Table of Contents

Mango **2**
Capability 2
Installation 2
System requirements 2
Installing the Mango plugin 2
Installing the Mango development platform 3
Getting started 5
Examples 11
User Manual 11
Bugs 12
Contributing to Mango 13
History of Mango 13

Mango

Mango is a Java™ assurance tool, deployed as an Eclipse workbench plugin. Mango is developed as the jpf-mango extension of JPF.

Capability

Mango builds a functional model of Java code, exposed as navigable browser pages. When used as a sanity checker, Mango will halt at every case-split, allowing the programmer to compare code versus model. The functional view of code is often quite different from the code itself, and potentially reveals hidden flaws.

Mango may be of value as an educational tool. Those learning Java as a first programming language have the opportunity to compare procedural and functional views of code. This comparison facilitates the exposition of formal modeling concepts. For this purpose, the Mango installation contains a comprehensive set of first year Java code examples, together with the corresponding Mango formal models.

Mango may potentially be used to generate and apply filters for known Java security issues. Oracle has published a set of secure programming guidelines. The idea is to apply Mango to code snippets exposing potential security problems. By analyzing the corresponding formal model, it may be possible develop rules to recognize security issues. Mango has a highly sophisticated rule-based generalized pattern matching capability to enable such rule development. This is currently an active area of Mango development.

The Mango formal model may be translated into ACL2, an automated theorem proving language. In the past, formal proofs of Mango generated conjectures have been accomplished. Mango was originally created for this purpose, and this direction represents the future. But in all honesty a lot of work remains before tangible results may emerge. Anyone interested in pursuing this line of thought should contact the principal Mango developer, frankrimlinger+mango at gmail.com.

Mango was presented as a "formal advisor" at the 2011 JFP Workshop. The paper A Formal Advisor for Eclipse Java Development and corresponding slides give a technical overview of Mango.

Installation

To use Mango, it is only necessary to install the plugin. Certainly this will suffice for learning the basic operation of Mango. Actual application of Mango to a real project will probably require the feedback and flexibility of the Mango development platform. The recommendation at this point is to first install the plugin to learn the Mango basics, and then switch to the development platform for more serious work.

System requirements

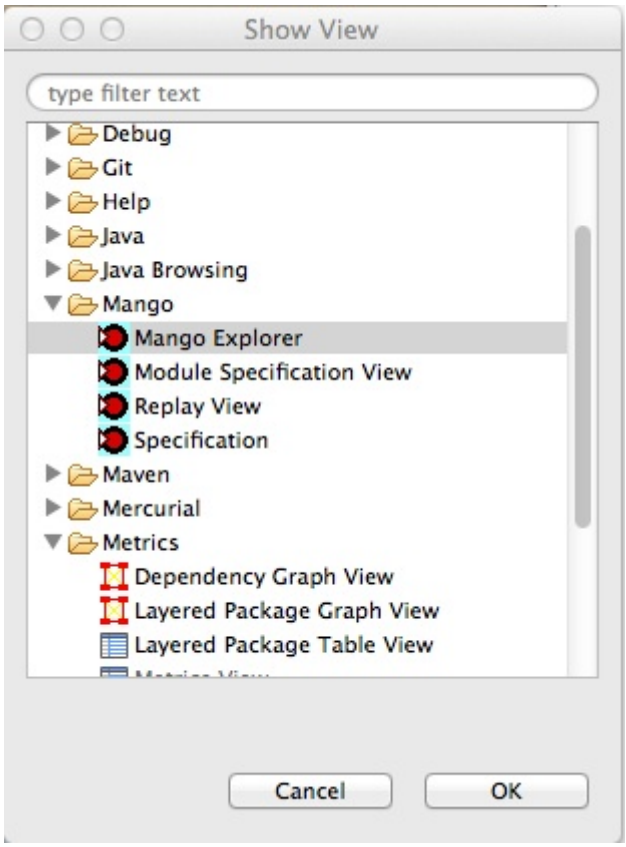
The plugin and development platform for Mango have been tested on the most current versions, as of April 27, 2012, of MacOSX, RedHat, and Ubuntu, using the latest release of the Eclipse workbench, Indigo. Mango should run anywhere the Eclipse workbench can be installed, assuming the underlying file system has Unix-like capabilities. Unfortunately, on Windows there is a hard limit of 260 characters on the length of a path name, which currently is a show-stopper for Mango. Mango relies on internals of jpf-core and the Eclipse platform itself, and these change over time, requiring changes in Mango. Every reasonable attempt will be made to keep Mango compatible with previous versions of its own database structure, but no guarantees. The plan is for active Mango development to support the most recent versions of jpf-core and the Eclipse workbench running on MacOSX, RedHat, and Ubuntu. However, the project is officially provided "as is", see the license for details.

Installing the Mango plugin

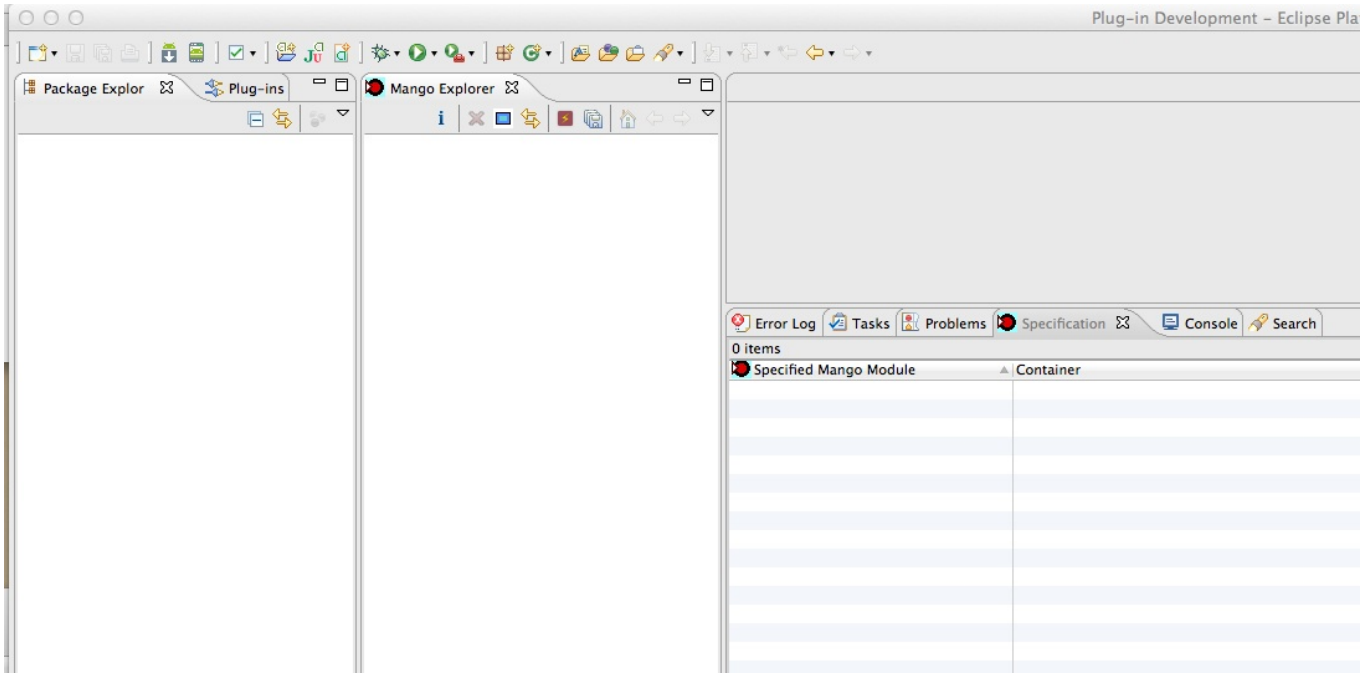
1. Install the most recent version of the Eclipse Workbench from the Eclipse downloads page. Mango is explicitly developed with the "Eclipse for RCP and RAP Developers" package, but there are no known conflicts with other packages. If you plan to install Mango into an existing Workbench, consider using a new workspace, at least until you have confidence in the stability of the plugin.
2. Switch to the Java perspective of the Eclipse Workbench.

