance the plan may not be valid if the rover is not moving when the hazard is perceived. Based on the above situation, a plan written in the Jason syntax would be:

```
@hazard_detection(during_motion)
+new_hazard(X,Y)[sensorType,detection_time]:rover(moving)& trusted(sensorType)
<- !stop(motion);
!identify(new_hazard,X,Y);
?new_hazard_properties(TYPE,SIZE,PLACEMENT);
!update_hazard_map(TYPE,SIZE,PLACEMENT);
!negotiate(hazard_map).
```

The above plan reacts to a hazard at location (X,Y) detected by a specific sensor at a noted time. The plan is only triggered if the rover is in motion and the sensor is trusted to be working correctly. Once the rover is stopped it is tasked with identifying the new hazard, which is assumed to result in the generation of a new item in the belief base relating the the properties of the new hazard such as the type, its size and the span of the hazard. This belief base item is queried, binding the type, size and placement for use in updating the self generated map which is then used to move safely through all the currently cataloged hazards. Each of the achieve items (those preceded by a "!") must have a corresponding set of plans, which may or may not have a set of guard conditions; for instance the subgoal of !stop(motion) may have the following associated plan:

```
+!stop(motion) : true <-
stopRover;
-rover(moving).
```

which is the same as:

```
+!stop(motion) <-
stopRover;
-rover(moving).
```

where upon detection of the achieve goal stop(motion) the internal task stopRover is run and the belief base is updated to account for the rover no longer moving. The implicit requirement here is that the agent is capable of executing specific items of code to control hardware that achieve the result of

stopping the rover, thus making the internally generated percept (internally generated percepts may be considered as mental notes of the agent) valid.

It is possible to build more complex plans based upon further conditioning within the plan itself. A trivial example of this would be:

```
+!move_to_item : direction_of_item(X) <-
    if (X==1){
    .println("Item is infront of me...");
    moveForward;
    };
    else{
    .println("Item is not infront of me...");
    moveBackward;
    }.
```

Whilst the same result could have been accomplished more succinctly without the *if-else* syntax, by using an internal action such as moveTowards(X), it was intended to show the ability to insert conditionality into a Jason plan. Further and more complete examples of Jason agent plans, including initializing agent beliefs, are available within literature and the interested reader is directed towards these sources. The intention here was to introduce the grammar for a Jason agent and to give some simple examples of formulating plans for conditional execution.
One final option within the creation of an agent goal, is creating a goal to which an agent is blindly committed to and is essentially a recursive action on the goal itself. An example for an overly social agent would be one which continually checks its email and could be written as:

```
+!checkEmail: true <-
.println("Checking inbox...");
updateInbox;
!checkEmail.
```

where the internal action of `updateInbox` may result in the generation of an email(pending) percept to which the agent would react.

Jason with sEnglish

As described in the section ”Definitions of Agent Reasoning” , the implementation of Jason with sEnglish is a hybridization of the two languages, generating a series of plan bodies with triggering events and guard conditions or contexts all expressed in natural language. In a *Jason*$^{+}$*sE* code, a term used to denote the sEnglish sentences based Jason code, the formal layout of Jason is retained, maintaining the original construction and connectives.

In *Jason*$^{+}$*sE* all components such as atomic formulae use a natural language presentation to increase readability of the resulting agent. The objective of this section is to provide an sEnglish relevant introduction into how Jason may be used in combination with Jason to produce an agent that is intrinsically linked into a repertoire of skills and where the cognitive processes are seamlessly integrated into the logic of the agent. It is assumed here that the reader is familiar with the use of the sEditor to develop sEnglish code that may be compiled through to executable code that is detailed within the *Getting Started* section of this guide: the development of agent specific code is completed in exactly the same manner, resulting in a hierarchy of code sentences that relate to specific actions. It is this collection of sEnglish sentences that represents the external actions that an agent may invoke. The difficulty now lies in the seamless integration of plans and sub-plans that lead to the achievement of a desired goal through the invocation of specific sEnglish developed code.

