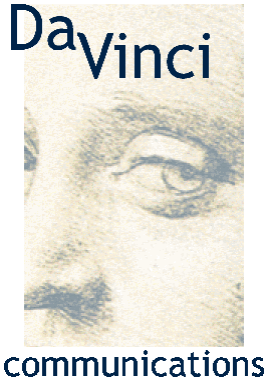


Da Vinci Communications Limited

Technology White Paper

TTCN Compiler Plugin for Leonardo Editor Pro

Da Vinci Communications Ltd.
PO Box: 38 078
Christchurch 8030
New Zealand

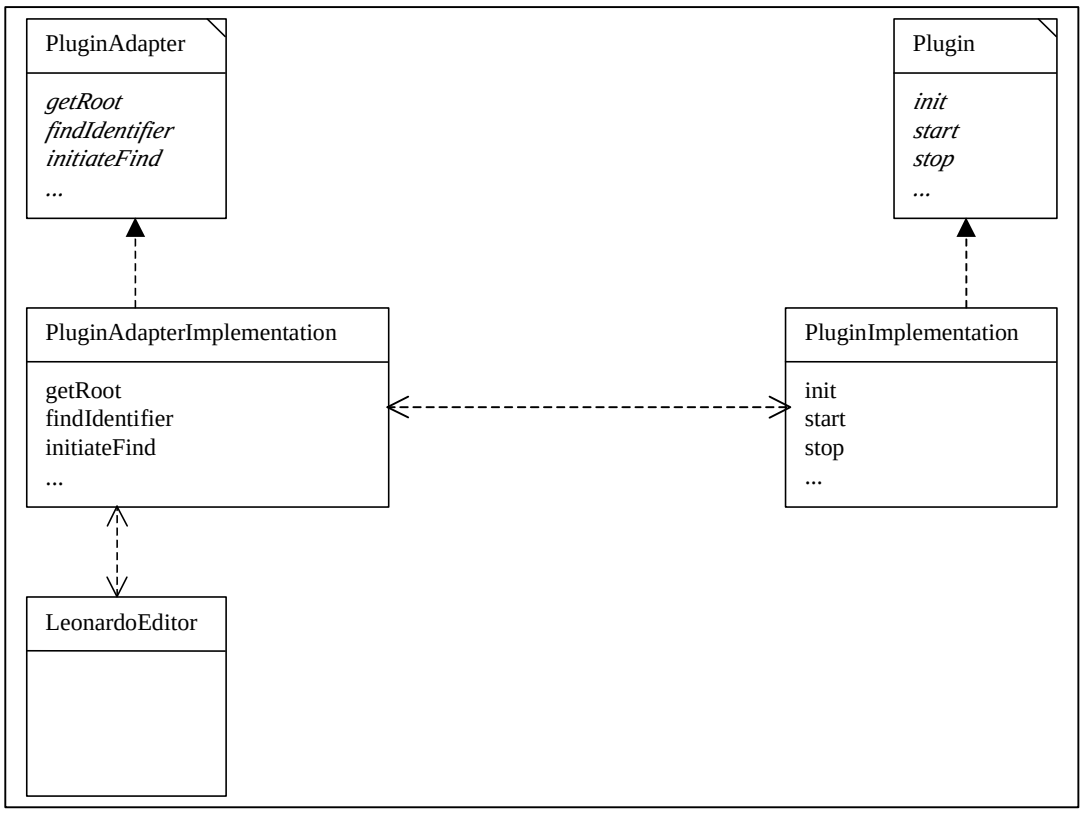
tel.: +64-3-3838311
fax: +64-3-3838310
email: info@davinci-communications.com
web: www.davinci-communications.com
WP100.02 05/01

Contents

Leonardo plugin interface

The Leonardo plugin interface is a relatively simple but very powerful component of the Leonardo Editor Pro. It enables the editor to integrate with third party tools without customising the editor. The main part of the editor can concentrate on being an excellent editor and delegate all non-essential tasks to individual plugins. The function of plugins is not limited to interfacing other tools, but can also serve to extend the functionality of the editor itself.

Architecturally the plugin interface consists of two main parts. These parts are Java interfaces which define the methods used by the editor and the plugin to communicate.
The following UML class diagram illustrates the concept:



The methods currently defined in the interface specifications reflect a first attempt to capture all useful and necessary requirements. However, it is anticipated that during the future design of advanced plugins new requirements will come up. These will create an evolution of the interfaces. Da Vinci Communications already has a strategy in place to allow for a smooth upgrade and backward compatibility with older type plugins. This will be accomplished by using inheritance, polymorphism and reflection. It will not be necessary to recompile all plugins in the field.

