GOAL Architecture Recommendations

W.Pasman, 6 may 2008

Introduction

This document gives recommendations for the GOAL architecture. We start with a review of previous work: a short review of [Pasman08] and comments by students on the prototype GOAL architecture (unpublished). A brief overview is given of GUI/IDE tools that were encountered in [Pasman08]. We will give recommendations already during the overview, based on common practice and findings. Then we proceed with the GOAL recommendations.

1. Lessons from Existing Systems

1.1 Architecture

The platform will have to run on some middleware, supporting launching, moving and killing of agents, and communication between agents. We want to support various middleware. In order to abstract from the middleware, I recommend a lightweight middleware abstraction layer as for instance in Jason.

On most agent platforms there seems only a weak integration with the middleware platform (e.g. JADE). The JADE debugger is often on conflicting terms with the framework debugger and the framework may appear to hang if the JADE debugger is used to step an agent. Also de behaviour introspector of JADE does not show the actual internal steps of the agent. In fact JADE is used only to pass around messages and spawn/move agents. The JADE tools probably just are a debug feature for the framework programmers, and should not be used by agent programmers. I recommend to be very clear of the role of the middleware, and not leave agent programmers in the dark about which debug buttons should be used and which not. Furthermore I recommend to hide the middleware tools (sniffer, debugger) from direct access from the agent platform and make them only available when the user really knows what he is doing.

I recommend that sub-modules come as single .jar files, and using Java interface definitions. Other modules can then simply attached (e.g. using a preferences pane like in JASON). Eclipse (being used as central controller in Jason) also has a sophisticated plug-in system that might contain good ideas for modularization. Automatic Java compilation by runtime core: I recommend to *avoid* this. Special compile options, dependencies, security issues, inclusion of special files, all make this auto compilation troublesome to impossible for more complex agents.

Some platforms launch the environment in a thread that is invisible to the middleware system, or as a shared object within the Java Virtual Machine (**JVM**). As in general the environment may run on a different JVM than the agent, this either implies that some separate remote procedure calls are used to call the environment, or that the framework will break down when run with multiple JVMs. I recommend to use the messaging system of the middleware also to contact the environment(s). This should be not too much work as for instance Jason already has this and we can inspect their code to see how to do this. The only requirement is that there is some message communication means with the environment that has to be attached, but that seems required anyway as we want to run on top of middleware which requires message communication.

The most advanced debugging features are delivered by the JADEX platform. Agent actions can be inspected at the ProcessEvent level, instead of just the next action of the agent as in most platforms. On the architecture level, this is at the same level as the JADE behaviour inspector. In fact the debugger as we have it already in the GOAL system is more advanced than the agent debuggers we investigated [Pasman08].

None of the available architectures has a global debug framework, all the architectures are limited to debugging their own code, and stop when the boundaries of the module are reached. As modularization gets more detailed, these boundaries are reached still sooner, and debugging gets even more limited. I propose to develop a debugging framework that can cross these boundaries. This will be discussed in some more detail in the section 1.3 Across-module Debugger below.

In general it was hard to attach another JVM to the discussed frameworks. Maybe we should spend some more time to test this further.

Moving around agents between JVMs seems an important aspect, e.g. for supporting a mobile user, load balancing, maintenance, efficiency etc. For various reasons, it may be necessary to not move an agent or even to fix it to a certain piece of hardware. To my knowledge JADE can move every Java agent to another platform and is not considering such issues. I propose that we add some platform requirements and a movability flag to GOAL agents, so that we can at least attempt to handle this on the GOAL level.

To work this out in more detail, consider the usage of the (platform-limited) SWI prolog by GOAL agents. A simple approach would be to create a swi.jar module (this module implements the GOAL Database object) that is fixed to a single platform. In that case any GOAL agent using the swi.jar module would be fixed to that platform as well. An advanced approach would be a swi.jar module that has as requirement that swi prolog has been installed on the machine. In that case, an agent using the swi.jar module can be moved (even at runtime) to any platform with prolog installed. Note that to realize this, the Database objects have to be serialized, which will involve for instance extracting the database contents from SWI Prolog on machine A and loading them into SWI Prolog on machine B. A third approach could be to change the swi.jar module into a swi engine as a separate agent (not a GOAL but a JADE agent) and to route all SWI Prolog calls to that engine. In this case there probably would be severe performance hits to do so, but maybe in other cases it is desirable to be flexible in the choice whether a module (.jar file) runs as an independent java thread on a remote object, and to convert function calls automatically to messages. I think it is even possible (with only a small performance hit per call) to create a .jar module that can be run either independently, communicating via JADE messages, or to be linked hard against the other code that used the module. This would give an extremely flexible modularization. If the module is hard linked, I expect a performance very similar to hard linking.

An important choice is whether the IDE is (1) a platform manager with (a.o.) edit functionality, or (2) an editor with (a.o.) a run button. Approach (1) will typically launch a new window whenever an edit action is requested, approach (2) when the application is run. The second one seems the popular view at this moment and is the usual approach when Eclipse, jEdit and jExt are used as the basis for an IDE.

Resetting an agent was not properly investigated. I propose that we check this out on the various platforms in more detail. Resetting and debugging are closely related, for instance resetting a module high up in a debug chain at least will need unlocking the debug chain. This is complex, for example a network problem may be the reason to issue the reset in the first place, and debug plugins may still be functioning but unreachable.

In all, this gives a suggested architecture picture for GOAL as in Figure 1:

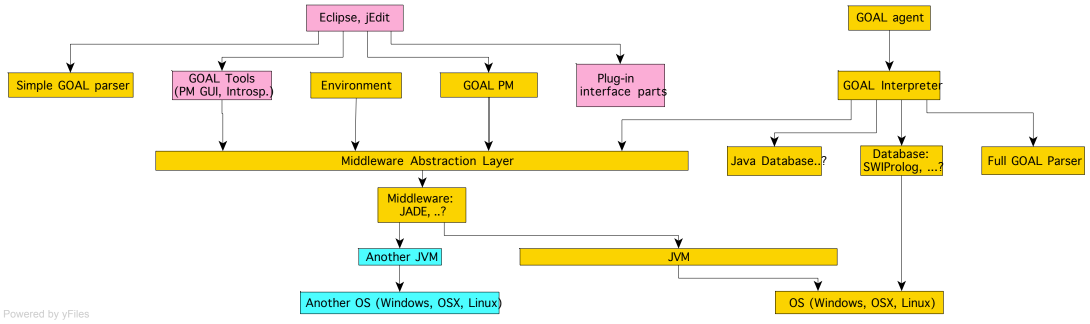


Figure 1. GOAL architecture. The blue part is an optional second JVM, the job of the middleware is to hide these 'hardware' details. Pink parts are parts involving mainly GUI.