Firstly, let us consider a scenario involving the operation of a cognitive agent equipped with two robotic arms and a vision system. Under the assumption that the vision system is capable of identifying objects within the agent's habitat, as well as determining the current kinematic and dynamic state of both manipulators, we can assume the existence of sEnglish sentences relating to the operation of both the vision system and the manipulators such as:

- `Identify physical object X called 'plate' within habitat.`
- `Return Z as the kinematic state of left manipulator.`
- `Plan safe motion M of right manipulator to the location of object X.`
- `Grasp physical object X with right manipulator after using M.`

The above sentences are operationally understandable high level commands that, subject to the underlying code being completed, may enable execution of specific tasks relating to hardware interface or computational processes.

These code fragments, whilst themselves relating to complex operations that may or may not involve feedback processes, are not sufficient for operation of the complete system. Some form of 'intelligence' is also required to run the code fragments in a specific manner in order to achieve any form of output; this is the role of the agent reasoning described in Jason^{+}sE . In addition to the relatively basic operation of searching for a specific object and picking it up, the agent must also be capable of handling failures and unexpected events such as dropping the object.

A suitable agent may be one constructed that is blindly committed to the enactment of human requests, provided that relevant plans exist for the given request. Following from the example of an agent controlled dual manipulator robot, a possible command is to place a specific item into a particular box; for instance `"Place item 0231 into box Z1"`. This command is already in sEnglish format and to denote this we shall encapsulate it within curly braces to result `![Place item 0231 into box Z1.]` being picked up by the agent.

Being equipped with two separate manipulators, the agent may achieve the new goal by using either manipulator and in this case would result in conditional instantiations of `![Grasp physical object O231 with left manipulator.]` or `![Grasp physical object O231 with right manipulator.]`. Intuitively the guard conditions for such action would relate to the current status of each manipulator or proximity to the particular designated item, which itself is required to be a physical object within the agents habitat.

More powerful reasoning is possible through the implementation of Prolog-like facts and rules that exist within the agent's belief base and act to condition the agents permissible behaviour through more complex belief context checking. Examples relating to the considered agent system are:

```
If [Goal A was sent by authorized human.] then [A is a valid goal to be
achieved.] .
If [Right manipulator is not in use.] & [Right manipulator is preferred.]
then [Should use right manipulator.] .
If [Right manipulator is not in use.] & [Left manipulator is in use.] then
[Must use right manipulator.] .
[Right manipulator is preferred.] :- [Left manipulator is not in use.] &
[Right manipulator is closest to object.]
[Should not damage object V.] :- [V is a physical object in habitat W.] &
[Motion plan of N towards X has been generated.] & [Motion plan does not
conflict with location of V.]
```

The above behavioural rules will act to condition the agents' response by providing rules to determine validity of a command, the availability of a manipulator and the inference that a particular item will not be damaged during a specific motion. Although the usage of Prolog style rules within the development of a Jason agent enables more succinct code to be written relating to specific Jason plans, care must be taken when implementing such Prolog functionality. Within Prolog the order in which clauses are added to the belief base has an explicit effect on execution: clauses defined earlier within a program take precedence over those defined later. An additional concern is that Prolog is capable of expressing non-terminating algorithms. Both of these issues may present themselves within an incorrectly configured Jason agent, resulting in differing agent action depending on rule precedence and the possibility of a non-terminating belief context check.

Returning to the case in which an agent receives the high level achieve goal of `![Place item 0231 into box Z1.]`, a conceivable set of plans that may satisfy the execution of the prescribed goal are:

```
+![Achieve goal A.] : [A is a permitted goal.] <-
    ![Execute actions for goal A.].
+![Execute actions for goal A.] : ([Should use right manipulator.] | [Must
use right manipulator.]) & ~[Goal A has been achieved.]<-
    [Identify physical object within habitat.];
    [Plan safe motion M1 of right manipulator to accomplish goal A.];
    ![Execute motion plan M1 using right manipulator.].
+![Execute motion plan M1 using right manipulator.] : [Should not damage
target object.] & [Should not damage left manipulator.]<-
    [Use right manipulator to perform motion plan M1.];
    -[Right manipulator is not in use.].
+[Goal A has been achieved.]<-true
    +[Right manipulator is not in use.].
```

During the execution of the plans listed above, there are specific cognitive abstractions that will be generated upon completion of specific sEnglish sentence code, and as they appear in the belief base, they can be used for further conditional action(s). For instance `"Identify physical object within habitat."` may result in the generation of the sEnglish abstraction of `"V is a physical object in habitat W."` and `"Use right manipulator to perform motion plan M1."` the eventual generation of abstraction `"Goal A has been achieved."`.