- 3. Install the Mango plugin at update site http://babelfish.arc.nasa.gov/trac/jpf/raw-attachment/wiki/projects/jpf-mango/update/
  1. Go to Help>"Install new Software..."
  2. Enter http://babelfish.arc.nasa.gov/trac/jpf/raw-attachment/wiki/projects/jpf-mango/update/ in the "Work with: " box and hit carriage return, do not press the "Add..." button.
  3. Check the box next to "Mango" when it appears in the window. Hit "Next>".
  4. The feature version will display. Hit "Next>".
  5. License terms must be accepted to acquire the plugin. If acceptable, select "I accept ..." and hit "Finish".
  6. There will be an "unsigned content" security warning. Mango is still developing and there is no content security. Hit "OK".
  7. Hit "Restart Now". Mango currently will not install hot, as this leads to window rendering errors.
- 4. Upon reboot of Eclipse, the "Mango Explorer" and "Specification" views should appear in the java perspective. If these views do not appear or have been closed, they can always be recovered as follows (see figure below):

1. Go to Window>"Show View">Other... and open the "Mango" folder.
2. Double-click on "Mango Explorer". The "Mango Explorer" view will appear in the java perspective.
3. Repeat step a, and double-click on "Specification". The "Specification" view will appear in the java perspective.



1. Recommendation: Drag the "Mango Explorer" tab so that both "Package Explorer" and "Mango Explorer" views are visible, next to each other (see figure below). If you have never moved views around before by dragging their tabs, this will require some experimentation. You can always go to Window>"Reset Perspective..." if things don't work out on the first try.

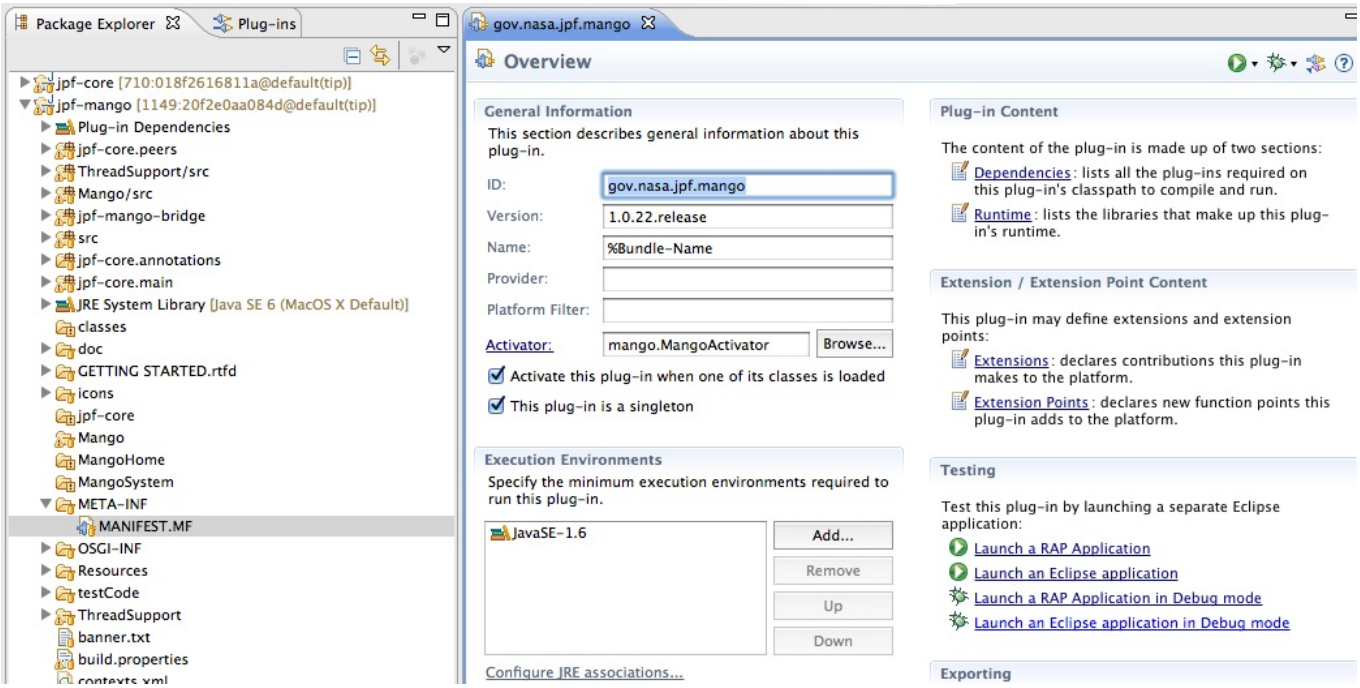


Installing the Mango development platform

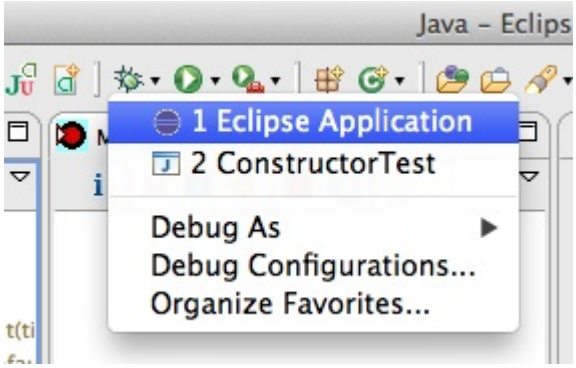
The Mango development platform allows you to extend Mango functionality and/or fix bugs. If you are new to Mango, you may wish to go directly to Getting Started.

1. Install the most recent version of Mercurial.
2. Install the Mercurial Eclipse plugin into the Eclipse workbench. The site for this plugin is http://cbes.javaforge.com/update
3. Add the **jpf-core** and **jpf-mango** projects.
    1. Go to File>New>"Project..."
    2. Open the Mercurial folder. Select "Clone Existing Mercurial Repository". Hit "Next".
    3. Enter the URL: http://babelfish.arc.nasa.gov/hg/jpf/jpf-core Hit "Next".
    4. The default directory revision will appear in the window. Hit "Next".
    5. The "Import Projects" window will appear. Hit "Finish".
    6. The jpf-core project now appears in the Package Explorer. Observe the project build messages in the console. If there is an indication that javac cannot be found, then the build has failed. The other possible error messages are normally harmless. If javac cannot be found, it is probably an installation problem with java or a failure to set the proper environment variables. Googling for instructions particular to your operating system should yield a solution.
    7. Repeat steps a-e for jpf-mango, using the URL http://babelfish.arc.nasa.gov/hg/jpf/jpf-mango. Sometimes **jpf-core** and **jpf-mango** get out of sync, and sometimes the most current version of **jpf-mango** is unstable. In this case, the Mercurial commit notes for **jpf-mango** should be consulted.