For the moment I suggest the PM tool also collects the messages from the agents (maybe we can also reroute the stdout stream through the middleware abstraction layer?). In large systems this might become a bottleneck though. Note that in this case users have to start a PM tool to see errors.

Fatal errors have to be reported up front, not somewhere deep down like in Fleeble. For non-fatal errors this is discussable, but I recommend that reporting all errors centrally is most useful, if there are many errors there probably is something wrong anyway. I also recommend using the Warning system as we have it now in GOAL but extending it for multi-agent multi-platform. This will involve routing all these messages through the middleware layers.

Resetting is poorly supported on most agent platforms. On paper JADE has a feature to restart agents whenever they crash but strangely it seems not possible to force a restart. The common practice for IDE-oriented agent platforms is to boot the entire middleware and MAS whenever a run is requested, and to kill the entire middleware when the stop button is pressed. Only Fleeble allows restarting of individual agents at runtime.

Restarting GOAL agents has shown to be tricky: often agents have certain initial assumptions (for example, assuming that the initial time is 0) which may prove incorrect at reset time, causing an agent to malfunction after reset. Nevertheless, for larger systems, keeping the middleware running and only loading a sub-MAS, environment and/or individual agents seems necessary. I recommend to be able to keep the middleware running even when a MAS is taken down. The basic component for this is an architecture that enables resetting individual agents as well as the environment.

1.2 Student Comments on GOAL

This section reviews the comments of students on a first prototype of the GOAL system.

Many comments were due to the lack of a file menu in the prototype. Obviously this has to be corrected, with menu items to load a MAS file, agent file, and environment, and to restart it all at once or separately.

A number of comments were on issues with the reset functionality. The GOAL prototype has a separate reset for the environment and for the agents, leading to malfunctioning and crashing agents if this is done out of sync. Resetting is complex and needs discussion, even more than debugging.

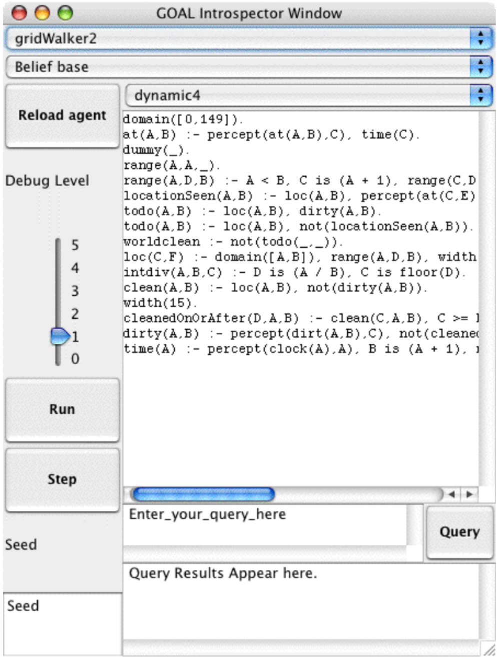
Some students requested a single GUI instead of several as it is now. In that case, functionality similar to the GOAL Tools would have to be copied inside Eclipse or jEdit. Duplicate work might be avoided if we can somehow separate the panels from the agent code. I would prefer to keep things separate though, also to increase awareness of the different components.

One student found it "strange" that the environment appears in a separate window. There apparently is a misconception that the environment is part of GOAL. Integrating the environment with GOAL components in a single panel will only reinforce this misconception. I recommend to keep the environment separate for now.

A few students requested better visibility of *changes* in the databases. We could make a "difference" view on the database, or create a separate "domain knowledge base" (that would keep growing, maybe not what the students would like to see?). I suggest determining the difference at runtime, based on a simple string comparison of the available list of facts. Some limit would have to be set to distinguish between what is old and what is new. A separate domain knowledge base does not sound like a good idea to me, for instance it does not allow retracting knowledge from the base set.

Two students requested a build-in editor for agents. That's exactly what the proposed editors in the architecture picture are for. I'm not sure whether a remote agent can always reach the edited source files, this will depend on the power of the middleware level.

One student asked this editor *as part of the introspector*. To me this seems an incorrect understanding of the introspector, which may be indicative of incorrect GUI design. This is one of the reasons that I propose separation of run-time and IDE windows.



```
domain([0,149]).
at(A,B) :- percept(at(A,B),C), time(C).
dummy(_).
range(A,A,_).
range(A,D,B) :- A < B, C is (A + 1), range(C,D
locationSeen(A,B) :- loc(A,B), percept(at(C,E)
todo(A,B) :- loc(A,B), dirty(A,B).
todo(A,B) :- loc(A,B), not(locationSeen(A,B)).
worldclean :- not(todo(_,_)).
loc(C,F) :- domain([A,B]), range(A,D,B), width
intdiv(A,B,C) :- D is (A / B), C is floor(D).
clean(A,B) :- loc(A,B), not(dirty(A,B)).
width(15).
cleanedOnOrAfter(D,A,B) :- clean(C,A,B), C >= 1
dirty(A,B) :- percept(dirt(A,B),C), not(cleane
time(A) :- percept(clock(A),A), B is (A + 1), 
```

Figure 2. Introspector window of the GOAL prototype system.

One student requests backward stepping of agents at runtime. This will require rewriting the entire GOAL interpreter and possibly the components that it relies on. One solution is to store the entire agent structure, including the stack and database contents, at each step that can be reversed. This functionality is called Serializing and we may want it anyway to enable moving an agent. I recommend leaving this for later.

Linux is urgently requested as supported platform. I propose that we move this up on the priority list.

Numerous students have problems with the installation. Some tried to install on non-standard configurations and even non-supported configurations. We need to improve the installation process.