Possible sEnglish Invocations

The use of sEnglish sentences within the body of a Jason plan is intended to unify concepts used with both the development of the agent and the code it may implement. The ontology used for the development of sEnglish sentences within the sEditor is the same as that used for the agent physical executor, thus endowing the agent with knowledge of all concepts used within the application area. The sEnglish implementation allows the full repertoire of Jason notations, including:

1. Composition of initial beliefs +[sEnglish sentence relating to belief base item.].
2. Composition of Prolog-like rules [sEnglish description.] :- [sEnglish description.].
3. Full use of logical connectives for context checking and rule formalizations.
4. Internal mental notes +/-^[sEnglish code relating to BB item.].
5. Plan context checking [sEnglish boolean test].
6. Action execution [sEnglish code reference.].

Sentences within the curly braces compile into atomic formulae to complete the declarative code that composes the Jason agent. Whilst items 1-5 are standard Jason functionalities, plan context checking and action execution sentences have options available.

Belief Base Updating

Belief base updating may occur as a result of a perception action made upon the habitat or as a result of an sEnglish code invocation. Habitat perception is enabled through the sEnglish statement [Monitor habitat for percepts.] being placed at the head of the plan list and will result in any MATLAB formed perceptions being transferred into the agent's belief base. Implementing this form of habitat monitoring obviously requires the relevant MATLAB based companion files or Simulink models to be formed appropriately: this is detailed within the Sections 'Interface with Simulink' and 'Interface with MATLAB'. When sEnglish code is run as part of a plan body, it may cause items to be inserted into the belief base of the agent and these may be subsequently used for latter plan steps or reasoned over by the agent.

Mental Notes

Development specific mental notes, as introduced by a programmer internal to the agent mind and not as a result of external belief base updating or through querying a specific habitat value, may be generated in two ways:

sEnglish Notation

If the user wishes to retain usage of the sEnglish notation, then a text note may be placed within curly braces that is preceeded by the 'hat' notation. For example, `^[Doing certain`

`task.]`, will generate the mental note `doing_certain_task` within the belief base and if desired this may be used to reason over at later stages when checking a plan context.

JASON Notation

The user may chose to retain the standard JASON notation and not use mental notes formed with sEnglish statements. To do this, simply state the predicate that is to be inserted such as functor($fact_1$,$fact_2$,...$fact_n$). A predicate does not necessarily require a set of facts to be associated with the functor, though it may be useful in some cases.

Plan Context Checking Through sEnglish

When context checking for relevant plans, it is possible to use a combination of belief base item checks and/or sEnglish based queries made upon the habitat. Such sEnglish based queries, for instance `[Goal A has been achieved.]`, must evaluate to a Boolean value. It is these Boolean evaluations, in combination with any negations that may be present, that are used to context check the current plan for relevance.

The sEnglish code denoting a Boolean test may be negated, `~[Goal A has been achieved.]`, or be present without negation. It may not be preceded with the keyword 'not', since this is used to denote default negation wherein it does not necessarily require the existence of a related belief base item. With implementation of sEnglish code based context checking, it implies a requirement of the belief item to exist and so only strong negation via '~' preceding the sEnglish sentence may be used. If the agent detects a false ~[sE Boolean test] (the negation of the sE Boolean test is not true) or false [sE Boolean test] (the sE Boolean test is not true), then the current plan is deemed not to be relevant and the agent considers other plans relating to the triggering event. Once all sE Boolean tests have been completed, the context is then checked against the current belief base for final confirmation of plan relevance: if it passes this step, then the plan is considered to be applicable.