- 4. You must now set up the runtime for the Mango development platform. Click on "Launch an Eclipse application in Debug mode".
    1. In the **jpf-mango** project, locate "META-INF/MANIFEST.MF" and open this file in an editor (see figure below).



- 4. continued:
    1. The Mango development platform will open. This is just an Eclipse application which recognizes the **jpf-mango** project as one of its plugins. You can set break-points in the **jpf-mango** project, and the Eclipse debugger in the base workbench will honor them as the development platform runs. This is the manner in which Mango is developed.
    2. You may need to switch to the Java perspective and adjust the Mango views as in step 5 of the Mango plugin installation.
    3. Close the Mango development platform. Henceforth, use the "Debug as..." icon to launch the development platform (see figure below).

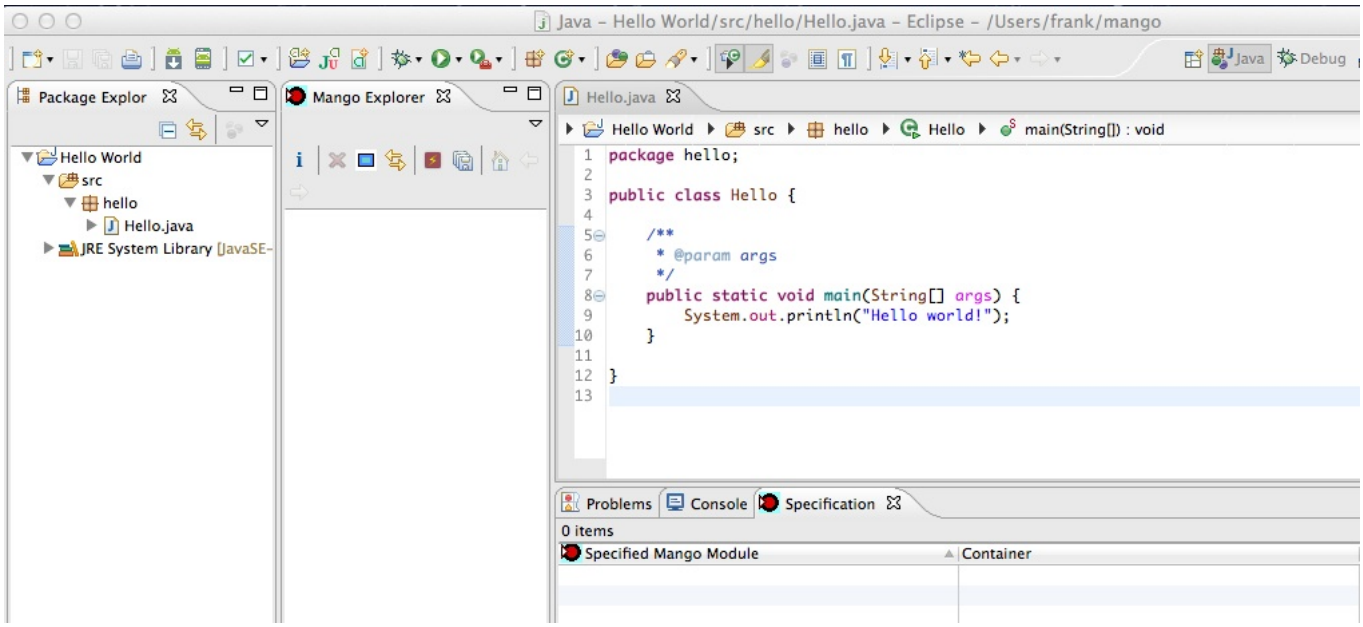


- 4. conclusion:
  1. In your home directory, Eclipse has created a runtime folder for the Mango development platform. Locate this folder, it should have a name such as "runtime-EclipseApplication". This folder should appear empty. In the next step you will provide content for this folder.
- 5. Add the MangoHome and MangoSystem projects to the Mango development platform. If you have already installed the plugin and built a specification, then versions of these projects already exist in a base workspace. However, these versions may not be in sync with the current jpf-mango project. Therefore, it is always best to get fresh versions from this site.
  1. At the bottom of this site, expand "Attachments". Double-click on "MangoNucleus.zip" and then on "downloading".
  2. Unzip the file. Ignore the MacOSX artifact if it appears, and locate the MangoSystem and MangoHome folders.
  3. Move MangoSystem and MangoHome to the Mango development platform runtime, located in step 4e.
  4. Re-open the Mango development platform, see step 4d.
  5. Select File>New>Project...>"Java Project". Hit "Next".
  6. Make sure "Use default location" is checked. This is the runtime folder.
  7. Enter "MangoHome" for the project name. The message "The wizard will automatically configure ..." should appear because MangoHome is in fact already in the runtime folder. Hit "Finish".
  8. Repeat steps e-g for "MangoSystem."