1.3 Across-module Debugger

As discussed, a debug structure is needed that can cross module boundaries. The debugger is an important of the GOAL interpreter, and for the set-up of the entire architecture. A debugger interface for a module needs two ports/message streams. The first port informs outsiders about the latest tracepoint that was reached (I propose a List of Strings) and whether the module is halted at that point or not. The second port is in the other direction, and allows outsiders to request tracepoint levels and to change the run mode (run mode being one-step, continuous, halt and maybe something to handle reset). The List of Strings seems fit both to stack-based architectures (eg Java), backtracking architectures (eg Prolog) and query systems (eg SQL server). If a call is done to other modules, the debugger interfaces of the involved modules should be connectable to ensure continuous debugging, without intervention of the higher modules that are using the debugger interface. This structure should work with direct function calls crossing modules, but maybe also for remote calls working via messages (for instance when a remote prolog server is being used). More research is needed to work this out in detail. The current GOAL system already does across-module debugging, but in a bit ad-hoc manner.

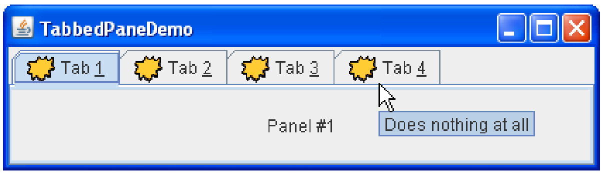
2. GUI / IDE Tools

In [Pasman08] a number of GUI related modules were discussed. This section investigates the possible use of them. We start with three basic GUI tools and then review a few more advanced editors / IDEs.

2.1 Basic GUI tools

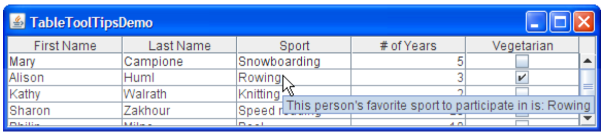
AWT is the standard GUI toolbox for Java (http://java.sun.com/developer/onlineTraining/awt/contents.html). It has a few layout managers, plus the components Button, Canvas, CheckBox, Choice, Label, List, Menu, Scrollbar, TextField, and TextArea . AWT is good enough for most GUIs but inconvenient when tabs, tables and split panes, file dialogs etc are needed.

Swing is a more advanced Java toolbox (http://java.sun.com/docs/books/tutorial/uiswing/components/menu.html) to create GUIs. It comes standard with Java as well and creates extra functionality over the standard AWT. Some mobile devices may not support Swing. It adds Combo Boxes, Color Choosers, File Chooser, Editor Panes, Dialogs, Internal Frames (Windows in Windows), Layered Panes, multi-column Lists, Password fields, Progress Bars, Tool tips (pop-up text being shown when hovering over a component), Trees (Figure 5), Sliders, Spinners (combo boxes but for integer values and no pop-up), Split panes (Figure 6), Tabbed Panes (Figure 3), and Tables (Figure 4). Swing is sufficient for almost all GUI designs.



Does nothing at all

Figure 3. Tabbed Pane



First Name	Last Name	Sport	# of Years	Vegetarian
Mary	Campione	Snowboarding	5	□
Alison	Huml	Rowing	3	☑
Kathy	Walrath	Knitting		
Sharon	Zakhour	Speed r		

This person's favorite sport to participate in is: Rowing

Figure 4. Table.

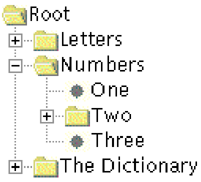


Figure 5. Example Swing Tree.

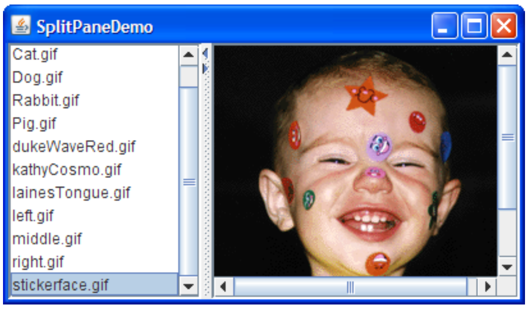
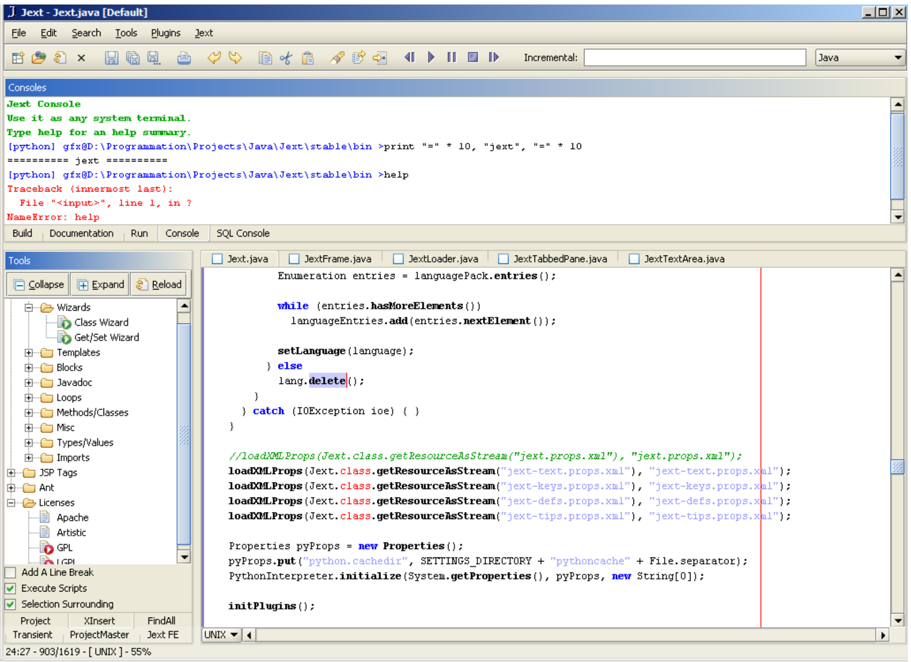


Figure 6. Split pane example. The split is the vertical grey dotted bar in the center, that can be dragged (sideways in this case).