The usage of mental notes has already been introduced and these may be used within context checking by the agent, through the same notation as used when introducing the mental note. For example, assuming the presence of two mental notes, introduced by `^[Mental note one.]` and `^[Mental note two.]`, then the following context check may be applied:

```
[event] : ^[Mental note one.] & ^[Mental note two.]  <- [plan body].
```

Equally, the full range of nested connectives may be applied to case of Boolean tests, sE derrived mental note, JASON mental notes and existing predicates within the belief base.

```
[event]: ~[Goal A has been achieved.] & (^[Mental note one.] | ^[Mental
note two.]  ) & belief_base_item(X,Y) <- [plan body].
```

Action Execution Through sEnglish

Action execution itself may relate to the tasks of:

- Initializing a piece of hardware or a particular control routine
- Starting a computational process with or without a parameter and retrieving the result
- Querying a piece of hardware or a computational process for a value

The above tasks directly relate to interaction with components that are external to the processes of the Jason agent, but will have direct impact upon its operation. Relating this to the vision assisted manipulator robot, the vision system must be initialized and sent commands relating to identification of particular items whilst also being linked into separate routines such as path planning and motion control. Such Jason initiated external interaction is of two forms: those that require a response, and

those that do not. The former relates to sentences indicating the need to update the Jason belief base with a specific knowledge item: responses are in the form of atomic formulae being added to the belief base with the predicate named appropriately for the particular call and the predicate terms holding required information for subsequent reasoning. For instance "Determine red items in habitat." indicates the requirement of a new belief base item such as +[Number of red items in habitat is X.], where X is bound to a value, being generated for the agent to use with subsequent reasoning. External interaction that does not require a response for direct reasoning relates to the initializations of processes and hardware, for instance "Initialize vision system." or "Move to position Y." do not necessitate any direct responses, although their invocation may result in the eventual appearance of believe base items such as +[Vision system is active.] or +[Reached position Y.].

Tasks that an agent may execute fall into two categories: those that require continuous action in the habitat and those that involve discrete actions. Discrete actions may involve the initialisation of a particular hardware item and its subsequent command/query interaction or the performance of a particluar numerical computation; continuous actions may be those where a particular control loop is to be repeated until a discrete request to terminate this loop is made. The sentences that represent discrete actions to be executed once are compiled into Jason external calls of the form

```
invoke(<executive_process_name>,runOnce,<mfile_name>,[<input_list>'],[<outpu
t list>])
```

A stipulation here is that only discrete requests are capable of returning an argument back to the Jason system: continuous actions do not return a predicate to the Jason agent, however the action of a continuous action may result in the change of a monitored percept or a discrete action may be invoked to query a specific item that the continuous action is involved with. The other two types of invoke commands are

```
invoke(<executive_process_name>,runRepeated,<mfile_name>,[<input_list>'],[<o
utput list>])
```
and
```
invoke(<executive_process_name>,stopRepeated,<mfile_name>,[<input_list>'],[<
output list>])
```

The runRepeated differs from runOnce in that it does not communicate back outputs to the Jason process. If the associated sentence has an output then that will be sent back to the Jason process when stopRepeated is applied. Whether runOnce, runRepeated or stopRepeated is used is determined by the sEnglish ro Jason compiler from the meaning of the sentence. If the meaning of the sentence contains Repeat this activity until stopped. then the type of the invoke command becomes runRepeated . To stop a repeated process by another sentence, its meaning must contain

```
This is a description of stopping this activity.
```

in its .sep file definition . If these sentences do not occur in the meaning then the invoke statement in Jason becomes of type runOnce. The meaning definition of all sEnglish sentences must contain a

reference to the process that it is invoked by. This may be achieved wither by including an additional sentence tag "`This sentence is interpreted by executive process <proc_name>`."

within the sEnglish code description of the sep file, or by the more concise method of completing the sep file field "process, repeat mode :: " to indicate to the Jason agent which executive process to communicate with to run the activity of the sentence.

There are some notational nuances with the execution of differing tasks within the habitat, depending upon the desired frequency of the action item that may be executed once or repeatedly until the agent requests the action to be stopped. The default action is a single instance: any sE sentences that are placed within the plan body will only be processed once, unless certain key words are used within the sentence formulation.