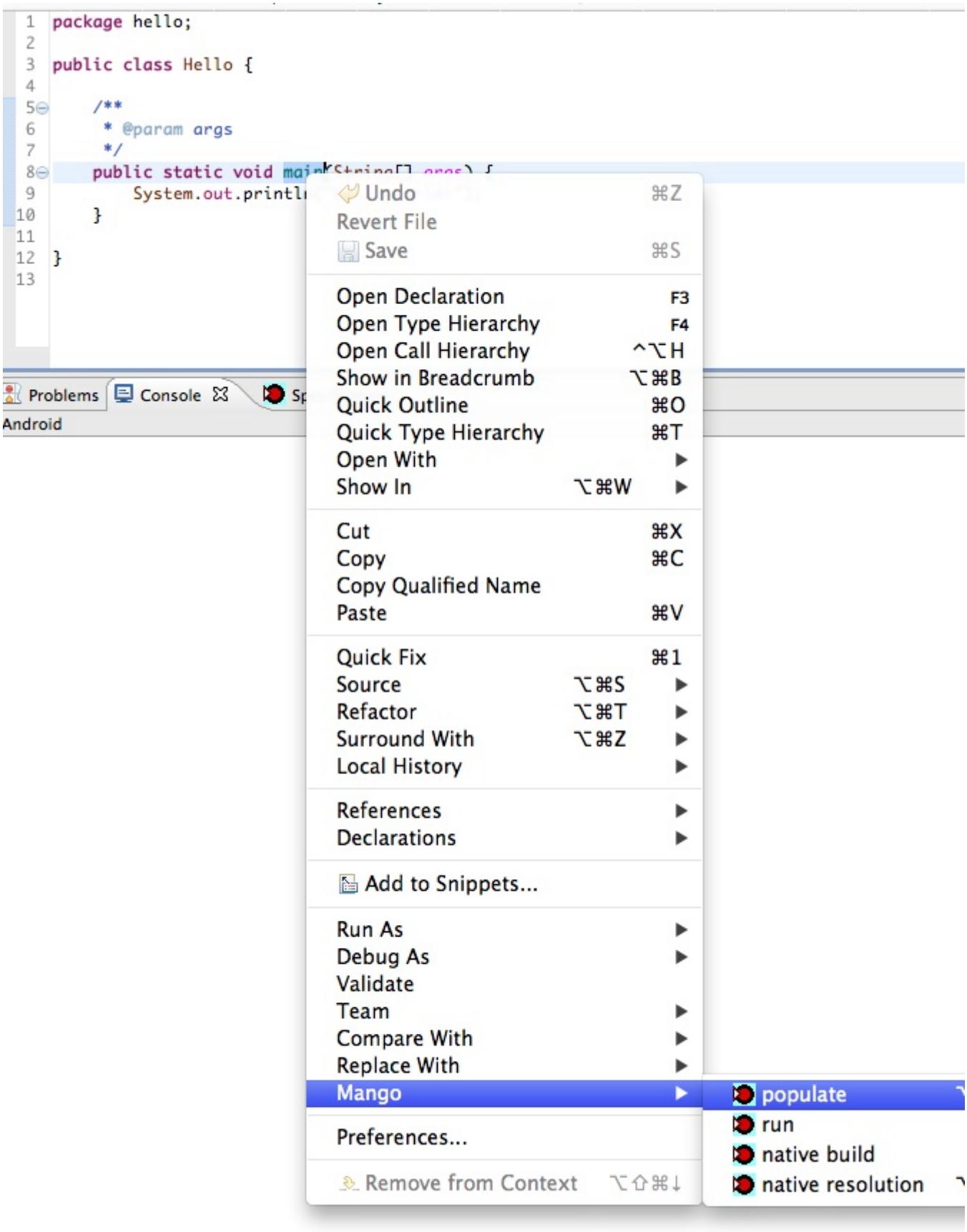
Thats all there is too it! Keep this in mind: " **jpf-mango** is to Mango development platform as Mango plugin is to base workbench. That is, whatever you can do in the base workbench with the Mango plugin installed, you can do in the Mango development platform while running/debugging **jpf-mango**. Because the base workbench is observing the development platform, you can also get much better feedback for diagnosing problems.

Getting started

- Let's work through the traditional "Hello World" example.
  1. In the Java perspective, go to File>New>"Java Project".
  2. Enter "Hello World" for the project name, and hit "Finish".
  3. Right click on the "src" folder for the "Hello World" project. Add a new class to the Hello World project, checking the box to build the main routine.
  4. Add the traditional "Hello world!" println to the main routine, (see figure).

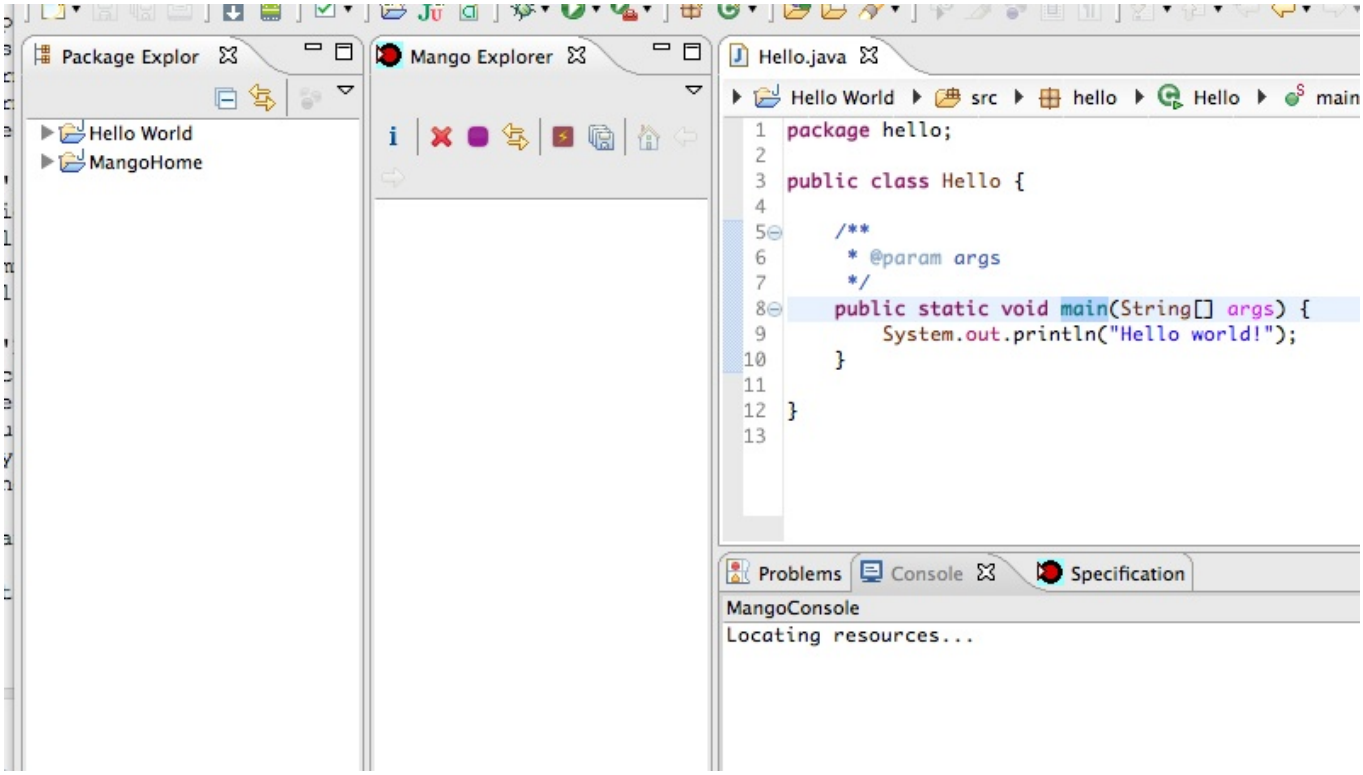


- "Hello World" continued:
  1. Build the project. This will happen automatically if Project>"Build automatically" is checked.
  2. Double-click on "main". This will ensure that both "main" and the "Hello World" tab are selected.
  3. Right-click on "main", and select Mango>populate (see figure).