The plugin implementation can only use the functions declared in the PluginAdapter interface and the editor can interact with the plugin implementation through the functions declared in the Plugin interface. Additionally, plugins have access to the internal data structure, which holds all the TTCN data. This is much more flexible than hook functions into the parse tree, as a plugin can freely navigate within the data.
Plugins are packaged as JAR (Java Archive) files. The archives can contain all the necessary data the plugin needs, eg. class files, images, data files, etc.. It is also possible to package several plugins into one archive. At startup, the editor searches a configurable directory and tries to load all plugins found. Each instance of the editor instanciates its own plugins.

Plugins can have state, which will be preserved between sessions. When the editor is exited, all plugins which signal their need to store information are asked to store their state in a property map. This state information is stored per plugin class. When the editor is started again, the editor delivers the stored information during the initialisation of the plugin and the plugin can restore its previous state. This is very useful for configuration options and the like. The plugin designer does not have to worry about the storage itself: the editor provides this service transparently.
Another useful plugin feature is the “auto start” option. Plugins which provide a background task, eg. listening to communication ports etc., can elect to be started automatically after initialisation without user interaction.

TTCN Compiler Plugin

Overview

The example TTCN compiler plugin introduced here will facilitate the use of a TTCN compiler from within Leonardo Editor Pro. The tight integration of both tools will increase the useability of the compiler by providing a graphical user interface and allowing the use of editor features for debugging purposes. It is also easily imaginable to provide a graphical user interface for the whole process cycle of editing, compilation, parameterisation, execution, tracking and debugging through a more extensive plugin implementation. This would have the advantage for the user of being able to accomplish all tasks from within the same familiar user interface.

The use of plugins to drive TTCN tools opens up other possibilities through the underlying Java environment. The example TTCN compiler plugin makes use of Remote Method Invocation (RMI) to invoke the compiler on a remote host. With the compiler running on a different host, the process is still completely transparent to the user, as in a local compilation. This remote compilation is not limited to a LAN but could be done across the Internet, provided sufficient bandwidth is available. A whole new business model could be possible through the use of these facilities.
We considered the following reasons for the implementation of a remote compilation feature:

- users can use the editor on platforms which are not yet supported by the TTCN compiler without losing complete GUI support
- the compiler can be deployed on the most powerful hardware
- compiler tasks can be distributed and load-shared across the network
- client hosts do not have to provide resources for the compiler
- compiler configuration and administration can be done centrally
- necessary libraries and modules can be stored centrally

Local compilation

In the case of local compilation the plugin will invoke the TTCN compiler on the local host. The location of the compiler executable, the generation directory, and all compiler options are configurable via the options dialog. The compilation can be cancelled from the output dialog.
The output of the compiler is captured and presented to the user in a dialog form. The plugin parses the output of the compiler before it is output to the user. The parsing process is done with the help of a Perl5 regular expression package, which is extremely powerful. The plugin searches for error or warning messages and presents the user with interactive elements to aid in the debugging process. At present, three elements are implemented.

1. Syntax errors

   A syntax error in the MP source code is always reported with a line number. A button after the error message allows the user to open an MP source view of the relevant portion of the source that contains this line. It is also planned to correlate the line number to the TTCN data object and have the editor display the relevant proforma.
2. Missing identifier

   The TTCN compiler reports missing identifiers that cannot be found in the source. A button after this error message lets the user open a Find dialog, initialised to the identifier string and configured for full text search. The user can use this dialog to search for occurrences of the identifier.
3. Semantic errors (implementation pending)

   The TTCN compiler can output editor invocation commands to call the editor to present the faulty definition for editing. The plugin will insert a button replacing this command to display the corresponding proforma in the editor.

This list is only a start. The very flexible mechanism used to implement it, can easily accommodate further features. All that is needed are detailed error messages from the compiler.

Remote compilation

The remote compilation feature of the TTCN compiler plugin is made possible by the use of the Java RMI API. To use this feature a small server application needs to run on the host where the compiler runs. This remote compiler server will advertise its services through an RMI registry created by the server. Clients need to be configured to contact this host in the plugin's options dialog. It is also be possible for the server to re-route the request to a different host where another remote compile server is running. This can be used to implement a load-sharing scheme if necessary.

