EasyLab Technical Paper
Writing Plugins for EasyLab

Michael Geisinger
geisinge@in.tum.de

Simon Barner
barner@in.tum.de

June 2, 2010

Note
This document contains information about the EasyLab software application. Please understand that as EasyLab is still under development, the information contained herein are subject to change without further notice.

Contents

1 Introduction **3**
1.1 Plugin Interaction in Modeling Mode . . . 3
1.2 Plugin Interaction in Simulation Mode . . . 4
1.3 Plugin Interaction in Debugging Mode . . . 6

2 Requirements **6**

3 Preparing the Build Environment **7**
3.1 Installing EasyLab SDK . . . 7
3.2 Installing Microsoft Visual Studio . . . 7
3.3 Installing CMake . . . 7
3.4 Installing Boost C++ Library . . . 7
3.5 Installing and Building Qt for Application Development . . . 8
3.6 Installing Xerces-C++ . . . 9

4 Adding a new Unit Type **10**
4.1 Introduction . . . 10
4.2 Writing a Unit Type Specification . . . 10
4.2.1 Unit Type XML Definition . . . 11
4.2.2 Type Template Specification . . . 11
4.2.3 Connector Specification . . . 13
4.2.4 State Variable Specification . . . 14
4.2.5 Connector GUI Specification . . . 14
4.2.6 State Variable GUI Specification . . . 14
4.2.7 GUI Appearance Definition . . . 14
4.2.8 Plugin Reference Specification . . . 16

4.2.9 Code Generation Specification 16
4.3 Writing a Unit Simulation Plugin 19
4.4 Writing a Unit GUI plugin 23
4.4.1 Writing a Unit Dialog Plugin 23
4.4.2 Writing a Unit Visualization Plugin 29

5 Adding a new Device Type **29**
5.1 Writing a Device Type Specification 29
5.2 Writing a Device Simulation Plugin 30
5.3 Writing a Device GUI Plugin 30
5.3.1 Writing a Device Dialog Plugin 30
5.3.2 Writing a Unit Toolbar Plugin 30

6 Adding a new Uploader Plugin **30**
6.1 Writing an Uploader Specification 30
6.2 Writing an Uploader Plugin 30

7 Adding a new Debugger Plugin **30**
7.1 Writing a Debugger Specification 30
7.2 Writing a Debugger Plugin 30

8 Adding a new Service Plugin **30**
8.1 Writing a Service Specification 30
8.2 Writing a Service Plugin 30

9 Frequently Asked Questions **30**

A Localization **33**

List of Tables **33**

List of Listings **33**

1 Introduction

The EasyLab software application allows you to extend its functionality in many ways. For new functionality to be implemented, it is not only necessary to add new unit or device type specifications using XML files, but also to implement so-called *plugins*. There are five different types of plugins currently available:

1. **Simulation plugins** are needed to *simulate* the functionality of a function block or a device when EasyLab runs in *simulation mode*. Such plugins will typically try to imitate the behavior of the function block in the real hardware or the respective device.

2. **Visualization plugins** extend the EasyLab graphical user interface. Such plugins can in turn be used at different occasions: either as part of a *unit dialog*/*device dialog*, as part of a *toolbar* or as part of a *view*. They visualize the status of a unit or device and allow the user to modify the state of the respective item during *modeling time*, *simulation time* or *debugging time*.

3. **Uploader plugins** implement the process of transferring compiled programs to the target hardware. If the target system is a microcontroller, this is typically an interface to some kind of bootloader. If an operating system is running on the target, uploading the image may be a simple file transfer.

4. **Debugger plugins** provide communication with an EasyLab generated program on the target system. They enable EasyLab to inspect the current program state and also to modify it when EasyLab runs in *debugging mode*. Although currently not supported, debugger plugins should also be able to set breakpoints and provide step-wise execution of a program.

5. **Service plugins** are similar to *debugger plugins* and also provide communication with an EasyLab generated program on the target system. However, communication is not limited to development time, but the program state can be inspected and modified after the application has been deployed (hence the term "service", meaning that you can maintain the application somehow). A *service plugin* implements a certain communication protocol for a common *service API* and hence makes it possible for third-party applications to communicate with the deployed target system.

1.1 Plugin Interaction in Modeling Mode

Modeling mode is active when EasyLab is not currently simulating the execution of a program nor it is currently in debugging mode. In this mode, the user has the full ability to design the logic for the respective application. This also means that it is possible to interact with all kinds of *visualization plugins*:

- **Device visualization plugins** may be used to set up preferences that are required to connect to the respective device or that influence the exact behavior of a device. For example, if the device is connected via Ethernet and communication runs via TCP/IP, then the *device visualization plugin* may be used to specify IP address and port for the remote connection. There may also be a "connect" button available, which allows to initiate the connection at a given point in time.

- **Unit visualization plugins** may be used to set up parameters of the units in the respective synchronous data flow (*abbrev.* "SDF") graphs.

1.2 Plugin Interaction in Simulation Mode

Simulation mode is active when the user has started a simulation run in EasyLab. In this mode, the *unit/device simulation plugins* are called in a schedule that corresponds to the modeled application logic.

Figure 1 shows when the respective plugins are called in a real-world example. Imagine you want your application to control an electric motor. The motor itself is a device that is "attached" to EasyLab using a *device simulation plugin*. Furthermore, a "motor speed unit" is added to be able to control the speed of the motor from an SDF. A *unit simulation plugin* is used to implement the functionality of that unit, which interacts with the device plugin to actually set the motor's speed.

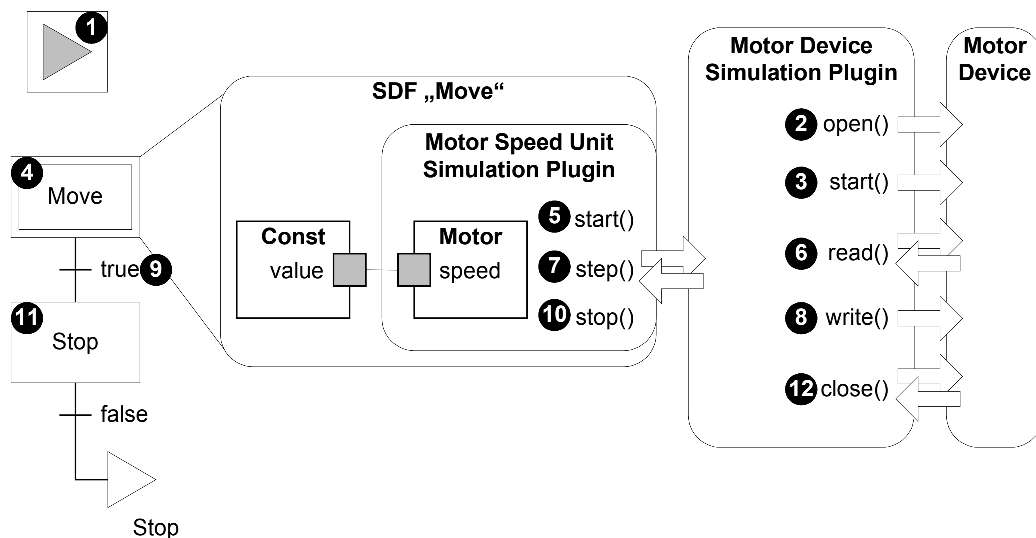


Figure 1: Interaction of application logic, device and unit simulation plugins in simulation mode; see text for description

Imagine the simulation of the program shown in the figure is to be started. The following actions will be carried out, in that exact order:

1. **Start simulation:** The user presses the "play" button to switch to *simulation mode*.