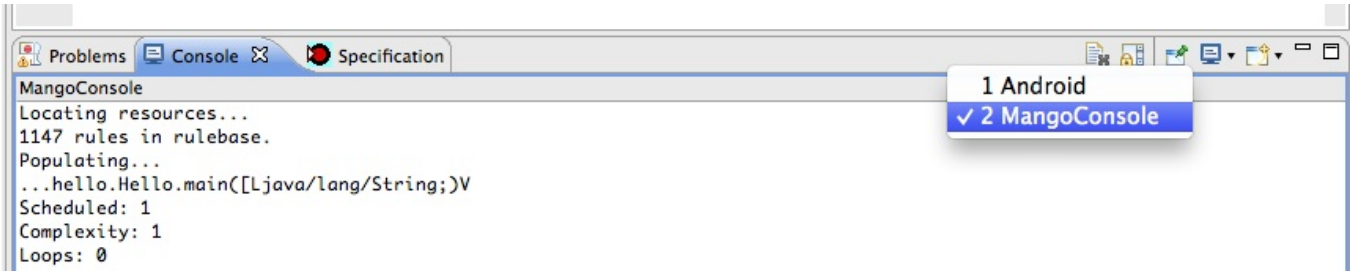


- "Hello World" continued:

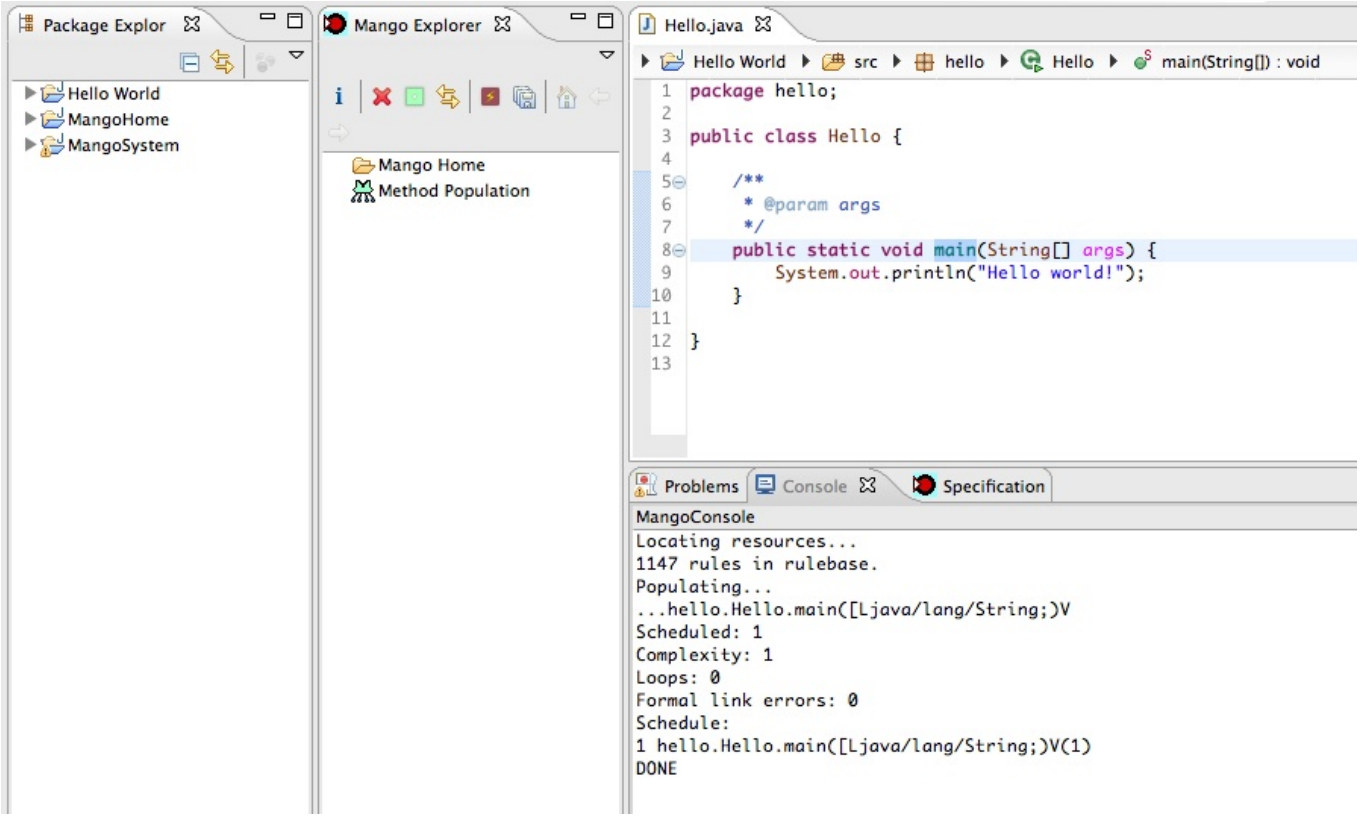
1. The first time the "populate" command runs, it must install MangoHome and MangoSystem in the workspace. This takes some time. The box just below the "Mango Explorer" tab is purple while the "populate" command is running. Also, various comments will appear in the Mango console view (see figure).



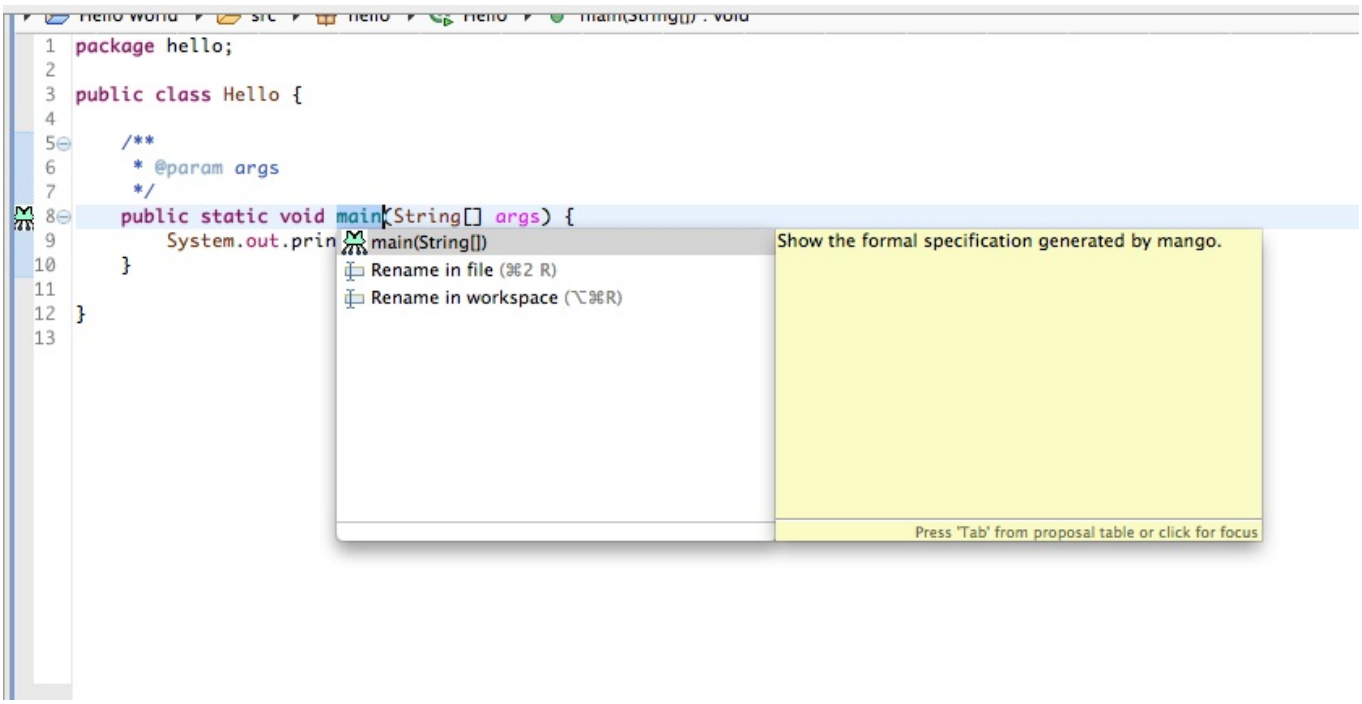
- "Hello World" continued:
  1. If you have no console view, now is a good time to install it, via Window>"Show View". If you have multiple plugins installed, the console view may not be tuned to Mango. You can view the Mango console by selecting it with the Console control (see figure).