One final point is that relating to the system a particular agent is connected and the degree of partitioning of the (non-agent) system architecture. If one is to assume that the agent in our robot manipulator example is capable of running multiple processes reacting to the operation of each distinct sub-system, ie the manipulator controllers and vision system are separate processes, then each system is capable of being interacted with through distinct calls. This minimizes the burden to the process dealing with perception updates and monitoring for desired agent actions. In some instances it may not be possible to configure a system such that all operational capabilities are spread across multiple systems and in this case will necessitate the process responsible for perception update and agent code invocation also being responsible for the invocation of each code item using its own resources. Whilst this is possible, execution of feedback controllers and other such processes, will severely constrain the system and if possible single process implementations should be avoided.

Another example

Fig. 3.1 displays the content of an example `rover.sej` file of a complete sEnglish definition of Jason based reasoning. It is based on the use of sEnglish sentence definition files :

```
battery_level_is_critical.sep,
move_through_environment_towards_position.sep,
new_position_request_arrived.sep, not_being_at_position.sep,
position_is_feasible_to_reach.sep, position_request_arrived.sep,
rover_is_faulty.sep, rover_is_visible.sep, stop_rover.sep.
```

The `.asl` file compiled from the `.sej` file is shown in Fig. 2 .

Initial belief statements

The first part of this sEj code (for sEnglish/Jason code) is about initial beliefs expressed in terms of pure Booleans value sentences that have no .sep file or sentence defined for them in the sEnglish document. These are mental notes. For instance `doing_a_mission` is an example that is an atom in Jason and its *true* if `doing_a_mission` is present in the belief base and *false* means that `~doing_a_mission` is present in the belief base.

```
INITIAL BELIEFS
~Doing a mission.

INITIAL ACTIONS
Initialize hardware system.

PERCEPTION PROCESSES
Monitor the following Booleans :
Rover is visible.
Battery level is critical.
New position request arrived.
Rover is faulty.

Monitor the following objects :
Request has arrived to go to position Goal.

REASONING
^[There is a reasonable request.] :- ^[Rover is visible.] & ~^[Battery level
is critical.]
& ^[Requested is different from current position.]
& ~^[Doing a mission.] & ~^[Rover is faulty.].

EXECUTABLE PLANS
+^[New position request arrived.] : [Not being at position Goal.]
<- +^[Requested is different from current position.].
+^[There is a reasonable request.] : [Position Goal is feasible to reach.]
<- +^[Doing a mission.] -~^[Doing a mission.].
+^[There is a reasonable request.] : ~[Position Goal is feasible to reach.]
<- .println("Cannot reach the requested goal.").
+^[Doing a mission.] : [Not being at position Goal.]
<- [Move through environment Map towards position Goal.]
+^[Doing a mission.] .
+^[Doing a mission.] : ~[Not being at position Goal.]
<- +^[Has arrived to target location.].
+^[Has arrived to target location.]. <- [Stop rover.] +~^[Doing a mission.].
```

Fig. 3.1 Content of the rover.sej agent logic definition file that follows the Jason syntax and is compiled into Jason.

Definition of perception Processes

The first part of perception processes must be headed by "`Monitor the following Booleans`:" and followed by sentences one per line. Each of these sentences must be defined with a single relation Boolean output in the sEditor and with no inputs. Actual inputs can of course be sensing devices that are not used in the .sej code. The meaning definition of each of these sentences must contain a sentence "`This sentence is interpreted by executive process <process name> .`", for instance "`This sentence is interpreted by executive process agv421exe .`" or the `Process, repeat mode :` : field of the sentence definition .sep file must declare `<process name> .`