2. **Ensure device connections:** First, EasyLab verifies that all *device simulation plugins* are actually "connected". A device plugin is connected if its `open()` method has been successfully executed and no disconnect has occurred since. Note that a connection is typically established before *simulation mode* is entered, for example when the project is loaded in EasyLab or when a new device instance is added to the project. If a device is currently not connected, EasyLab calls the `open()` method to try to establish a connection.

   The device may be "disconnected" at any time. The exact scope of the disconnect is device specific. A *device simulation plugin* must detect a disconnect. When a disconnect happens, the plugin must call the `disconnected()` method on the simulation callback to signal EasyLab that the device is not ready for simulation any more.

In this step, EasyLab also calculates static schedules for all SDF models that are contained in the current project.

3. **Start devices:** The `start()` methods for *all* device instances in the project are called in an undetermined order. This gives the device simulation plugin the possibility to initialize the device with its default values or to verify that the connection is still alive.

   Note that regarding the order of execution, EasyLab only guarantees that the `start()` methods for parent devices are called before the respective methods in the children devices are called. Do not assume a certain order of execution across multiple devices!

4. **Launch application:** The main program is entered. In our example, this means that the "Move" state of the SFC gets active.

5. **Start SDF:** The `start()` methods for *all* unit instances in the currently active SDF are called in the same order they are executed later on. This only happens once per state transition and gives the *unit simulation plugins* the possibility to initializes the state of the respective unit instances. In our example, the `start()` method of the "constant unit" simulation plugin would be called, followed by a call to the "motor speed unit" simulation plugin's `start()` method.

6. **Read device state:** The `read()` methods for all *device simulation plugins* for which unit instances exist in the current SDF are executed in an undetermined order. This allows the device to capture a snapshot of the real world using its sensors and store it in the respective buffers.

   Note that regarding the order of execution, EasyLab only guarantees that the `read()` methods for parent devices are called before the respective methods in the children devices are called. Do not assume a certain order of execution across multiple devices!

   Note also that the `read()` method should return as quickly as possible. Otherwise, the snapshot might not be "atomic" enough to provide the application with an adequate image of the real world in case a large number of devices is involved.

7. **Execute units:** The *unit simulation plugins'* `step()` methods corresponding to the unit instances in the SDF are being called according to the static schedule that has been calculated in advance. The simulation plugin may use the `getDeviceCallback()` method to retrieve the context of the associated device where the variables buffered in the previous step are beinbg stored. In our example, the simulation plugin uses this interface to output a new speed value for the motor.

8. **Write device state:** The `write()` methods for all *device simulation plugins* for which unit instances exist in the current SDF are executed in an undetermined order. This allows the device to output the values written by the *unit simulation plugins* in the previous step. In this example, the new speed for the motor will be written to the real hardware.

   Note that regarding the order of execution, EasyLab only guarantees that the `write()` methods for parent devices are called before the respective methods in the children devices are called. Do not assume a certain order of execution across multiple devices!

9. **Evaluate transition condition:** As execution of the SDF has now finished, the transition condition associated to the SFC state currently being executed is evaluated. In this example, the transition condition is always true and hence the "Move" state will be left. If the transition condition would not evaluate to a true value, execution would continue at step 6.

10. **Stop SDF:** As a result of the state change, the "Move" SDF units' `stop()` functions are called. The order of calls is the reverse order in which the `start()` methods have been called.

11. **Continue execution:** Execution continues in the "Stop" state. In this example, the "Stop" state is never left.

12. **Disconnect devices:** Finally, the `close()` methods of the *device simulation plugins* are invoked for the device instances in the project. This may happen for individual device instances when they are deleted or for all device instances when the respective project or EasyLab is closed. In addition, device instances may be disconnected by the choice of the user at any point in time.

In addition, the user has the possibility to modify the application logic while in *simulation mode*. However, some operations may block until a suitable point in time is reached for them to be executed. For modifications in SDF models, this is typically the case after the simulation has completed one execution cycle of the SDF. However, in single-step mode, modifications take effect immediately.

1.3 Plugin Interaction in Debugging Mode

This section will be added later on. Please contact us if you need help.

2 Requirements

The following software components are required for authoring simulation, debugger, uploader and service plugins:

- **EasyLab SDK** (section 3.1)
- **Microsoft Visual Studio 2008**[1] (section 3.2)
- **CMake 2.6** or later (section 3.3)
- **Boost C++ Library** (section 3.4)

The following software component is needed in addition for developing visualization plugins and also for uploader plugins that require user interaction in the EasyLab GUI:

- **Qt for Application Development 4.5.1** for building necessary `*.lib` files (section 3.5)

The following software component is needed in addition for developing uploader plugins:

- **Xerces-C++ 3.0.1** (section 3.6)

[1]Microsoft Visual Studio 2005 is not guaranteed to work with future versions of the EasyLab plugin architecture. Microsoft Visual Studio 2008 Express has not yet been tested, but should work as well.

3 Preparing the Build Environment

3.1 Installing EasyLab SDK

First of all, download and install the newest version of EasyLab SDK[2] from the groupware:

1. Navigate to https://easykit.informatik.tu-muenchen.de/egroupware/.
2. Click on "MyDMS".
3. Expand "Root-Folder" → "EasyLab".
4. Expand the newest dated folder and download the installer.
5. Install EasyLab by following the instructions. Choose "Yes" when prompted to install CMake, you will need it for building the plugins.
6. When you need to specify the `EASYLAB_INSTALL_DIR` somewhere in this document, use the directory where you installed EasyLab to (e.g., `C:\Program Files\EasyLab`).

3.2 Installing Microsoft Visual Studio

Install Microsoft Visual Studio 2008 and Visual Studio 2008 Service Pack 1. We currently only maintain the build under Windows using the 2008 version, so you might experience problems when trying to build with Microsoft Visual Studio 2005. Please let us know.

Note In the future, we would also like to investigate how well this works using Microsoft Visual Studio Express 2008. However, this has not been tested yet. If you try it, please let us know the results.

3.3 Installing CMake

EasyLab already ships with a version of CMake. If you have chosen not to install CMake, please either reinstall EasyLab or install CMake 2.6.3 or newer manually:

1. Navigate to http://www.cmake.org/ and download the latest version of the installer (Resources → Download).
2. Make sure that you choose to add CMake's `bin` directory to your `%PATH%` during setup.
3. Test whether it works by opening a command shell and typing `cmake`. CMake's help output should appear. If you see a message that the command could not be found, double-check your `%PATH%` settings.