SWT seems mainly an alternative for Swing, and its main use is in Eclipse. It offsers a slightly different visualization of very similar components and has slightly less functionality than Swing (e.g. limited Table support). SWT is sometimes reported to be slightly faster, but the difference seems marginal [Yucel04, Kiewe06]. In the past, Swing seems to have been supported better than SWT [Ditchendorf06]. Especially as platform independence is relevant, SWT might better be avoided.

2.2 Editors/IDEs

jExt [jExt05] is a syntax-directed editor/IDE developed in 2000 (Figure 7). It had been become popular since 2003 with the peak in 2005 with nearly 15000 downloads per month. The control structure, pop-ups, actions, and even Java actions controlling jExt are all placed in xml files. The latest release is from july 2004 so this seems to get a bit old.

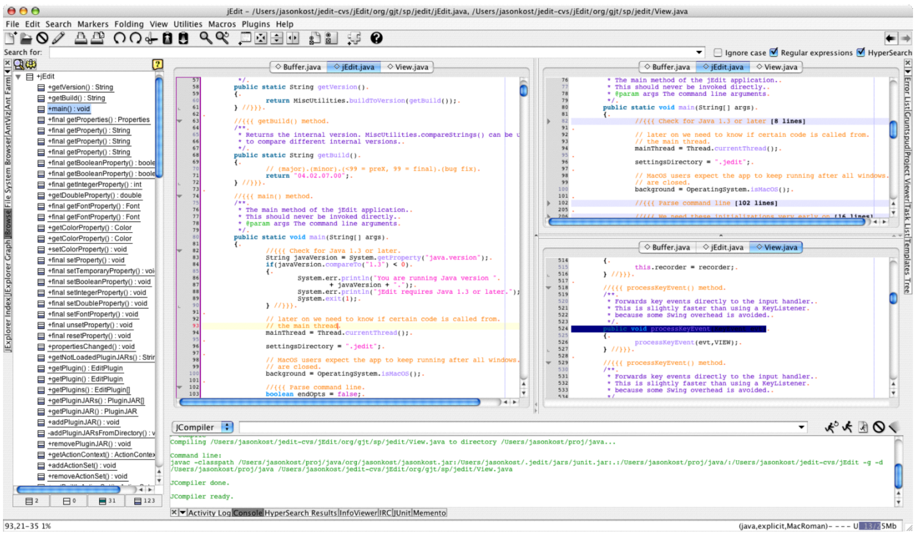


```
Jext Console
Use it as any system terminal.
Type help for an help summary.
[python] gfx@D:\Programmation\Projects\Java\Jext\stable\bin >print "=" * 10, "jext", "=" * 10
========== jext ==========
[python] gfx@D:\Programmation\Projects\Java\Jext\stable\bin >help
Traceback (innermost last):
  File "<input>", line 1, in ?
NameError: help
```

```
        Enumeration entries = languagePack.entries();

        while (entries.hasMoreElements())
          languageEntries.add(entries.nextElement());

        setLanguage(language);
      } else
        lang.delete();
    }
  } catch (IOException ioe) { }
}

//loadXMLProps(Jext.class.getResourceAsStream("jext.props.xml"), "jext.props.xml");
loadXMLProps(Jext.class.getResourceAsStream("jext-text.props.xml"), "jext-text.props.xml");
loadXMLProps(Jext.class.getResourceAsStream("jext-keys.props.xml"), "jext-keys.props.xml");
loadXMLProps(Jext.class.getResourceAsStream("jext-defs.props.xml"), "jext-defs.props.xml");
loadXMLProps(Jext.class.getResourceAsStream("jext-tips.props.xml"), "jext-tips.props.xml");

Properties pyProps = new Properties();
pyProps.put("python.cachedir", SETTINGS_DIRECTORY + "pythoncache" + File.separator);
PythonInterpreter.initialize(System.getProperties(), pyProps, new String[0]);

initPlugins();
```

Figure 7. jExt editing a Java file.

jEdit (Figure 8) was created in 1999 and its popularity peaked in 2006 with about 60000 downloads per month. It also uses xml files to control the editor but it does not include Java in the XML and DTD files and there are much less xml files than in jExt. There is still project activity, with for instance about 80 issues closed and 93 new reported in march 2008. It looks pretty similar to jExt, only the button looks could use some modernization.


jEdit - /Users/jasonkost/jedit-cvs/jEdit/org/gjt/sp/jedit/jEdit.java, /Users/jasonkost/jedit-cvs/jEdit/org/gjt/sp/jedit/View.java
File Edit Search Markers Folding View Utilities Macros Plugins Help
Search for:
☐ Ignore case ☑ Regular expressions ☑ HyperSearch
+jEdit
+getVersion() : String
+getBuild() : String
+main() : void
Buffer.java jEdit.java View.java
Error List Gruntspud Project Viewer Task List Templates Tree
JCompiler
Activity Log Console HyperSearch Results InfoViewer IRC JUnit Memento
93,21-35 1%
(java,explicit,MacRoman)- - - - U 13/25Mb

```
        */
        public static String getVersion()
        {
                return MiscUtilities.buildToVersion(getBuild());
        } //}}}

        //{{{ getBuild() method
        /**
         * Returns the internal version. MiscUtilities.compareStrings() can be u
         * to compare different internal versions.
         */
        public static String getBuild()
        {
                // (major).(minor).(<99 = preX, 99 = final).(bug fix)
                return "04.02.07.00";
        } //}}}

        //{{{ main() method
        /**
         * The main method of the jEdit application.
         * This should never be invoked directly.
         * @param args The command line arguments
         */
        public static void main(String[] args)
        {
                //{{{ Check for Java 1.3 or later
                String javaVersion = System.getProperty("java.version");
                if(javaVersion.compareTo("1.3") < 0)
                {
                        System.err.println("You are running Java version "
                                + javaVersion + ".");
                        System.err.println("jEdit requires Java 1.3 or later.");
                        System.exit(1);
                } //}}}

                // later on we need to know if certain code is called from
                // the main thread
                mainThread = Thread.currentThread();

                settingsDirectory = ".jedit";

                // MacOS users expect the app to keep running after all windows
                // are closed
                background = OperatingSystem.isMacOS();

                //{{{ Parse command line
                boolean endOpts = false;
```