The second part of perception processes must be headed by "`Monitor the following objects`:" and followed by sentences one per line. Each of these sentences are used to update the values of certain objects for the agent. For instance the "`Request has arrived to go to position Goal.`" regularly updates the value of the `Goal` object in the associated executive process. This

object is not however instantiated in the Jason/Java environment, it is only referred to symbolically as "`Goal`" by a string in Jason.

```
// INITIAL GOAL AND BELIEFS
!checkWorld.
~doing_a_mission.
// INITIAL ACTIONS
initialsing_hardwar_system.
// PERCEPTION PROCESSES
// Note: Perception Boolean tag names have to be identical to their sentence in lower
case.
@observe+!checkWorld <-
configureBoolean(agv421,rover_is_visible);
configureBoolean(agv421,battery_level_is_critical);
configureBoolean(agv421,new_position_request_arrived);
configureBoolean(agv421,rover_is_faulty);
configureObject(agv421,position_request_arrived,[],["Goal"]);
!checkWorld .
// REASONING BY RULES
there_is_a_reasonable_request :- rover_is_visible & ~battery_level_is_critical
& requested_is_different_from_current_position
& ~doing_a_mission & ~rover_is_faulty .
// EXECUTABLE PLANS AND TESTS
+new_position_request_arrived : testBoolean(agv421,not_being_at_position,["Goal"])
<- +requested_is_different_from_current_position .
+there_is_a_reasonable_request                                                    :
testBoolean(agv421,position_is_feasible_to_reach,["Goal"])
<- +doing_a_mission; -~doing_a_mission .
+there_is_a_reasonable_request                                                    :
~testBoolean(agv421,position_is_feasible_to_reach,["Goal"])
<- .println("Cannot reach the requested goal.") .
+doing_a_mission; : testBoolean(agv421,not_being_at_position,["Goal"])
<-
invoke(agv421,runOnce,move_through_environment_towards_position,["Goal","Map"],[]);
+doing_a_mission .
+doing_a_mission; : ~testBoolean(agv421,not_being_at_position,["Goal"])
<- +has_arrived_to_target_location .
+has_arrived_to_target_location.     <-     invoke(agv421,runOnce,stop_rover,[],[]);
+~doing_a_mission .
```

Fig. 3.2. The contents of the .asl file compiled from the .sej file in Fig. 1 .

Summary of sEnglish to Jason conversion rules

The following table lists an extensive set of example to illustrate when simple predicates are used and, when the `invoke()`, `configureBoolean()`, `configureObject()`, `testBoolean()`

Location in Jason+ code	Example	Jason predicate

`INITIAL BELIEFS`	`~Arrived at position P0.`
	
	`Mission is pending.`
		`~arrived_at_position("P0") .`
		
		`mission_is_pending .`

INITIAL ACTIONS	Initialize system.
	
	Read timed circle manoeuvre Tc from file 'c:\tcm1.txt'.
	
	Get starting point P0 from Tc.
		invoke(ttsat01,runOnce,initializing_system,[],[]);
		
		invoke(ttsat01,runOnce,reading_timed_path_from_file,['c:\tcm1.txt'],["Tc","P0"]);
		
		invoke(ttsat01,runOnce,getting_starting_point_from_path,["Tc"],["P0"]);

PERCEPTION PROCESSES	Arrived at position P0.
	
	Completed first circle manoeuvre.
		configureBoolean(ttsat01,arrived_to_position("P0"));
		
		Returns predicate arrived_to_position("P0") .
		
		configureBoolean(ttsat01,completed_first_circle_manoeuvre);
		
		Returns predicate completed_first_circle_manoeuvre .

REASONING	If ^[Mission is pending.] & ~^[Mission is active.] then ^[Ready to move initial position P0.]. The fact ^[Arrived to position P0.] implies ~^[Mission is pending.] . The fact ^[Arrived to position "P0".] implies ~^[Mission is pending.] .	ready_to_move_initial_position("P0") :- mission_is_pending & ~mission_is_active . Temporarily places ready_to_move_initial_position("P0") to belief based for reasoning cycle. ~mission_is_pending :- arrived_to_position(P0) . Note that in the top example "P0" and the bottom example P0 was used for Jason+ by the compiled as the conclusion variable was not matched by a variable from the premises so it needs to be found by other means . In the second example we do not want the P0 to be concrete as we want to make the implication general. Below is an example to show how to make P0 appear in quotes for the Jason+ program. ~mission_is_pending :- arrived_to_position("P0") . Hence automatic conversion of variables to double quoted name is only done in conclusions if not resolved by premises.