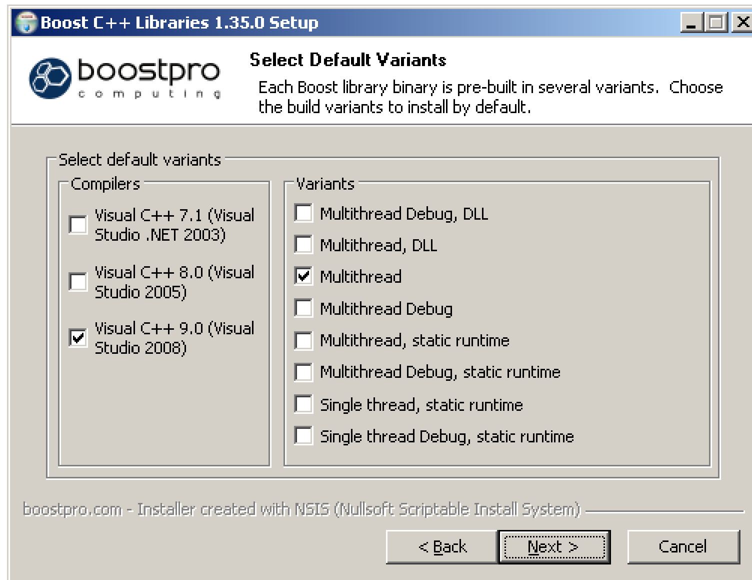
3.4 Installing Boost C++ Library

1. Download version 1.35.0 or a newer version of Boost installer from http://www.boostpro.com/download. You may have to register before you can download the distribution. Be careful not do download a version that does not match your version of Microsoft Visual Studio.

   We have not yet tested newer versions than 1.35.0. Please let us know if you experience any problems.

[2] The SDK version of EasyLab contains additional files needed for developing plugins.

2. Run the installer. When you are asked for which variants of the library you want to install, select the following (if you are using Microsoft Visual Studio 2008):



3. When you need to specify the `BOOST_INSTALL_DIR` somewhere in this document, use the directory you specified during installation of Boost (e.g., `C:\Program Files\boost\boost_1_35_0`).

3.5 Installing and Building Qt for Application Development

Note Only necessary if you want to build visualization plugins or other GUI related plugins (e.g., uploader plugins that require user interaction). Due to recent changes in Qt licensing terms, we will soon ship a development version of Qt together with EasyLab SDK in the near future. Until then, please follow the instructions below to configure your build environment.

Note The following instructions apply to Windows only.

1. Navigate to http://get.qtsoftware.com/qt/source/

2. Download `qt-win-opensource-src-4.5.1.zip`. It is not guaranteed that never versions will work. Older versions will probably *not* work.

3. Extract the archive to a directory that has *no spaces* in its name. For example, extract it to `C:\Qt`, *not* to `C:\Documents and Settings\`*`User`*`\Desktop` (the suffix `qt-win-opensource-src-4.5.1` will be appended automatically). Note that you will later have to reference the files in that directory from your plugin build scripts, so you should choose a permanent location.

4. Open a command shell and navigate to the directory where you have extracted the archive to (the following instructions assume the suggested location):
```
> C:
> cd C:\Qt\qt-win-opensource-src-4.5.1
```

5. Run the `configure` application. You may have to supply a platform name, like so:
   `> configure -platform win32-msvc2008`

   If you encounter problems, follow these instructions:

   - If you receive a message that indicates that `nmake` could not be found, make sure that the following directories are in your `%PATH%` (adapting them accordingly):
     `C:\Program Files\Microsoft Visual Studio 9.0\VC\bin`
     `C:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE`
   - If you receive a message that indicates that standard include files could not be found, make sure that the `%INCLUDE%` environment variable exists and contains the following directory (adapting it accordingly):
     `C:\Program Files\Microsoft Visual Studio 9.0\VC\include`
   - If you receive a message that linking failed, make sure that the `%LIB%` environment variable exists and contains the following directory (adapting it accordingly):
     `C:\Program Files\Microsoft Visual Studio 9.0\VC\lib`

6. After the message "Qt is now configured for building" appears, you may start the build by typing:
   `> nmake`
   This will take some time (typically a few hours). If you encounter problems during build, it may help to simply restart it. Compilation will continue from where it failed. You may also interrupt the build as soon as both the files `QtCore4.lib` and `QtGui4.lib` appear in the `C:\Qt\qt-win-opensource-src-4.5.1\lib` directory, because these are the only files that we need. This should be the case after about one hour of building the project.

7. After the build, navigate to the `C:\Qt\qt-win-opensource-src-4.5.1\lib` directory, where you will find `QtCore4.lib` and `QtGui4.lib`. These files are needed to build visualization plugins for EasyLab. Follow the instructions in section 4.4.1 or 5.3.1 to actually create a new visualization plugin using the Qt libraries.

8. When you need to specify the `QT_INSTALL_DIR` somewhere in this document, use the directory `C:\Qt\qt-win-opensource-src-4.5.1`.

If you need to start from scratch for some reason, type
`> nmake confclean`
and then reconfigure using
`> configure -platform win32-msvc2008`

3.6 Installing Xerces-C++

Note Only necessary if you want to build uploader plugins.
Xerces-C++ is a library for parsing and writing XML files.

1. Navigate to `http://xerces.apache.org/xerces-c/download.cgi` and download XercesC++ 3.0.1 or newer suitable for your version of Microsoft Visual Studio (vc9 means Visual Studio 2008, vc8 means Visual Studio 2005). The file should be named `xerces-c-3.0.1-x86-windows-vc-9.0.zip` or similar.

2. Extract the archive to a directory of your choice (for example to `C:/Program Files/XercesC`).

3. You should now have a directory like `C:/Program Files/XercesC/xerces-c-3.0.1-x86-windows-vc-9.0` When you need to specify the `XERCESC_INSTALL_DIR` somewhere in this document, use that directory.

4 Adding a new Unit Type

4.1 Introduction

Units (also called *function blocks*) represent the functional primitives in an application modeled in EasyLab. A unit can be realized in software only (so-called *generic units*) or it can provide access to the functionality of a hardware device.

Units have the following interface:

- One or more **input connectors** for receiving data
- An arbitrary amount of **state variables** for storing data
- One or more **output connectors** for emitting data

All these connectors and variables are typed (i.e., their data has a certain data type).

In this section, we will give an example on how to add a new unit type to EasyLab. The new unit type will have the following interfaces and state variables:

- One input connector named `input`
- One state variable named `value`
- One output connector named `output`

The purpose of the unit is to receive one value on the input port and memorize this value until the unit is executed the next time. The unit buffers a value for one discrete time step. The name of this unit is `Hold Element`. Note that since the hold element should be able to buffer any data type available (e.g., integer numbers, floating-point numbers, Boolean values, vectors, images). The source code presented in this document is also available in EasyLab SDK in the directory `units/Math/HoldElement.unit`. When starting unit plugin development, it is recommended to first try to set up the build environment correctly and get the example code to work and only then start development of own unit plugins.

4.2 Writing a Unit Type Specification

New units can be declared by providing a specification in XML format which defines the following main attributes:

1. Local **type templates** that define which data types may be used on the unit's connectors
2. **Interface definitions** (*input connectors* and *output connectors*):
   - Connector names
   - Connector types or template type identifiers
   - Connector layout (for EasyLab GUI)
   - Connector descriptions

3. **State variable definitions** (used to visualize the internal state of a unit during debugging)
4. **Code fragments** or **code references** for code generation (so-called *code templates*)
5. Optionally **localization** information (translations into different languages)

4.2.1 Unit Type XML Definition

New unit types can be added by creating a new folder somewhere below the `%EASYLAB%/units` directory. This is the default folder that gets parsed during launch of EasyLab. The name of the folder for the new unit has to contain the suffix `.unit` to allow EasyLab to recognize it correctly. Note that the directory for the new unit may be located anywhere below the `units` directory. You should choose some kind of structure for new units, or re-use the existing structure. Also note that it is recommended to not have spaces in any such folder names. In this example, we create a folder named `HoldElement.unit` below the `%EASYLAB%/units/tutorial` directory.