When the remote compile server receives the request, it opens a socket connection to the client and transfers the MP source and the selection file to the remote server. All data is stored in a directory, which consists of the client host name and a unique temporary directory:

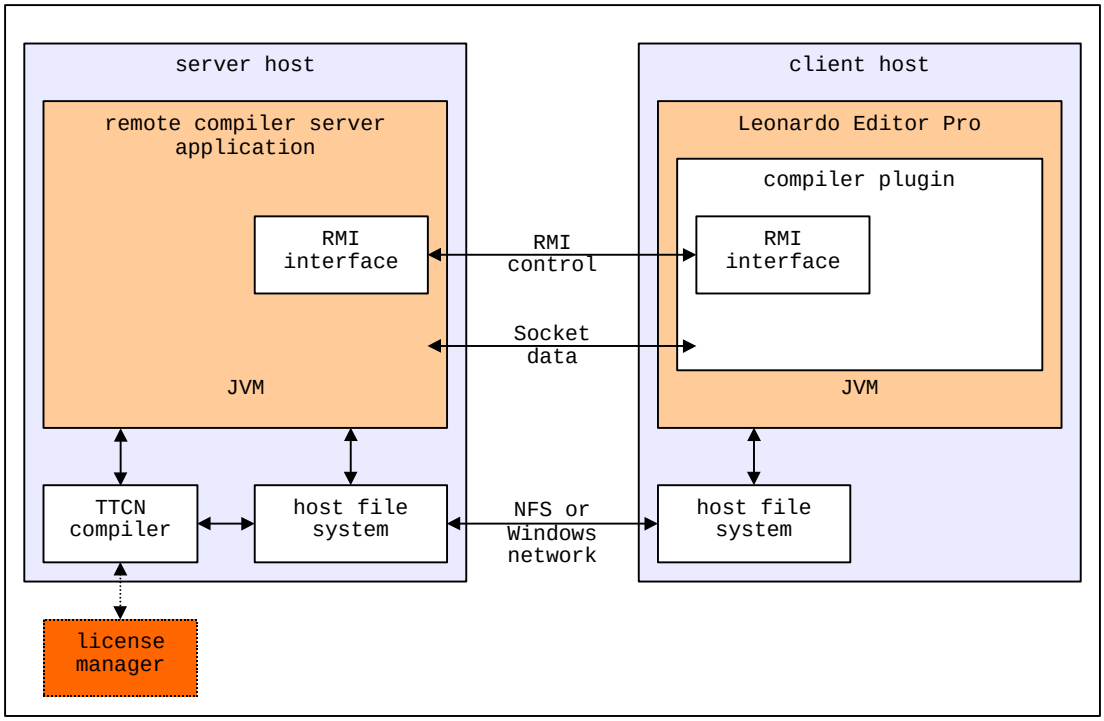
```
<server root dir>/client.host.name/TTCN12345/
```

A symbolic link is created in the `client.host.name` directory with the name `latest` that links to the most recently created temporary directory. This will make it easy to find the latest compilation results when several temporary directories are present. In the case of a compilation error, the temporary directory is removed after the compiler has exited. When the compiler has exited successfully with exit value zero, the temporary directory is not removed.

The compiler messages from STDOUT and STDERR are routed to the client and displayed there in the output dialog.