EXECUTABLE PLANS	If ^[Ready for first circle manouevre.] while ^[Mission is active.] then [Announce that 'I am ready to go around'.] [Perform timed circle manoeuvre Tc.] [Stop where you are.].
	
	If ^[Ready to move initial position P0.] while ~^[Mission is active.] then [Announce that 'I am moving towards my initial position'.] [Move to position P0.] [Stop where you are.] +^[Mission is active.].
	
	If ^[Mission time is far exceeded.] while ^[Mission is active.] then [Announce 'Mission time has been far exceeded and therefore I am stopping'.] [Stop where you are.] [Close down all your operations.].
		+ready_for_first_circle_manouevre :
		mission_is_active <-
		invoke(ttsat02,runOnce,announcing_something,['I am ready to go around'],[]);
		invoke(ttsat01,runOnce,performing_timed_circle,["Tc"],[]);
		invoke(ttsat01,runOnce,waiting_for_buddy,[],[]) .
		
		Here the Tc has been converted to "Tc" as it is not resolvable by the triggering event.
		
		+ready_to_move_initial_position(P0) :
		mission_is_active <-
		invoke(ttsat02,runOnce,announcing_something,['I am moving towards my initial position'],[]);
		invoke(ttsat01,runOnce,moving_to_a_position,[P0],[]);
		invoke(ttsat01,runOnce,waiting_for_buddy,[],[]);
		+mission_is_active .
		
		Here the P0 has not been converted to "P0" as it is resolvable by the triggering event.
		
		+mission_time_is_far_exceeded :
		mission_is_active <-
		invoke(ttsat02,runOnce,announcing_something,['Mission time has been far exceeded and therefore I am stopping'],[]);
		invoke(ttsat01,runOnce,waiting_for_buddy,[],[]);
		invoke(ttsat01,runOnce,closing_down_all_operations,[],[]) .

Automatic initialisation of non-existing .sep files

Programming a new agent and its reasoning can be significantly speeded up by initiating all .sep files just by writing up a .sej reasoning file. This is achieved by writing after the period "." within the square brackets [...] of the sEnglish sentences used.

Text after the ". " within [...] can be used to define the name of and create a new .sep file automatically if no sentence meaning was found by the .sej to .asl compiler. In the perception part, where no [..] is used, but one sentence per line, the wanted tag for a .sep file name can be written after the ".", if the sentence did not exist before. If it did, that does not cause error, the text after the "." will be ignored. Such a system now allows to write first a .sej file and skeletons of .sep files are automatically generated for all sentences used, they are all placed into Section 1. Later they can be edited and relocated to other sections and tags after the "." removed. Note that after creation of the .sep file the section after the "." within [...] will be erased in order not to over-write already worked on files at the next compilation of the .sej file.

Bibliography

L Dennis, M Fisher, A Lisitsa, N Lincoln and S M Veres. Satellite Control Using Rational Agent Programming. *IEEE Intelligent System Magazine*, Vol 25, No. 3, p.7.

N K Lincoln and S M Veres. Natural language programming of complex deployable agent systems : a fusion of sEnglish and Jason. *Draft paper – submitted to IEEE Transactions on Robotic Systems.*

S.M.Veres. Natural language programming of agents and robotic devices: Publishing for machines and humans in sEnglish. *SysBrain Ltd, London, 2008*. ISBN 978-0-9558417-0-5.

S. M. Veres. Reference manual of the senglish syntax and semantics. *Technical report*, *SysBrain Ltd, Southampton,UK, 2011*.

S. M. Veres and N. K. Lincoln. Sliding mode control of autonomous spacecraft. (half written insEnglish). *Proc. 9th Conference Towards Autonomous Robotic Systems* (TAROS'08), September 2008, Edinburgh.

S. M. Veres and L. Molnar. Documents for intelligent agents in English. *Proc. IASTED Conference on Artificial Intelligence and Applications*, Feb 2010, Innsbruck, Austria.

S. M. Veres. Theoretical foundations of natural language programming and publishing for intelligent agents and robots. *Proc. 11th Conference Towards Autonomous Robotic Systems* (TAROS'10), 2010.