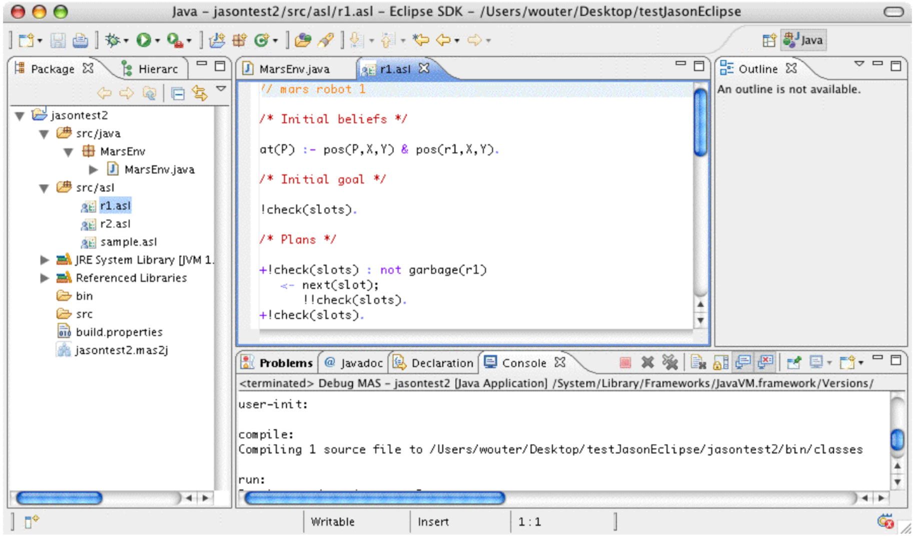
```
         * The main method of the jEdit application.
         * This should never be invoked directly.
         * @param args The command line arguments
         */
        public static void main(String[] args)
        {
                //{{{ Check for Java 1.3 or later [8 lines]

                // later on we need to know if certain code is called from
                // the main thread
                mainThread = Thread.currentThread();

                settingsDirectory = ".jedit";

                // MacOS users expect the app to keep running after all windows
                // are closed
                background = OperatingSystem.isMacOS();

                //{{{ Parse command line [102 lines]

                //{{{ We need these initializations very early on [16 lines]
```

```
        {
                this.recorder = recorder;
        } //}}}

        //{{{ processKeyEvent() method
        /**
         * Forwards key events directly to the input handler.
         * This is slightly faster than using a KeyListener
         * because some Swing overhead is avoided.
         */
        public void processKeyEvent(KeyEvent evt)
        {
                processKeyEvent(evt,VIEW);
        } //}}}

        //{{{ processKeyEvent() method
        /**
         * Forwards key events directly to the input handler.
         * This is slightly faster than using a KeyListener
         * because some Swing overhead is avoided.
         */
```

```
Compiling /Users/jasonkost/jedit-cvs/jEdit/org/gjt/sp/jedit/View.java to directory /Users/jasonkost/proj/java...

Command line:
javac -classpath /Users/jasonkost/proj/java/org/jasonkost/jasonkost.jar:/Users/jasonkost/.jedit/jars/junit.jar:.:/Users/jasonkost/proj/java/:/Users/jasonkost/jedit-cvs/jEdit -g -d
/Users/jasonkost/proj/java /Users/jasonkost/jedit-cvs/jEdit/org/gjt/sp/jedit/View.java

JCompiler done.

JCompiler ready.
```

Figure 8. jEdit editing a few java files.

Eclipse is another IDE similar to jEdit and jExt. It is way more popular though, with a peak in 2004 with 30 million downloads that year (I have no information after 2005). Again XML and DTD files are used to control the editor. It seems that the plug-in structure is more sophisticated than with jExt and jEdit, if I remember it right the editor is not an integrated part in Eclipse but a plug-in as well. For instance in the Jason Eclipse plug-in I can find references to an ASLEditor and a MAS2JEditor, and at even deeper level I find jEdit. Eclipse by default already comes with many plug-ins as Cvs, Ant, debugging, refactoring, and this make it a more powerful IDE than jEdit 'out of the box'. For GOAL though, debugging and refactoring modules would have to be rewritten, the defaults will not be of much help.



```
// mars robot 1

/* Initial beliefs */

at(P) :- pos(P,X,Y) & pos(r1,X,Y).

/* Initial goal */

!check(slots).

/* Plans */

+!check(slots) : not garbage(r1)
   <- next(slot);
      !!check(slots).
+!check(slots).
```

```
<terminated> Debug MAS - jasontest2 [Java Application] /System/Library/Frameworks/JavaVM.framework/Versions/
user-init:

compile:
Compiling 1 source file to /Users/wouter/Desktop/testJasonEclipse/jasontest2/bin/classes

run:
```

Figure 9. Eclipse as Jason Central Controller

3. GOAL GUI Recommendations

This section starts with suggestions for the run-time GUI panels. Then integration into an IDE is discussed.

There are good standards for GUI design [apple07, UXG, javalf01]. As we write the interfaces in Java, the interfaces will automatically partially use the look and feel of the actual platform. I recommend to use the Java look and feel [javalf01] as our application may span multiple OSs.

The Platform controller GUI has to show a list of (GOAL) agents. This has to be a scrollable list, anticipating large agent sets. A global "run" and "freeze" button seems in place here. The halt button (the block) kills the entire GOAL platform. If you click on an agent, an introspector window might pop up for that agent. Furthermore it has to have a number of menus, one to add/kill/move agents, one to connect to other platforms, and one to load/kill an entire platform. Finally the platform controller may be the right place to show the general messages, warnings and errors of all agents. We may want to think about this further, anticipating nested GOAL agents. A proposal is shown in Figure 10.

The console output shows the "standard output" and all error messages and warnings. No debug information is shown here. As was recommended, all important messages are shown in a single output window, to make sure the user does not miss them.

The figure already shows a backward step button, we need to think about backward stepping, for instance up to 10 steps. To keep the agents consistent, backward stepping probably should not be applicable to a single agent but only to the entire MAS. We might be able to do backward steps by storing the entire agent state after every step. To keep some performance it would be nice if we need to do this only for agents that have an open introspector window but as it is now every step of every agent would need to be stored, because the back step is placed along with the run and forward step button in the GOAL PM GUI. We have to decide then also what to do when stepping back and forward again over a non-deterministic step. Finally we might consider a reference/link to original source code that causes the current step to be made in order to visualize effects of stepping. But that would require annotations in the belief base and as this beliefbase is currently only stored in prolog there seems no easy way to do this.