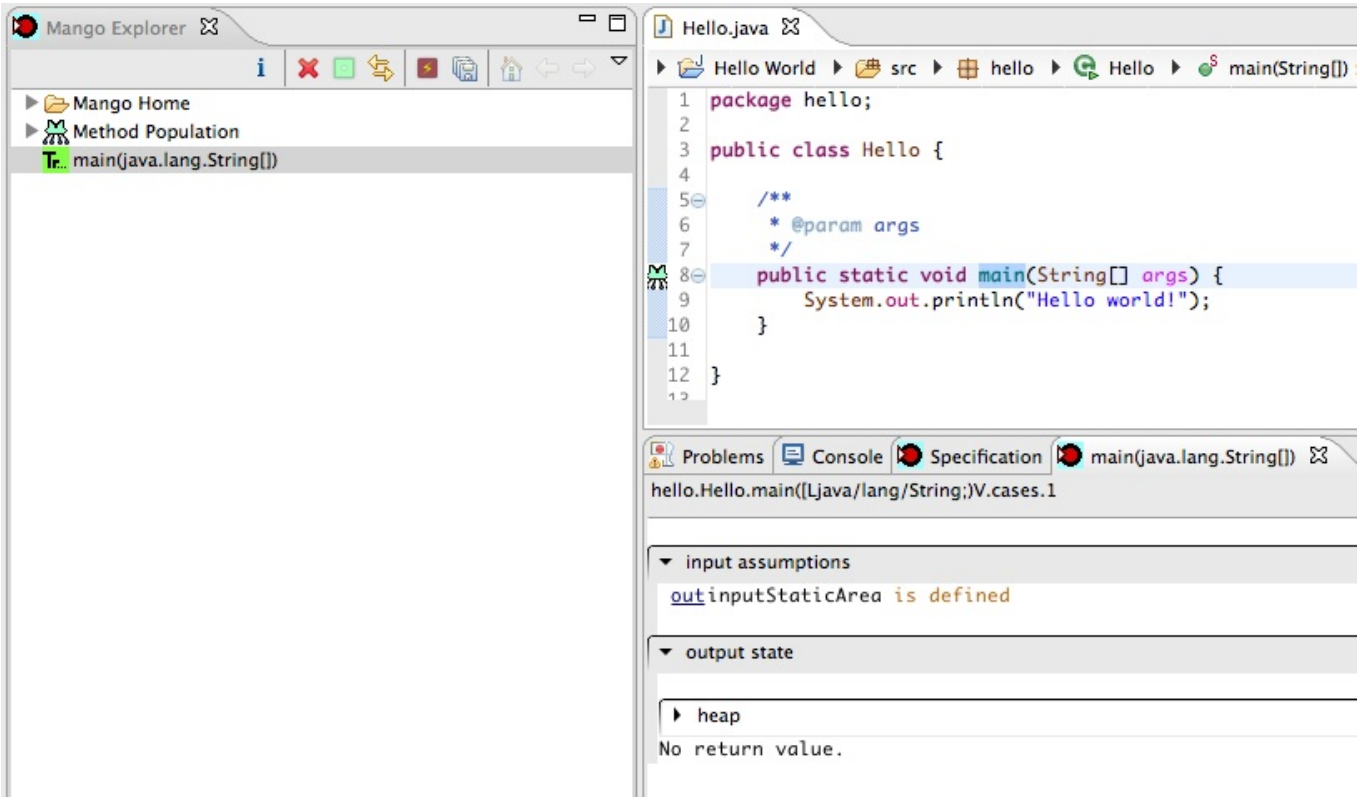
- "Hello World" continued:
  1. When the "populate" command completes, the box will turn green, and both MangoHome and MangoSystem will appear as projects in the workspace (see figure). These projects are required for the proper functioning of Mango. You can learn more about them in the examples and Users Manual sections.



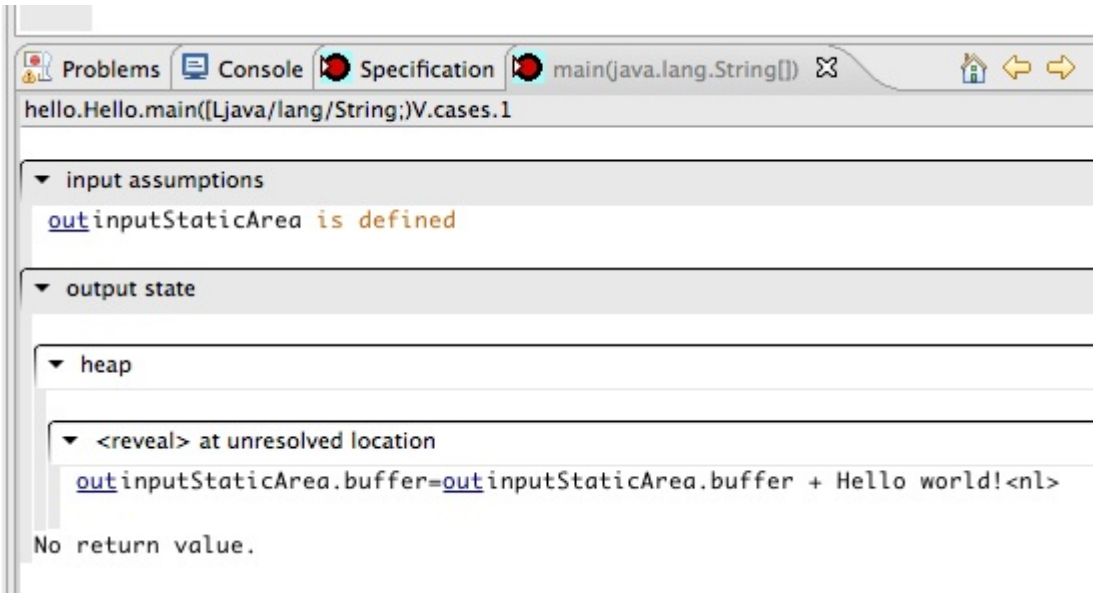
- "Hello World" continued:
  1. Double-click on "main" and then right-click, selecting Mango>run. While Mango is running, the box will appear red. It will turn green when Mango is done. A green "method icon" will appear in the editor view gutter next to "main". (Because of a bug, this icon may appear as a folder, but it will work just the same.) Click on this icon to reveal the specification task. (see figure).