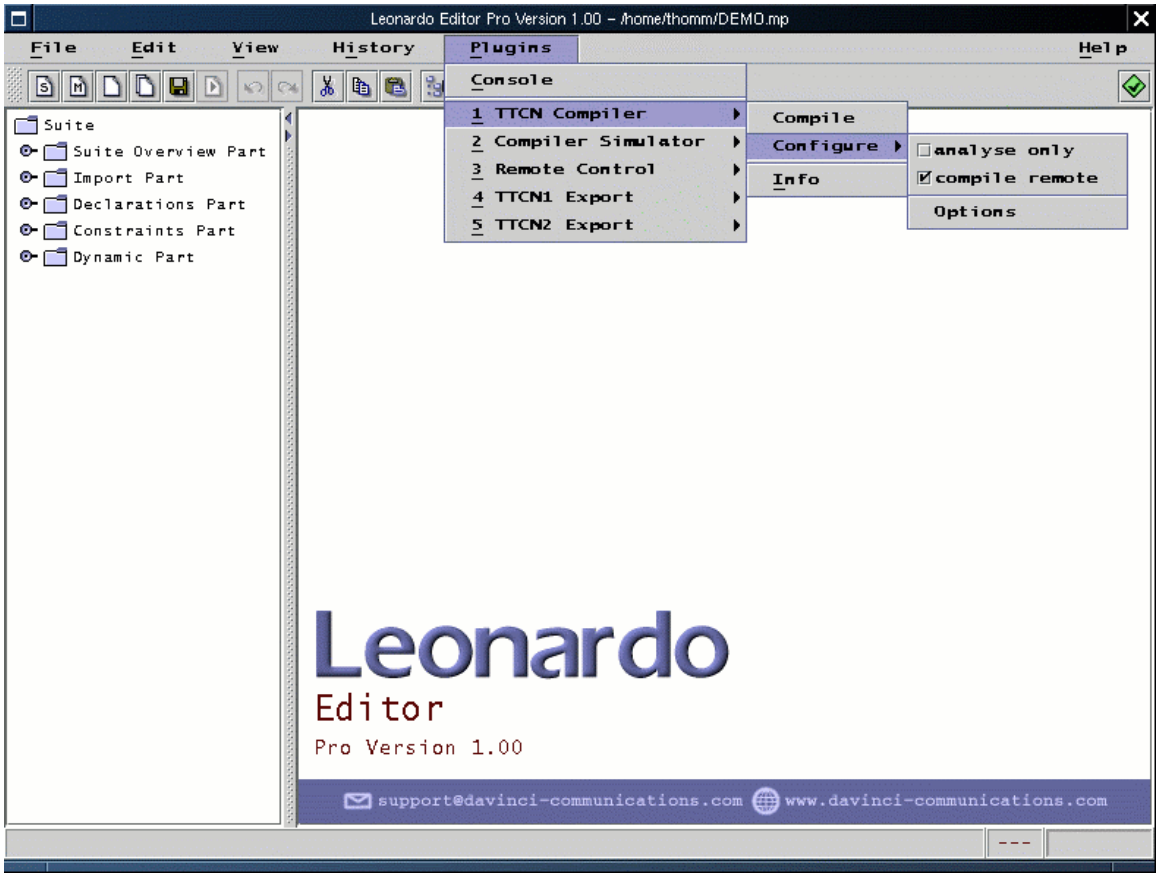
The client can get the generated files by mounting the `<server root dir>/client.host.name/` via NFS or Windows network drives into its file system. Just as with the local option, the client can cancel the compilation from the output dialog.

The following drawing shows the architecture of this solution:

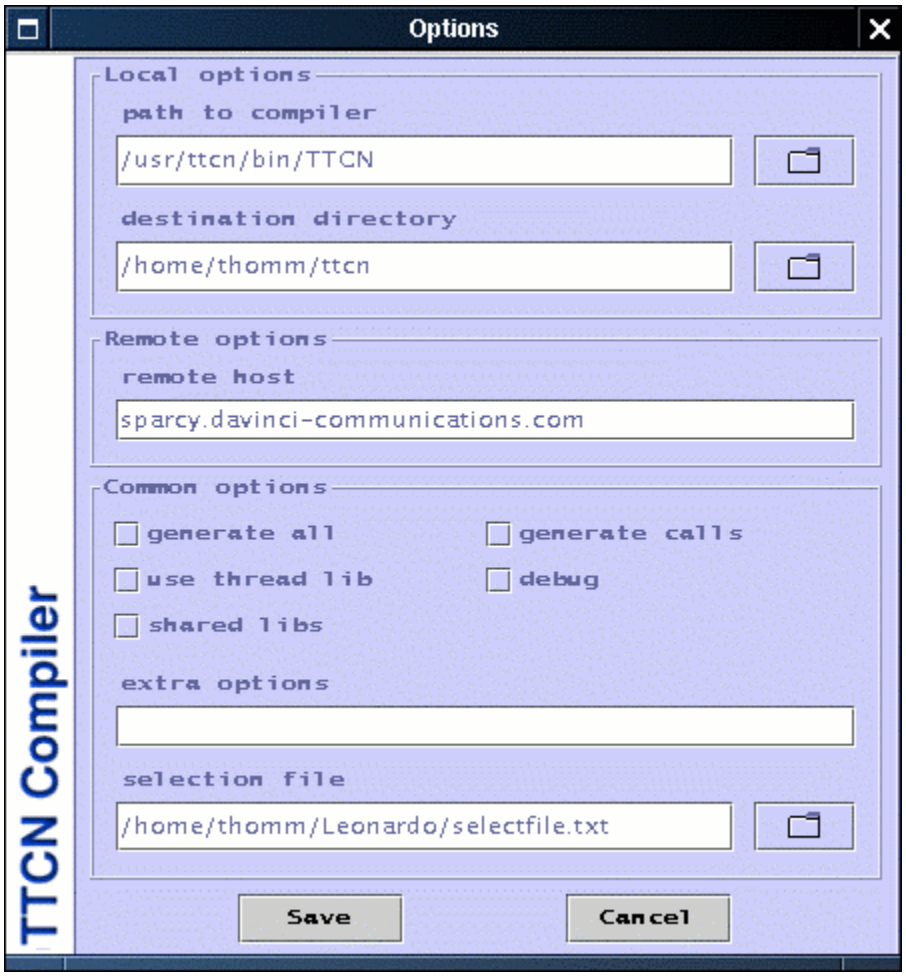


Screenshots

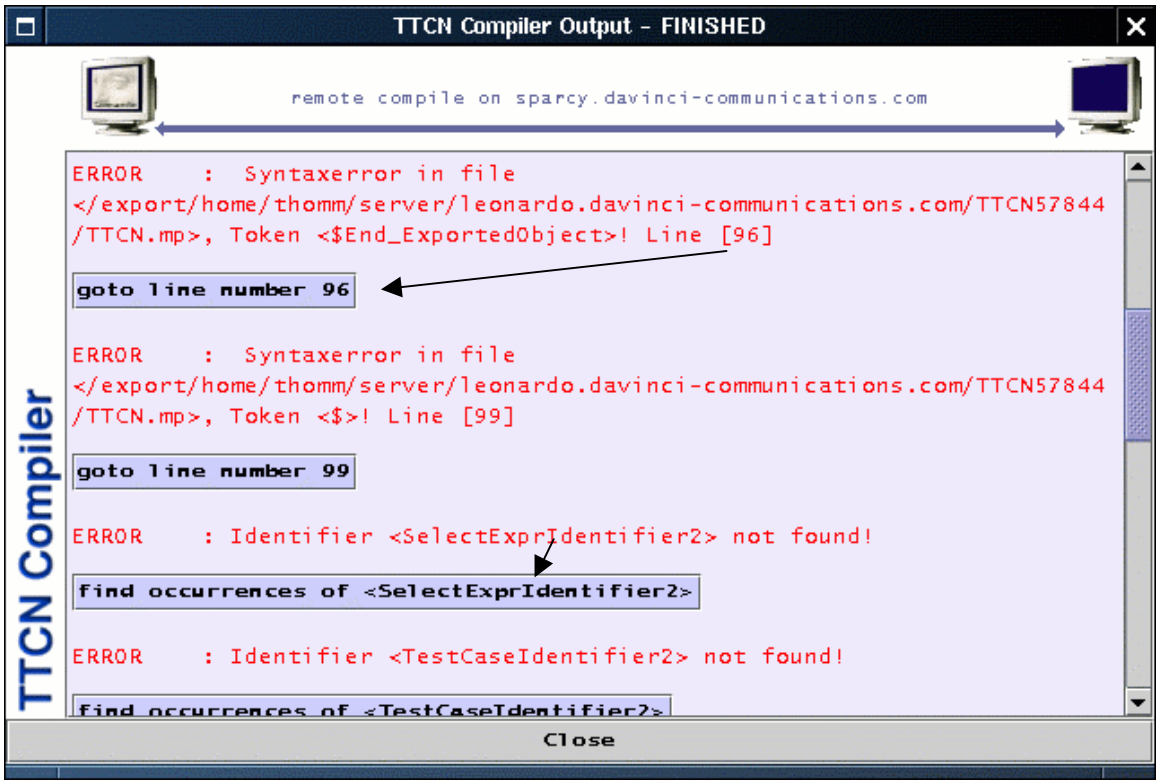
This screenshot shows the integration of the plugin into the menu structure of the editor. The plugin also provides a custom menu "`Configure`" which gives access to frequently used functions as well as an options dialog to configure the compiler.



This screenshot shows the "`Options`" dialog which can be used to configure the compiler. It is split into three sections: "`Local options`", "`Remote options`" and "`Common options`".
The local options are needed for local compilation, the remote options if you use a compiler server, and the common options are valid for both cases.



This screenshot shows the compile dialog of a remote compile session. The header area shows which host is serving the compile request. The main text area shows the standard output and standard error of the compiler. This output is already preprocessed and interactive elements have been inserted. By parsing the compiler output, the plugin recognises compiler errors and provides easy access to debugging tools.



This screenshot shows a complete debugging session and the interaction of plugin and editor.