Inside the newly created directory, create a file called `unit.xml`. This is the file that will contain your new unit type specification. The syntax in the file must follow a certain schema, which can be found in the `%EASYLAB%/schema/unit.xsd` file (for those who know about how to interpret XML schema definitions). If EasyLab fails to load a unit file, it will place a message in its log window (there will be *no* message box being displayed!). The message typically indicates in which line the problem is located, so you should check the log window if you experience problems.

Listing 1 shows the basic structure of a unit type XML file. Every unit type needs to have a *unique* identifier (see line 17), which is by convention the name of the company or institution which developed the unit type plus a descriptive name of the actual functionality of the unit. We will use the name `My Company Hold Element`. Although the name may be an arbitrary string according to the XML format specification, it is recommended to keep it simple yet unique. Also avoid using fancy characters in the unit name and use English terms where applicable. The name can be later translated into different languages by using different means.

The different properties of a unit type are explained more in detail in the following sections and illustrated on the example of the hold element. Note that due to the structure of the `unit.xml` file, it is also possible to define multiple unit types in a single file (by specifying multiple `<unit:unittype>` nodes). This may be meaningful if you want to model unit types whose functionality is closely related.

4.2.2 Type Template Specification

This section allows the specification of local type templates. Formally, a type template is a list of (primitive or complex) data type names and a unique identifier that is assigned to the list. The unique identifier can then be used to specify which data types are valid for each connector and state variable of the unit. The user can then select the precise type for the respective interface or variable during *modeling time*. Note that if a connector or state variable's type is fixed, you do *not* have to define a type template for it.

Multiple type templates can be specified in this section of the XML file, however each type template must have its own unique identifier assigned. Note that names do not have to be unique across different unit type specifications nor across multiple unit type XML files. Type template names are by convention upper-case letters (e.g., `T`, `S`, `R`).

Listing 1: Unit type definition in XML, `unit.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<unit:unit
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://.../schema/unit unit.xsd"
  xmlns:plugin="http://.../schema/plugin"
  xmlns:unit="http://.../schema/unit"     <!-- URIs truncated,     -->
  xmlns:var="http://.../schema/variable"  <!-- see a real unit.xml -->
  xmlns:tmpl="http://.../schema/template" <!-- file for full URIs. -->
  xmlns:tr="http://.../schema/translation">

  <unit:translations>
    <!-- Localization information (see appendix A) -->
  </unit:translations>

  <unit:unittypes>

    <unit:unittype unit:id="REC GmbH Addition">
      <unit:typeTemplates>
        <!-- Type templates (see section 4.2.2) -->
      </unit:typeTemplates>
      <unit:connectors>
        <!-- Connectors definitions (see section 4.2.3) -->
      </unit:connectors>
      <unit:state>
        <!-- State variable definitions (see section 4.2.4) -->
      </unit:state>

      <unit:guiinfo>
        <unit:connectors>
          <!-- Connector GUI definitions (see section 4.2.5) -->
        </unit:connectors>
        <unit:state>
          <!-- State variable GUI definitions (see section 4.2.6) -->
        </unit:state>
        <unit:description>
          <!-- GUI appearance definition (see section 4.2.7) -->
        </unit:description>
      </unit:guiinfo>

      <unit:plugins>
        <!-- Plugin references (see section 4.2.8) -->
      </unit:plugins>

      <unit:codegen>
        <!-- Code generation specification (see section 4.2.9) -->
      </unit:codegen>
    </unit:unittype>

  </unit:unittypes>
</unit:unit>
```

In our example, the hold element unit type should be suitable for a large set of data types. This means we have to define a type template that includes these data types. But which data types are actually available? This question in answered by having a look at the file `%EASYLAB%/config/types.xml`. This file defines the various *data types* and *data type groups* that are available. Note that the data type identifiers defined herein may not be the ones used for the actual code generation process, since different compilers support different basic data types. Instead, these data types are the common ones that are use for modelling in EasyLab. Data type groups are used to be able to easily reference groups of data types as a whole. For example, the group `group_unsigned` contains all unsigned integer data types.

Listing 2 shows how the type template definition for the hold element may look like[3]. Due to the definition of the `group_number` data type group, this definition is equivalent to the one shown in listing 3 (note that we manually added the `bool` data type in both cases, which is of course not part of the `group_number` data type group). Note that the name for the new type template is `T`. The advantages of the first approach are that it is less error-prone and much easier to read.

Listing 2: Unit type definition in XML, type template specification (compact form)

```
1      <!-- ... -->
2      <unit:typeTemplates>
3        <unit:typeTemplate var:typename="T">
4          <var:specializationGroup var:group="group_number"/>
5          <var:specialization var:type="bool"/>
6          <var:default var:type="int16"/>
7        </unit:typeTemplate>
8      </unit:typeTemplates>
9      <!-- ... -->
```

4.2.3 Connector Specification

The connector specification defines the internal names and data types for input and output interfaces of the unit type. In addition, a default values has to be specified for each interface. Listing 4 illustrates the implementation for the hold element example. Note that both connectors are of type `T`, which is the template type defined in section 4.2.2. If a connector should have a particular type, just specify the type name according to `%EASYLAB%/config/types.xml` in the `var:type` attribute instead (e.g., `bool`, `int16`, `uint64`).

The fact that both connectors share the template type `T` also implicitly means that if the data type of one connector is changed, the type of the other connector is automatically adapted. In turn, this also means that if one of the connectors is attached to an edge, neither the data type of the `input` nor the one of the `output` connector can be modified (at least not without changing the data type of the connector on the opposite side of the edge as well).

As stated above, a default value has to be specified for each connector. If type templates are used, a default value has to be specified for each possible template type. In the case of

[3] In section 4.1, we said that the hold element should be able to buffer any data type available. For this purpose, we could use the `group_all` data type group. However, this group should be used with care, because a unit author can never know which fancy new data types a user defines, which may then not be supported by the unit.

Listing 3: Unit type definition in XML, type template specification (explicit form)

```
<!-- ... -->
<unit:typeTemplates>
  <unit:typeTemplate var:typename="T">
    <var:specialization var:type="int8"/>     <!-- Signed -->
    <var:specialization var:type="int16"/>
    <var:specialization var:type="int32"/>
    <var:specialization var:type="int64"/>
    <var:specialization var:type="uint8"/>    <!-- Unsigned -->
    <var:specialization var:type="uint16"/>
    <var:specialization var:type="uint32"/>
    <var:specialization var:type="uint64"/>
    <var:specialization var:type="float"/>    <!-- Decimal -->
    <var:specialization var:type="double"/>
    <var:specialization var:type="fixed"/>
    <var:specialization var:type="bool"/>     <!-- Boolean -->
    <var:default var:type="int16"/>
  </unit:typeTemplate>
</unit:typeTemplates>
<!-- ... -->
```

the hold element, we use a value of zero for all data types in the `group_number` group and a value of `false` for the `bool` data type.

4.2.4 State Variable Specification

This section specifies which state variables the unit type has and what their internal name and data type is. Listing 5 shows the realization for the hold element unit type. Note that just like connectors, state variables can use template type, in this case `T`. Instead of using a template type, we could also have specified a particular data type according to `%EASYLAB%/config/types.xml`.

Using template type `T` here implicitely means that the data type of the two connectors of the unit (which use the same template type) must always be set to the same data type than the state variable has. Default value specification for a state variable works exactly as for input/output connectors.

4.2.5 Connector GUI Specification

This section will be added later on. Please contact us if you need help.

4.2.6 State Variable GUI Specification

This section will be added later on. Please contact us if you need help.

4.2.7 GUI Appearance Definition

This section will be added later on. Please contact us if you need help.

Listing 4: Unit type definition in XML, connector specification