If you press the run, stop or step button with an agent selected, that action will be applied to that agent only. If no agent is selected, all agents will run (or one of them or all will make a step, to be determined).

Introspection and debugging are very closely related. Therefore I suggest to integrate the introspector and debugger into a single GUI, as in Jason and 2APL. I consider the "query" buttons in the current GOAL introspector as very useful. Assuming that users mostly will want to see the beliefs and sometimes the goals, a tab-based approach for these components, like Jadex and what we have already in the GOAL introspector, seems the best way. This gives the picture of Figure 11.

Clicking on a debug message might show the full "stack trace" to that point. We can also show the entire stack trace all the time like in Jadex and Eclipse/Java but I think that is not too useful, one mostly wants to see the bottom steps.

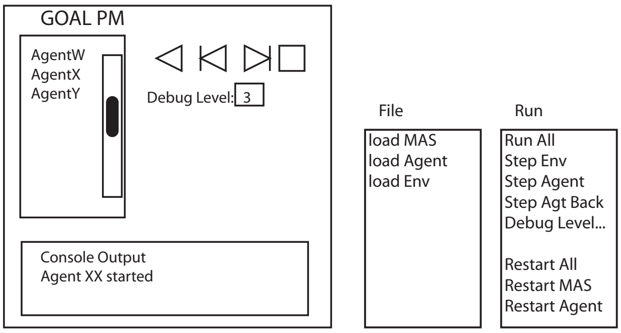


Figure 10. Proposed GOAL Platform Controller and its file and run menus.

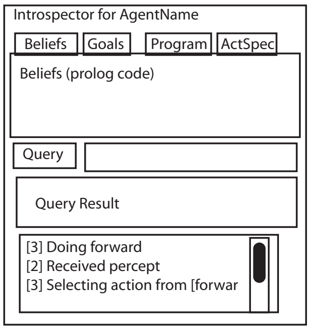


Figure 11. Proposed GOAL Introspector panel.

If you doubleclick on an agent, the introspector will be launched for that agent. You can view the databases and even do queries to the database.

A message sniffer may be a nice extra tool. I propose to keep it separate from the platform controller, like in JADE. By default it should show only messages from or to GOAL agents, not internal messages like RMI calls.

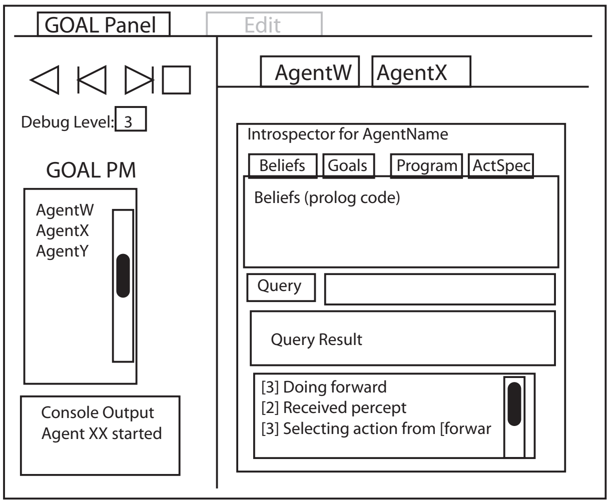
All functionality also has to be available through the menus (Figure 12). The middleware is loaded and started when needed. It is not yet clear whether we want to be able to handle multiple middleware layers.

Figure 12. File and Selection menus. Load can handle .goal, .mas, .jar (environment) files.

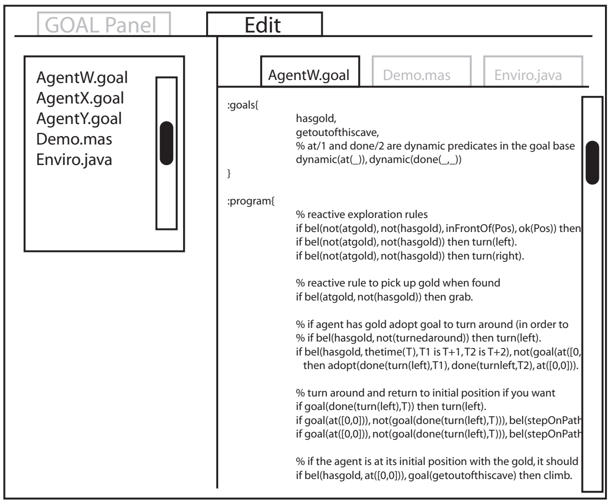
File	Selection...
load Preferences	Run Step Step Back Debug Level... Reload Kill

3.1 Integration into an IDE

The introspection button is a sort of properties window, and following the guidelines for property inspector [uxg08 p.469] or inspector windows [apple07] these should be a separate window. Of course the panels of Figure 10 and Figure 11 can be integrated into a single larger panel, or even an IDE e.g. Figure 13 and Figure 14. But I recommend keeping them separate. Separation makes it easier to resize important introspectors, stacking them etc. Resizing is important for introspectors to enable good view on the database and query results.



```
[3] Doing forward
[2] Received percept
[3] Selecting action from [forwar
```



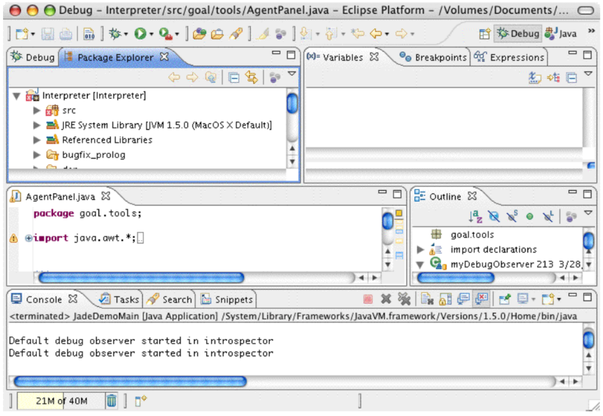
```
:goals{
                hasgold,
                getoutofthiscave,
                % at/1 and done/2 are dynamic predicates in the goal base
                dynamic(at(_)), dynamic(done(_,_))
}

:program{
                % reactive exploration rules
                if bel(not(atgold), not(hasgold), inFrontOf(Pos), ok(Pos)) then
                if bel(not(atgold), not(hasgold)) then turn(left).
                if bel(not(atgold), not(hasgold)) then turn(right).

                % reactive rule to pick up gold when found
                if bel(atgold, not(hasgold)) then grab.

                % if agent has gold adopt goal to turn around (in order to
                % if bel(hasgold, not(turnedaround)) then turn(left).
                if bel(hasgold, thetime(T), T1 is T+1, T2 is T+2), not(goal(at([0,
                  then adopt(done(turn(left),T1), done(turnleft,T2), at([0,0])).

                % turn around and return to initial position if you want
                if goal(done(turn(left),T)) then turn(left).
                if goal(at([0,0])), not(goal(done(turn(left),T))), bel(stepOnPath
                if goal(at([0,0])), not(goal(done(turn(left),T))), bel(stepOnPath

                % if the agent is at its initial position with the gold, it should
                if bel(hasgold, at([0,0])), goal(getoutofthiscave) then climb.
```