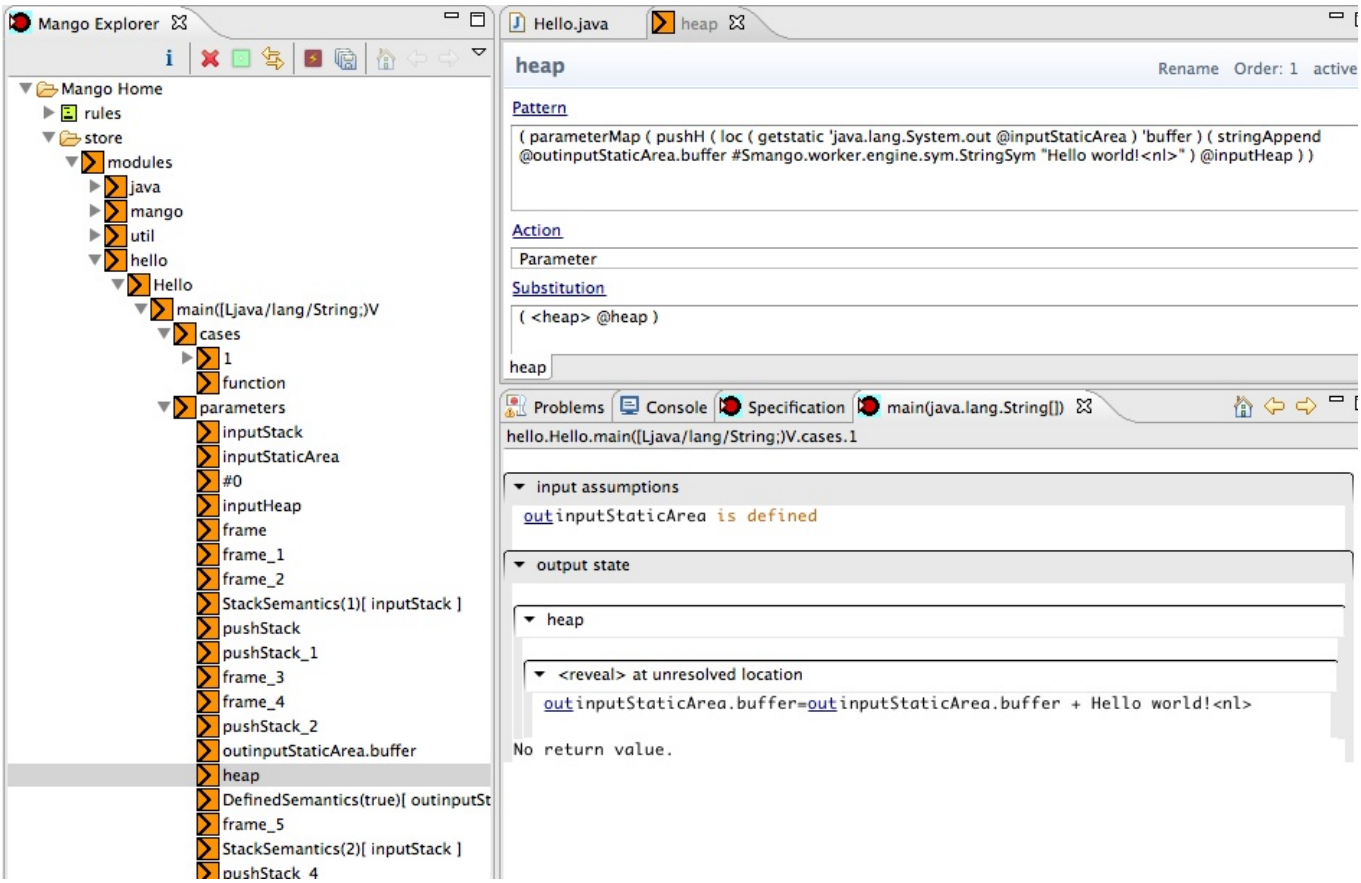
- "Hello World" continued:
  1. Select the "main(String[])" task, and hit carriage return or double-click. The method translation icon, a green "Tr" icon, will now appear in the Mango Explorer view. Finally, double-clicking on the translation icon will reveal the translation for the main method (see figure).



- "Hello World" continued:
  1. The most interesting content is contained in the heap, which currently is closed. Open the drop flags for the heap and individual heap item to reveal this content. As expected, the string "Hello world!" is added to the buffer of the "out" object (see figure).



- "Hello World" continued:
  1. The navigable translation of step 13 is a rendering of a precise specification stored persistently in the MangoHome project. You can view MangoHome content in the MangoExplorer. Try this now, navigating to the main([Ljava/lang/String;)V method folder. View the heap editor by double-clicking on the "heap" item in the "parameters" folder. This description of the heap is in so-called Mango Formal Language (MFL). (see figure). You can also double click on the main([Ljava/lang/String;)V folder to recover its translation icon, which is not persistent.



- "Hello World" concluded:
  1. This concludes the "Hello world" exposition. Of course, there is a lot more going on. From here, you can take a look at the examples or the user manual. If you would like to become a Mango developer, please also take at the bugs and current development.

Examples

===CarRecall=== The Mango plugin installation contains various example projects. The largest of these projects is FirstYearCode, which contains example code from a freshman high school Java course. The `CarRecall` example illustrates how Mango reports case-splits and loops. There are about 60 such examples in the FirstYearCode package. Mango builds the specification for 50 of these, the other ten contains errors, see bug #3 and the "First year status" attachment to this page. The procedure for generating and rendering the `CarRecall` specification is basically the same as the Hello World example, but more user interaction is available to promote greater understanding.

CarRecall exposition.

===ItsAWrap=== The testCode/rbk directory contains projects written with the intent of proof artifacts for ACL2. For one example, `ItsAWrap`, proof artifacts were in fact generated and actual proofs in the ACL2 logic were accomplished. However, the code within Mango for this pipeline is currently broken, see bug #6. The exposition of `ItsAWrap` shows how the user can introduce hypotheses into Mango to guarantee loop termination. Ideally, such hypotheses should be generated automatically, seebug #7.

ItsAWrap exposition.

===Sample Sync Adapter=== Several classes of the Android Sample Sync Adapter code have also been specified using Mango. Analysis of this middleware code necessary introduces lots of uninterpreted primitives representing code called but not modeled by Mango. So-called "formal link errors" can be resolved by Mango in a number of different ways. These techniques are still early in the development cycle and not suitable for exposition here. Moreover, introduction of large numbers of uninterpreted primitives makes the Mango specification cryptic, to say the least. However, the point of the Sample Sync Adapter exercise is to lay the foundation for automated testing of security properties, as discussed above.

User Manual

The users manual is currently under development. Content should be available for each of these links in the near future.

How to report a bug.

How to delete the gutter icons.

How to delete the gutter icons.

How to display a spec whose gutter icon is deleted.

How to recover from a spec failure (synchronization)

How to serially specify multiple targets (synchronization)

How to replay an existing spec (package explorer + synchronization)

How to satisfy a link error

1. Native build
2. Native resolution.
3. Addition of MangoSystem. (trickiness with interfaces and MangoFormal)
4. Adding a project to search scope (in preferences)
5. Creating native spec by hand.

How to moderate specification behavior (discussion of preferences and how they affect specification)

How to swap in a different MangoHome + MangoSystem

How to swap in individual modules from a different MangoHome (storeConfig regeneration)

How to write a rule (rule base gui explanation)

How to write an action (for developers only)

Bugs

- This is the list of the most important issues and bugs at the time of the initial plugin release. To work on a bug, please let me know (Frank Rimlinger: frankrimlinger+mango at gmail.com). I will create a bug ticket for you and update it as progress is reported. The ranking here should not be interpreted as priority, all of these bugs are important.