```
<!-- ... -->
<unit:connectors>
  <unit:in>
    <unit:connector var:type="T" var:varname="input">
      <var:default var:val="0">
        <var:specializationGroup var:group="group_number"/>
      </var:default>
      <var:default var:val="false">
        <var:specialization var:type="bool"/>
      </var:default>
    </unit:connector>
  </unit:in>
  <unit:out>
    <unit:connector var:type="T" var:varname="output">
      <var:default var:val="0">
        <var:specializationGroup var:group="group_number"/>
      </var:default>
      <var:default var:val="false">
        <var:specialization var:type="bool"/>
      </var:default>
    </unit:connector>
  </unit:out>
</unit:connectors>
<!-- ... -->
```

Listing 5: Unit type definition in XML, state variable specification

```
<!-- ... -->
<unit:state>
  <unit:statevar var:type="T" var:varname="value">
    <var:default var:val="0">
      <var:specializationGroup var:group="group_number"/>
    </var:default>
    <var:default var:val="false">
      <var:specialization var:type="bool"/>
    </var:default>
    <var:access>read-only</var:access>
  </unit:statevar>
</unit:state>
<!-- ... -->
```

4.2.8 Plugin Reference Specification

Plugin references specify the simulation and – optionally – GUI plugins that are available for a unit type. Note that multiple plugins for a specific unit type can also be combined into a single library file.

For a detailed description of simulation and GUI plugins, see sections 4.3 and 4.4, respectively. Listing 6 shows an excerpt from a unit XML file specifying one simulation and two GUI plugins (a *dialog* and a *schematic plugin*). Both GUI plugins are located in the same shared library file (named `unit_holdelement_gui.dll` under Windows and `unit_holdelement_gui.so` under UNIX/Linux).

We will show how to implement suitable simulation and GUI plugins in sections 4.3 and 4.4, respectively.

Listing 6: Unit type definition in XML, plugin reference specification

```
1  <!-- ... -->
2  <unit:plugins>
3    <unit:plugin plugin:type="simulation"
4      plugin:library="unit_holdelement_simulation"/>
5    <unit:plugin plugin:type="dialog"
6      plugin:library="unit_holdelement_gui"/>
7    <unit:plugin plugin:type="schematic"
8      plugin:library="unit_holdelement_gui"/>
9  </unit:plugins>
10 <!-- ... -->
```

4.2.9 Code Generation Specification

This section defines multiple code template segments that are used to implement the functionality of the respective unit type when generating code. The basic structure is shown in listing 7. Note that all elements below the `<unit:codegen>` node are optional, i.e., you can skip any subset of `<unit:global>`, `<unit:local>`, `<unit:start>`, `<unit:step>` and `<unit:stop>`.

For the hold element, the code template list for the `<unit:step>` element is structured as shown in listing 8.

The `<tmpl:toolchain>` element is required, because we may need to generate different code for different compiler toolchains. If a code template does not work with a specific compiler, the *emergency* solution is to add another `<tmpl:toolchain>` node and specify a different code template. However, we strongly discourage from using this facility, because it leads to code inconsistency. Write compatible code instead, or use functions from the EasyLab runtime library to access low-level functionality.

As we have specified the hold element to be a unit type using type templates, we also must specify for which combination of data types selected for `T` the respective code segments are valid. In this case, only primitive data types have to be supported, and all of them support the assignment operator. Hence, we only need *one* code template in this case. If this would not be the case, we had to specify multiple `<tmpl:template>` nodes. If multiple type templates were used, we had to specify code templates for all possible combinations of them (using multiple `<tmpl:typeTemplate>` elements before the `<tmpl:content>` node).

The code template itself is quite simple in this case: set the output connector's value to the current state, then set the state variable's value to the current input connector

Listing 7: Unit type definition in XML, code generation specification

```
1      <!-- ... -->
2      <unit:codegen>
3        <unit:global>
4          <!-- Code template list for global scope -->
5        </unit:global>
6        <unit:local>
7          <!-- Code template list for local scope -->
8        </unit:local>
9        <unit:start>
10         <!-- Code template list for initialization -->
11       </unit:start>
12       <unit:step>
13         <!-- Code template list for execution -->
14       </unit:step>
15       <unit:stop
16         <!-- Code template list for finalization -->
17       </unit:stop>
18     </unit:codegen>
19     <!-- ... -->
```

Listing 8: Unit type definition in XML, code template

```
1        <!-- ... -->
2        <unit:step>
3          <tmpl:toolchain tmpl:name="default">
4            <tmpl:template>
5              <tmpl:typeTemplate tmpl:name="T">
6                <tmpl:specializationGroup tmpl:group="group_number"/>
7                <tmpl:specialization tmpl:type="bool"/>
8              </tmpl:typeTemplate>
9              <tmpl:content>
10               <tmpl:code>
11 <![CDATA[
12   *output = state->value;
13   state->value = input;
14 ]]>
15               </tmpl:code>
16             </tmpl:content>
17           </tmpl:template>
18         </tmpl:toolchain>
19       </unit:step>
20       <!-- ... -->
```

value. Input variables can be directly accessed, output variables have to be dereferenced (`*` operator) and state variables can be accessed using the `state` structure. Note that the reason that the assignments in lines 12 and 13 works for all data types is that `input`, `output` and `value` always have the *same* data type (which is guaranteed by specifying the same template type `T` for all of them).

Alternatively to specifying the code for the hold element within the XML file, we could also have referred a code template from an external file. Listing 9 shows how this achieved. Listing 10 shows the respective file. Note that the file contains special comments that are used to mark which code segments should be used.

Listing 9: Unit type definition in XML, code reference

```
1  <!-- ... -->
2  <tmpl:content>
3    <tmpl:codeRef tmpl:file="code/holdelement.c"
4      tmpl:begin="beginStep" tmpl:end="endStep"/>
5  </tmpl:content>
6  <!-- ... -->
```

Listing 10: External code template, `code/holdelement.c`

```
1  /* ... */
2
3  /* @beginStep */
4  *output = state->value;
5  state->value = input;
6  /* @endStep */
7
8  /* ... */
```

Instead of using the attributes `tmpl:begin` and `tmpl:end` to reference the section in the external file to extract, one can also use a single attribute named `tmpl:section`. Assuming that this attribute is set to `stepSection`, listing 11 shows how to achieve same effect than in the scenario above. Note the trailing `@` in line 6.

Listing 11: External code template, `code/holdelement.c` (alternative solution)