Figure 13. Possible integration of run-time windows in a larger GOAL panel in an IDE.

Figure 14. The Edit window of the IDE.

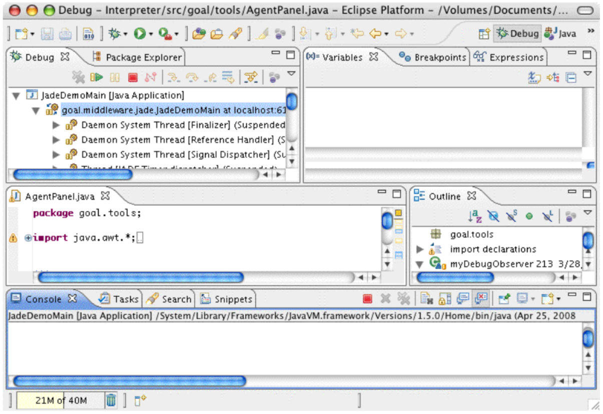
A second step could be integration of this run-time panel into the IDE. In spite of the similarity in layout with many editor windows (Figure 7, Figure 8, Figure 9) this GOAL panel has very different meaning. We are not *editing files* here in some file directory but *looking at agents* a multi-platform middleware system. It is my impression that file editors (jExt, jEdit) will not be able to make such a change in meaning.

Eclipse can make such drastic changes, for instance when using Java, if you switch to debugger mode your eclipse window layout will change from Figure 16 to Figure 17. Nevertheless the similarity in layout and contents hides this mode change, which is confusing. They seem even to have made this similarity to a design goal. For instance, the editor window is still there but now it is used as a tracer indicator. The Debug and Console tabs stay open all the time, as if the run time environment is still running. I recommend a clear visual distinction between the IDE that is for file editing, and a runtime GUI that is for managing an agent platform.


Debug – Interpreter/src/goal/tools/AgentPanel.java – Eclipse Platform – /Volumes/Documents/...
Debug
Java
Debug
Package Explorer
Interpreter [Interpreter]
src
JRE System Library [JVM 1.5.0 (MacOS X Default)]
Referenced Libraries
bugfix_prolog
Variables
Breakpoints
Expressions
AgentPanel.java
Outline
goal.tools
import declarations
myDebugObserver 213 3/28,
Console
Tasks
Search
Snippets
21M of 40M

```
package goal.tools;

import java.awt.*;
```

```
<terminated> JadeDemoMain [Java Application] /System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Home/bin/java

Default debug observer started in introspector
Default debug observer started in introspector
```


Debug – Interpreter/src/goal/tools/AgentPanel.java – Eclipse Platform – /Volumes/Documents/...
Debug
Java
Debug
Package Explorer
JadeDemoMain [Java Application]
goal.middleware.jade.JadeDemoMain at localhost:61
Daemon System Thread [Finalizer] (Suspended
Daemon System Thread [Reference Handler] (S
Daemon System Thread [Signal Dispatcher] (Su
Variables
Breakpoints
Expressions
AgentPanel.java
Outline
goal.tools
import declarations
myDebugObserver 213 3/28,
Console
Tasks
Search
Snippets
21M of 40M

```
package goal.tools;

import java.awt.*;
```

```
JadeDemoMain [Java Application] /System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Home/bin/java (Apr 25, 2008
```

Figure 16. Eclipse Java editing layout

Figure 17. Eclipse Java debugging layout

For an IDE we can opt with either jEdit or Eclipse. I have no specific recommendation here, I suggest we first investigate which one will be easiest to adopt to our needs.

I recommend to launch a run time interface in a separate window at the moment a run button is pressed in the editor. This is also the best way to keep things modular and manageable. Finally I recommend a clear visual difference between the IDE and the GOAL runtime interface.

4. Choices for GOAL platform

Choices were made for the GOAL platform together with Koen Hindriks. Table 1 presents the choices that were made for the GOAL platform, based on the recommendations of the previous chapters. Also, implementation priorities were determined.

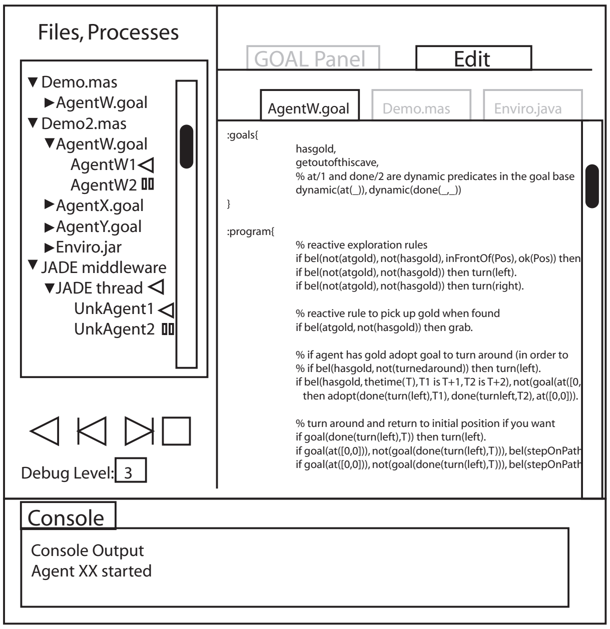
Table 1. Choices and priorities. Higher numbers indicate higher priority.