1. **No Windows support**. Mango uses the native file system as a sort of ready-made data base. This causes trouble if there is a design limitation on the number of segments in a path. Apparently Windows has a 260 character limitation on the length of a path name, which causes trouble. Windows is not the only issue here. The expedient of using the file system also causes annoying latency at the end of a specification, when potentially hundreds or thousands of tiny files have to be pushed to the file system. Two possible solutions come to mind: 1) interpose an open-source memory-based virtual file system between Mango and the native file system, or 2) move to a full-fledged open-source data base solution such as the Apache DP project.
1. **Incomplete/inconsistent support for editing of shadow rules and native specification**. The implementation of cut, paste, save, etc. do not work uniformly in a predictable manner across all the different models supported by the Mango Explorer.
1. **Some FirstYearCode examples have bugs which prevent complete specification**. See the "First year status" attachment to this page for a list of these bugs. These bugs directly impact the soundness of Mango, and I will work on them as time allows. A deep knowledge of Mango internals is probably required to make much progress on these bugs.
1. **No rule base search capability**. A long time ago there was an capability to search the rule base by rule key appearing in the pattern, substitution, or hypothesis, or any combination thereof. This capability was exposed in a familiar, easy to use manner. Needless to say, it is still needed today by those who need to write rules and understand rule base behavior.
1. **Some formal link errors are not reported correctly during the population build, especially those involving the MangoSystem**. Mango uses Eclipse internals to search the workspace projects. This can be challenging because documentation for these internals is spotty. In addition, the dependence search for MangoSystem is simply broken at this point.
1. **ACL2 proof artifact generation is not hooked up**. The code for artifact generation was written about two years ago, and has broken due to subsequent design drift. This code needs to be tested and updated. Familiarity with the ACL2 theorem prover is a pre-requisite, because you need to know what the point of these artifacts is in the first place. Ideally, this capability would be integrated with a solution to bug 7 below to produce end-to-end automated theorem proving for Mango conjectures.
1. **Automated loop termination hypothesis generation is missing**. Currently, Mango tracks each path through a loop and reports on exit point conditions, but no use is made of this information. This problem is obviously unsolvable in general, but practical solutions abound. Familiarity with the literature on the loop termination problem would be a plus here.
1. **Specification rendering is often unintelligible or uninformative**. Although the structure of the Mango formal language (MFL) and the MangoHome has settled down, rendering is still in a more or less experimental, ad hoc phase. One particular issue is the need for look-behind to report something

more informative than "op0" for an operand. Also, the whole issue of resolving assumptions and state on-the-fly as the user navigates a specification really needs work. Moreover, the choice of FormText as the content delivery technology is questionable, as is the choice of using Eclipse platform views, as opposed to editors.

1. **2D support for the case split algorithm and the loop algorithm is missing**. Since these algorithms are pretty stable now, this is not an operational concern. However, for pedagogical reasons, it would be good to be able to view the generated graphs. In earlier times, these graphs were beautifully rendered by a custom Sugiyama layout algorithm driven by the open-source graphics engine Gumbo. More recently, GEF and Zest have been employed. Because of difficulties acquiring the correct version these plugins, they have been omitted from the plugin release.
1. **The Mango gui has bugs and omissions**. Aside from the embarrassing appearance of the folder icon noted in the starter demo, there are lots of issues. For example, the MangoHome drop flag does not appear in the Mango Explorer until a specification is posted or a sync operation occurs. This is puzzling because this used to work just fine. Also, the Mango commands should respond to package level targeting. This is especially necessary for regression testing when large numbers of classes are involved.

Contributing to Mango

If you are a student and potentially would like to contribute to Mango via the Google Summer of Code Program 2013, please let me know as soon as possible (Frank Rimlinger: frankrimlinger+mango at gmail.com). I know that Summer 2013 is a long time from now, but a show of interest would be most helpful. If you are a member of the open-source community and would like to contribute to Mango on an informal basis, see the bug list.

History of Mango

To gain a sense of how Mango has evolved, a look at the Archived Examples is instructive. The following was written in 2008 at the origination of the Mango project.

Purpose: To provide a case by case specification of Java™ source code, analogous to Javadoc but more rigorous. To provide "proof artifacts" to a theorem prover, enabling the mathematical proof of code properties.

Ultimate Goal: To integrate the generation of code, specification, and correctness proofs so that a programmer may produce a more reliable product with the same level of effort.

Theory of Operation: The code is first converted to a large graph of vertices (instructions) and directed edges (branch conditions), often referred to as the flow control diagram. This conversion occurs at the byte code level, which is convenient because the byte codes are described very succinctly in terms of the state of the Java Virtual Machine (JVM). This conversion is essentially accomplished by the JPF core engine. The Mango formal peer code then generates for each byte code a description in terms of the Mango formal model. A graph subdivision algorithm is applied to the control flow, generating a hierarchical sequence of graphs required to describe the loops within the code as recursive functions. JPF is then used to walk these graphs for the purpose of generating the specification and proof artifacts. The backtracking ability of JPF is utilized to generate cases, and the ability to trace back along trails is leveraged in order to compute loop invariants. Ideally, the process would be fully automated, but in practice when Mango requires guidance from the user, JPF will block and a gui thread will interact with the user to obtain the information required to proceed. The specification and proof artifacts are stored in persistent form exposed to the user in a rule base format, enabling reuse and incremental, distributed operation.

Limitations: Mango only specifies "good" cases, exposing input constraints required to satisfy such cases. Some user guidance is required to determine what constitutes a good case. It may happen that even the number of good cases grows exponentially. The user must then provide case generalization logic to abstract away explosive case growth. Such abstractions must also be accompanied by type and translation logic, which the user must provide. Although Mango does generate loop termination conjectures, such conjectures are generally in terms of loop output. Typically the user must provide guidance to form hypotheses for loop termination in terms of loop inputs. Correctness of such hypotheses may be confirmed by an automated theorem prover, which typically requires expert guidance.

Status: Mango is based on technology released by the Nasa/Ames Software Release Authority in September, 2008. Much of the original code base was written in C++. By Spring, 2009, the code base was migrated to 100% pure Java, and integrated with Eclipse RCP, the rich client platform. The original code base did not use JPF. Full integration with JPF commenced in the Summer of 2009 and should be complete by January 2010. The tool should achieve an initial operational capability by Spring 2010. Ultimately, the tool will be deployed as an Eclipse workbench plugin.

Research: Previous versions of the tool did generate artifacts for the ACL2 theorem prover and substantial proofs were accomplished. Efforts are just getting started to lightly embed the current Mango model in ACL2. However, the proof artifacts are essentially just expressions in the Mango model of definitions, hypotheses and conjectures, and as such are theorem prover neutral. There is no automatic facility for generation of hypotheses on loop inputs required to prove termination. Needless to say, this is a fundamental tool weakness and contributions in this area would be most welcome. (Caveat emptor: there is no general purpose algorithm for determining loop termination, but constructive solutions exist for typical circumstances.)