```
1  /* ... */
2
3  /* @stepSection */
4  *output = state->value;
5  state->value = input;
6  /* stepSection@ */
7
8  /* ... */
```

It is also important to know that not all types of connectors or variables can be accessed in all code template segments, as summarized in table 1.

Code template segment	Availability		
	Inputs	State variables	Outputs
`<unit:global>`	no	no	no
`<unit:local>`	no	no	no
`<unit:start>`	no	yes	no
`<unit:step>`	yes	yes	yes
`<unit:stop>`	no	yes	no

Table 1: Availability of connectors and state variables in code templates

4.3 Writing a Unit Simulation Plugin

Unit simulation plugins implement the behavior of a unit during *simulation time*. They mimic the function of the respective unit instance in the real software application by reading/writing the input, output and/or state variables of the respective unit. For example, a counter unit's simulation plugin may increase the counter value each time an execution step is performed.

Overview The simulation plugin implements a set of functions that are invoked by EasyLab at the appropriate points in time. These functions are all part of a C++ class, (i.e., they are methods). Table 2 gives an overview of all methods that can be implemented in this context. Table 3 shows the names of the methods that can be used to implement the actual behavior of the simulation plugin.

Method	Meaning	Mandatory?
`start()`	Called when a unit's SDF is being entered	no
`step()`	Called when a unit is executed	yes
`stop()`	Called when a unit's SDF is left	no

Table 2: Overview of callback methods in a unit simulation plugin

Method	Meaning
`readInput()`	Retrieves the value of an input port
`writeInput()`	Writes the value of an input port
`readStateVariable()`	Retrieves the value of a state variable
`writeStateVariable()`	Writes the value of a state variable
`readOutput()`	Reads the value of an output port
`writeOutput()`	Writes the value of an output port

Table 3: Overview of data access methods in a unit simulation plugin

Instructions To write a new unit simulation plugin, follow these steps:

1. Below the respective unit directory, create a folder named `plugin/simulation`. In case of the hold element function block, we create the directory `HoldElement.unit/plugin/simulation`. Then, add your source and header files to that directory. By convention, at least three files are required for a simulation plugin:

- A shared library entry point definition file, typically called `main.cpp` (listing 12).
- A plugin declaration file, typically called *FunctionBlockName*`.h`, where *FunctionBlockName* is the unit type name without spaces, each word starting with an uppercase letter (e.g., `HoldElement.h`; listing 13).
- A plugin implementation file, typically called *FunctionBlockName*`.cpp` (listing 14). Pass either `plugin::simulation::Deterministic` or `plugin::simulation::NonDeterministic` to the base class constructor, depending on whether your unit produces the same output when it receives the same input or not. This is not true for sensor and function blocks with dynamic internal state. If your function block is deterministic, certain optimizations may be applied.

Note: The first argument to the `ADD_UNIT()` macro in `main.cpp` (listing 12) must exactly match the unit type identifier specified in `unit.xml` (see section 4.2.1).

Note: In listings 13 and 14, we define the methods `start()` and `stop()`, but leave their body empty. This is not necessary. If you do not need these functions, you can simply not define them. Note however that the `step()` function is pure virtual and must be implemented in any case.

Listing 12: Unit simulation plugin, `plugin/simulation/main.cpp`

```
1 #include "plugin/simulation/Interface.h"
2
3 #include "HoldElement.h"
4
5 BEGIN_SIMULATION_INTERFACE("My Company", "Hold Element")
6   BEGIN_UNITS
7     ADD_UNIT("My Company Hold Element", HoldElement)
8   END_UNITS
9 END_SIMULATION_INTERFACE
```

2. Add a file called `CMakeLists.txt`[4] with the content shown in listing 15.

   Note: The plugin name specified in line 12 of listing 15 is also used for the name of the shared library that contains the plugin. Hence, it must correspond to the filename specified in the plugins section of `unit.xml`. See section 4.2.8 for more information.

3. Then, in the directory where `CMakeLists.txt` is located, create a folder named `build`. Open a command shell, navigate to that folder and execute CMake as shown below:

```
> C:
> cd "C:\Program Files\EasyLab\units\tutorial\HoldElement.unit"
> cd plugin\simulation\build
> cmake -G "Visual Studio 9 2008" ..
```

   Note the trailing “..” in the last command. For a list of valid values to the `-G` argument, run `cmake` without any arguments and look in the “Generators” section of the output. You should see output similar to the one shown in listing 16.

[4] Please use exactly this case for the filename, because it is a special name that is recognized by CMake.

Listing 13: Unit simulation plugin, `plugin/simulation/HoldElement.h`

```
#ifndef _HOLDELEMENT_H_
#define _HOLDELEMENT_H_

#include "plugin/simulation/Unit.h"

class HoldElement : public plugin::simulation::Unit
{
public:
  HoldElement (plugin::simulation::UnitDelegate& del);

  virtual void start (void);
  virtual void step (void);
  virtual void stop (void);
};

#endif // #ifndef _HOLDELEMENT_H_
```

Listing 14: Unit simulation plugin, `plugin/simulation/HoldElement.cpp`

```
#include "HoldElement.h"

HoldElement::HoldElement (plugin::simulation::UnitDelegate& del)
: plugin::simulation::Unit(del, plugin::simulation::NonDeterministic)
{
}

void HoldElement::start (void) {
  // Nothing to do
}

void HoldElement::step (void) {
  writeOutput("output", readStateVariable("value"));
  writeStateVariable("value", readInput("input"));
}

void HoldElement::stop (void) {
  // Nothing to do
}
```

Listing 15: Unit simulation plugin, `plugin/simulation/CMakeLists.txt`

```
cmake_minimum_required(VERSION 2.6)

# Only use release configurations
set(
  CMAKE_CONFIGURATION_TYPES "Release;RelWithDebInfo;MinSizeRel"
  CACHE TYPE INTERNAL FORCE
)

# Replace "holdelement" with your unit type name
set(
  UNITPLUGIN_SIMULATION_NAME
  unit_holdelement_simulation
)
project(${UNITPLUGIN_SIMULATION_NAME})

file(GLOB UNITPLUGIN_SIMULATION_SRCS *.cpp)
file(GLOB UNITPLUGIN_SIMULATION_HEADER *.h)

# Adapt paths as needed
set(EASYLAB_INSTALL_DIR "C:/Program Files/EasyLab")
set(BOOST_INSTALL_DIR "C:/Program Files/boost/1_35_0")

include(
  # Add as many ../ as are necessary to refer to the respective file
  # (depending on where your unit is located in the "units" directory)
  "${CMAKE_CURRENT_SOURCE_DIR}/../../unitplugin_simulation.cmake"
)
```

Listing 16: Unit simulation plugin, CMake output