Recommendation (from page)	Choice	priority
Middleware continues running at MAS/Agent takedown	As recommended	-1
No ambiguities about status of middleware (1)	As recommended	-1
Hide middleware from direct access (1)	As recommended	-1
Error & Warning reporting via single console (1)	As recommended	-1
Use Java LAF (1)	As recommended	-1
Integration of debugger and introspector window (9)	As recommended	-1
Separate windows for GOAL runtime tools and IDE (4, 10)	Ignore recommendation	
Separate environment window (4)	As recommended	
Where possible use Swing, not SWT (6)	As recommended	
Clear visual distinction between runtime and edit environments (10)	Ignore recommendation	
Increase priority level for Linux support (4)	As recommended	-2
Improve installation process (4)	As recommended	-3
Provide message sniffer tool (9)	As recommended	-4
Investigate how hard it is to adapt jEdit and Eclipse (10)	As recommended	-4
GOAL runs on top of middleware (1)	As recommended	-5
Across-module debugging framework (1)	As recommended	-5
Errors & Warnings routing via middleware (1)	As recommended	-5
Environment control via middleware (1)	As recommended	-5
Use middleware abstraction layer (1)	As recommended	-5
Platform requirements and movability flag for GOAL agents (1)	As recommended	-6
Provide backward step button (4)	As recommended	-7

As indicated, the recommendations for the IDE are ignored. Koen chooses an alternative layout for the IDE as in Figure 18, Figure 19. The Files and Processes panel on the left shows an overview of the available files and processes. Processes that are shown under a .goal file are filtered out in the JADE thread info. Presumably the .mas file specifies the names of its instantiations; processes started with the run button The run-time GOAL Panel will have to stay empty and could even be unavailable until the run button is pressed. The console at the bottom is also part of the "Files, Processes" window, showing the status of the various processes.

Single clicking on a .mas, .goal or thread will select that item for control with the run buttons. Pressing run with a .goal or .mas file launches a new agent. Pressing run with a thread selected will resume that thread. The stop and step buttons work similar. Maybe it should not be possible to change the run status of unknown agents in the JADE thread.

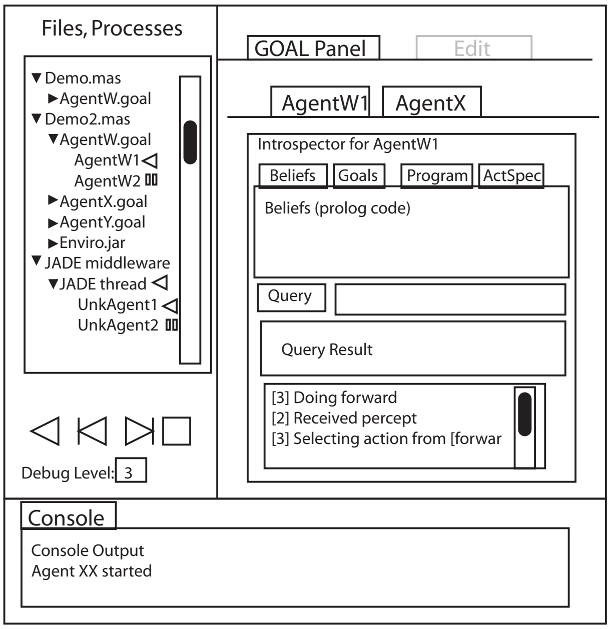
Double clicking a .goal or .mas file opens the file an editor. Editing a agent.goal file does not affect agents already running. Double clicking a thread associated with a .goal file opens the introspector for that thread.



```
:goals{
            hasgold,
            getoutofthiscave,
            % at/1 and done/2 are dynamic predicates in the goal base
            dynamic(at(_)), dynamic(done(_,_))
}

:program{
            % reactive exploration rules
            if bel(not(atgold), not(hasgold), inFrontOf(Pos), ok(Pos)) then
            if bel(not(atgold), not(hasgold)) then turn(left).
            if bel(not(atgold), not(hasgold)) then turn(right).

            % reactive rule to pick up gold when found
            if bel(atgold, not(hasgold)) then grab.

            % if agent has gold adopt goal to turn around (in order to
            % if bel(hasgold, not(turnedaround)) then turn(left).
            if bel(hasgold, thetime(T), T1 is T+1, T2 is T+2), not(goal(at([0,
              then adopt(done(turn(left),T1), done(turnleft,T2), at([0,0])).

            % turn around and return to initial position if you want
            if goal(done(turn(left),T)) then turn(left).
            if goal(at([0,0])), not(goal(done(turn(left),T))), bel(stepOnPath
            if goal(at([0,0])), not(goal(done(turn(left),T))), bel(stepOnPath
```

```
Console Output
Agent XX started
```

Figure 18. Editor of IDE.



```
[3] Doing forward
[2] Received percept
[3] Selecting action from [forwar
```

```
Console Output
Agent XX started
```

Figure 19. GOAL Runtime GUI. Running agents are green, halted agents red.

References

[apple07] Apple Human Interface Guidelines (2007). http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIGuidelines/XHIGIntro/chapter_1_section_1.html

[Ditchendorf06] Ditchendorf, T. (2006). Why bother with SWT over Swing? http://www.damnhandy.com/2006/02/12/why-bother-with-swt-over-swing.

[javalf01] Java Look and Feel Design Guidelines. http://java.sun.com/products/jlf.

[jEdit] jEdit Programmer's Text editor. http://www.jedit.org

[jExt05] Jext Free source code editor. http://www.jext.org/

[Kiewe06] Kiewe, H. (2006). SWT vs. Swing & Eclipse vs. NetBeans. Available Internet: http://www.javalobby.org/java/forums/t63186.html.

[Pasman08] Pasman, W. (2008). Architecture of Java-based Agent Frameworks. Available Internet: http://mmi.tudelft.nl/~wouter/publications/publ.html

[SWT] SWT: The Standard Widget Toolkit. http://www.eclipse.org/swt/.

[UXG] Windows Vista User Experience Guidelines. http://msdn2.microsoft.com/en-us/library/aa511258.aspx

[Yucel04] Yucel, S. (2004). What does "Swing is Slow" mean? Available Internet: http://www.javalobby.org/articles/swing_slow/index.jsp.