```
-- Check for working C compiler: cl.exe
-- Check for working C compiler: cl.exe -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: cl.exe
-- Check for working CXX compiler: cl.exe -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to:
    C:/.../units/tutorial/HoldElement.unit/plugin/simulation/build
```

If the output is that `cmake` could not be found, ensure that CMake's `bin` directory is in the `%PATH%` and relaunch the command shell. If an error occurs during processing, make sure that `CMakeLists.txt` is set up correctly and all paths match.

4. Now we can load the generated Solution in Visual Studio. Navigate to the `build` directory and open `unit_holdelement_simulation.sln`. Switch the build profile to "Release". If everything went OK, you should now be able to compile your project.

   If the build fails with missing include files or linker library files, please double-check the paths specified in `CMakeLists.txt`.

   Note: If you corrected an error, you do normally *not* need to run CMake again, but instead CMake should auto-detect that you modified `CMakeLists.txt` and start a new configuration run by itself. In this case you should see the text "CMake is re-running because build system is out-of-date" in the Visual Studio console and Visual Studio may prompt you to reload the solution or specific projects, because they have been changed externally. You should confirm the reload.

5. After the build is complete, we have to copy the compiled shared library to the unit directory. To do this, simply execute the `INSTALL` target in Visual Studio (this will only work if the DLL has been built in "Release" configuration). You have to execute this deployment step after every change to the plugin!

6. If you have access to the EasyLab Subversion repository, you might want to commit your work. Please respect the following rules when adding files to the repository:
   - In the directory `plugin/simulation`, add just the source and header files as well as the file `CMakeLists.txt`. In our example this would be `main.cpp`, `HoldElement.h`, `HoldElement.cpp` and `CMakeLists.txt`.
   - *Never* add the `build` directory to version control.
   - Do *not* add compiled versions of the plugin (i.e., `*.dll` files) to version control.

4.4 Writing a Unit GUI plugin

4.4.1 Writing a Unit Dialog Plugin

Unit dialog plugins present the user the current state of a function block and allow the user to modify its state by entering new values for input, output or state variables. The name "dialog plugin" is due to the fact that such plugins own a specific area in the EasyLab GUI that has the look and feel of a dialog box. The area can also be moved around and resized to fit the needs of the user.

Overview The dialog plugin implements a set of functions that are invoked by EasyLab at the appropriate points in time. These functions are all part of a C++ class, (i.e., they are methods). Table 4 gives an overview of all methods that can be implemented in this context. Table 5 shows the names of the methods that can be used to implement the actual behavior of the dialog plugin.

Updating the dialog The `update()` method is called asynchronously in certain time intervals (every 40 ms by default, that is a framerate of 25 fps). When the method is called, a `UnitHistoryBundle` object is passed to the plugin. This object contains all changes to the variables since the last call to `update()` the dialog plugin has registered for. Registering

Method	Meaning	Mandatory?
`update()`	Called when the dialog fields should be updated	no
`translate()`	Called when the GUI language has changed	no

Table 4: Overview of callback methods in a unit dialog plugin

Method	Meaning
`readInput()`	Retrieves the value of an input port
`writeInput()`	Writes the value of an input port
`getInputProperty()`	Retrieves an input connector's properties
`setInputProperty()`	Sets an input connector's properties
`readStateVariable()`	Retrieves the value of a state variable
`writeStateVariable()`	Writes the value of a state variable
`getStateVariableProperty()`	Retrieves a state variable's properties
`setStateVariableProperty()`	Sets a state variable's properties
`readOutput()`	Retrieves the value of an output port
`writeOutput()`	Writes the value of an output port
`getOutputProperty()`	Retrieves an output connector's properties
`setOutputProperty()`	Sets an output connector's properties

Table 5: Overview of data access methods in a unit dialog plugin

for a specific interface or state variable is possible with the methods of the `UnitDelegate` class. An instance of this class is passed in the constructor of the dialog plugin, so a typical registration looks like the one in listing 19, line 17.

History size You might now ask what the second parameter to these registration functions is used for. This is the so-called *history size*. It determines how many values are delivered at max to the `update()` function if it has not been called for some time. To explain this in detail, it is important to know that the `update()` method is called asynchronously on a regular basis using a timer. In this context, it does not matter in which speed the simulation runs, because the update is performed completely asynchronous. It may be the case that the simulation runs very fast in comparison to the update timer. In this case, multiple values will be generated for a single connector or state variable between two calls to `update()`. The history size determines how many values of the respective connector or state variable are conserved and finally sent to the `update()` method using the `UnitHistoryBundle` parameter. If more values are generated than history size specifies, older values are *potentially* discarded. Note that this does not guarantee that values are dropped. This is due to reasons of efficiency.

Now how dou you use this feature? In most cases, history size should be set to `1`. This is due to the fact that most unit dialogs provide a facility to display and edit only the current value of an interface or a state variable. For example, the hold element unit wants to display the last input value received. However, other units might have different requirements in this context. Imagine a “digital oscilloscpoe” unit dialog, which tries to display the last 100 values and interpolates between them. In this case, history size must be set to 100 to ensure that even in extreme cases (when the clock on the simulation and the update timer run at totally different frequencies), all values are eventually displayed in the scope (i.e., no values are potentially dropped).

A plugin author can access the new value(s) for a state variable named `"value"` in a call to `update()` as shown in listing 19, line 23. Line 27 shows how to access the last

element in the history buffer. The VariableHistoryBuffer object also offers the functions `getInputHistory()` and `getInputHistory()` for retrieving updates for input and output connectors, respectively.

Localization If there is any text displayed in a unit dialog, the text strings must be updated in the `translate()` method. This method is called whenever the user changes the current language of the program. Note the usage of the `tr()` function in line 33, which does the actual translation. The localized strings are specified elsewhere.

Instructions To write a new unit dialog plugin, follow these steps:

1. Below the respective unit directory, create a folder named `plugin/gui`. In case of the hold element function block, we create the directory `HoldElement.unit/plugin/gui`. Then, add your source and header files to that directory. By convention, at least three files are required for a dialog plugin:

   - A shared library entry point definition file, typically called `main.cpp` (listing 17).
   - A plugin declaration file, typically called *`FunctionBlockName`*`Dialog.h`, where *`FunctionBlockName`* is the unit type name without spaces, each word starting with an uppercase letter (e.g., `HoldElementDialog.h`; listing 18).
   - A plugin implementation file, typically called *`FunctionBlockName`*`Dialog.cpp` (listing 19).

   Note: The first argument to the `ADD_UNITWIDGET()` macro in `main.cpp` (listing 17) must exactly match the unit type identifier specified in `unit.xml` (see section 4.2.1).

Listing 17: Unit dialog plugin, `plugin/dialog/main.cpp`

```
1  #include "plugin/gui/Interface.h"
2
3  #include "HoldElementDialog.h"
4
5  BEGIN_GUI_INTERFACE("My Company", "Hold Element")
6    BEGIN_UNITWIDGETS
7      ADD_UNITWIDGET("My Company Hold Element",
8        "dialog", HoldElementDialog)
9    END_UNITWIDGETS
10 END_GUI_INTERFACE
```

2. Add a file called `CMakeLists.txt`[5] with the content shown in listing 20.

   Note: The plugin name specified in line 12 of listing 20 is also used for the name of the shared library that contains the plugin. Hence, it must correspond to the filename specified in the plugins section of `unit.xml`. See section 4.2.8 for more information.

[5]Please use exactly this case for the filename.

Listing 18: Unit dialog plugin, `plugin/gui/HoldElementDialog.h`

```
1  #ifndef _HOLDELEMENTDIALOG_H_
2  #define _HOLDELEMENTDIALOG_H_
3
4  #include "plugin/gui/UnitDialog.h"
5
6  class HoldElementDialog : public plugin::gui::UnitDialog
7  {
8    Q_OBJECT
9  public:
10   HoldElementDialog (plugin::gui::UnitDelegate& del);
11
12   virtual void update (const plugin::gui::UnitHistoryBundle& buf);
13   virtual void translate (void);
14
15 private:
16   QLabel* _label;
17   QLCDNumber* _inputDisplay;
18
19 };
20
21 #endif // #ifndef _HOLDELEMENTDIALOG_H_
```

3. Then, in the directory where `CMakeLists.txt` is located, create a folder named `build`. Open a command shell, navigate to that folder and execute CMake as shown below:
   ```
   > C:
   > cd "C:\Program Files\EasyLab\units\tutorial\HoldElement.unit"
   > cd plugin\gui\build
   > cmake -G "Visual Studio 9 2008" ..
   ```
   Note the trailing ".." in the last command. For a list of valid values to the `-G` argument, run `cmake` without any arguments and look in the "Generators" section of the output. You should see output similar to the one shown in listing 21.

   If the output is that `cmake` could not be found, ensure that CMake's `bin` directory is in the `%PATH%` and relaunch the command shell. If an error occurs during processing, make sure that `CMakeLists.txt` is set up correctly and all paths match.

4. Now we can load the generated Solution in Visual Studio. Navigate to the `build` directory and open `unit_holdelement_dialog.sln`. Switch the build profile to "Release". If everything went OK, you should now be able to compile your project.

   If the build fails with missing include files or linker library files, please double-check the paths specified in `CMakeLists.txt`.

   Note: If you corrected an error, you do normally *not* need to run CMake again, but instead CMake should auto-detect that you modified `CMakeLists.txt` and start a new configuration run by itself. In this case you should see the text "CMake is re-running because build system is out-of-date" in the Visual Studio console and Visual Studio may prompt you to reload the solution or specific projects, because they have been changed externally. You should confirm the reload.

Listing 19: Unit dialog plugin, `plugin/gui/HoldElementDialog.cpp`

```
#include "HoldElementDialog.h"

HoldElementDialog::HoldElementDialog (plugin::gui::UnitDelegate& del)
: plugin::gui::UnitDialog(del)
{
  QHBoxLayout* layout = new QHBoxLayout();
  setLayout(layout);

  _label = new QLabel();
  layout->addWidget(_label);

  _inputDisplay = new QLCDNumber();
  layout->addWidget(_inputDisplay);

  // Register to be notified when a new value is set on the "input"
  // variable. We only need the most recent value (history size 1).
  del.registerStateVariable("value", 1);
}

void HoldElementDialog::update
  (const plugin::gui::UnitHistoryBundle& buf)
{
  plugin::gui::VariableHistoryBuffer* valueBuf =
    buf.getStateVariableHistory("value");
  if (valueBuf) {
    // Update LCD number widget
    _inputDisplay->display(valueBuf->last().toDouble());
  }
}

void HoldElement::translate (void) {
  // Re-translate all texts visible to the user
  _label->setText(tr("Value:"));
}
```

Listing 20: Unit dialog plugin, plugin/gui/CMakeLists.txt

```
cmake_minimum_required(VERSION 2.6)

# Only use release configurations
set(
  CMAKE_CONFIGURATION_TYPES "Release;RelWithDebInfo;MinSizeRel"
  CACHE TYPE INTERNAL FORCE
)

# Replace "holdelement" with your unit type name
set(
  UNITPLUGIN_GUI_NAME
  unit_holdelement_dialog
)
project(${UNITPLUGIN_GUI_NAME})

file(GLOB UNITPLUGIN_GUI_SRCS *.cpp)
file(GLOB UNITPLUGIN_GUI_HEADER *.h)

# Adapt paths as needed
set(EASYLAB_INSTALL_DIR "C:/Program Files/EasyLab")
set(BOOST_INSTALL_DIR "C:/Program Files/boost/1_35_0")
set(QT_INSTALL_DIR "C:/Qt/qt-win-opensource-src-4.5.1")

include(
  # Add as many ../ as are necessary to refer to the respective file
  # (depending on where your unit is located in the "units" directory)
  "${CMAKE_CURRENT_SOURCE_DIR}/../../unitplugin_gui.cmake"
)
```

Listing 21: Unit dialog plugin, CMake output

```
-- Check for working C compiler: cl.exe
-- Check for working C compiler: cl.exe -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: cl.exe
-- Check for working CXX compiler: cl.exe -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Looking for Q_WS_X11
-- Looking for Q_WS_X11 - not found.
-- Looking for Q_WS_WIN
-- Looking for Q_WS_WIN - found
-- Looking for Q_WS_QWS
-- Looking for Q_WS_QWS - not found.
-- Looking for Q_WS_MAC
-- Looking for Q_WS_MAC - not found.
-- Found Qt-Version 4.5.1
-- Configuring done
-- Generating done
-- Build files have been written to:
    C:/.../units/tutorial/HoldElement.unit/plugin/gui/build
```

5. After the build is complete, we have to copy the compiled shared library to the unit directory. To do this, simply execute the `INSTALL` target in Visual Studio (this will only work if the DLL has been built in "Release" configuration). You have to execute this deployment step after every change to the plugin!

6. If you have access to the EasyLab Subversion repository, you might want to commit your work. Please respect the following rules when adding files to the repository:

    - In the directory `plugin/gui`, add just the source and header files as well as the file `CMakeLists.txt`. In our example this would be `main.cpp`, `HoldElementDialog.h`, `HoldElementDialog.cpp` and `CMakeLists.txt`.
    - *Never* add the `build` directory to version control.
    - Do *not* add compiled versions of the plugin (i.e., `*.dll` files) to version control.

4.4.2 Writing a Unit Visualization Plugin

Unit visualization plugins allow the author of a unit to define a custom appearance for instances of that specific unit type in various views used in EasyLab GUI. For example, a hold element unit may display its current value in the *SDF schematic* view. To write a new unit visualization plugin, follow these steps:

This section will be added later on. Please contact us if you need help.

5 Adding a new Device Type

5.1 Writing a Device Type Specification

This section will be added later on. Please contact us if you need help.

5.2 Writing a Device Simulation Plugin

This section will be added later on. Please contact us if you need help.

5.3 Writing a Device GUI Plugin

5.3.1 Writing a Device Dialog Plugin

This section will be added later on. Please contact us if you need help.

5.3.2 Writing a Unit Toolbar Plugin

This section will be added later on. Please contact us if you need help.

6 Adding a new Uploader Plugin

6.1 Writing an Uploader Specification

This section will be added later on. Please contact us if you need help.

6.2 Writing an Uploader Plugin

This section will be added later on. Please contact us if you need help.

7 Adding a new Debugger Plugin

7.1 Writing a Debugger Specification

This section will be added later on. Please contact us if you need help.

7.2 Writing a Debugger Plugin

This section will be added later on. Please contact us if you need help.

8 Adding a new Service Plugin

8.1 Writing a Service Specification

This section will be added later on. Please contact us if you need help.

8.2 Writing a Service Plugin

This section will be added later on. Please contact us if you need help.

9 Frequently Asked Questions

Q: What should my plugin do if it encounters a critical error?

A: If your simulation plugin encounters an unexpected situation, it should *not* cause an assertion failure (this would crash EasyLab). Instead, it should raise an exception of class `easygen::plugin::PluginException` or a derived class. This will allow EasyLab to safely terminate the simulation and display the error message to the user. The plugin may also provide its own error handling, for example by adding an additional output port named `error`. However, we discourage the use of such mechanisms. Listing 22 shows an example of how to raise an exception on a critical error.

Listing 22: Example for raising an exception during simulation

```
1  void Division::step (void) {
2    double dividend = readInput("dividend").toDouble();
3    double divisor = readInput("divisor").toDouble();
4
5    if (0 != divisor) {
6      writeOutput("quotient", dividend / divisor);
7    } else {
8      // Terminate simulation
9      throw plugin::simulation::PluginException("Division␣by␣zero!");
10   }
11 }
```

Q: How can I ensure functional consistency between simulation and code templates?

A: At the moment, we do not provide a mechanism to guarantee this. We suggest keeping the code as simple as possible and testing it with all possible data types (in case of units with type templates).

Q: Why do I have to explicitly specify the state variables for my function blocks? Wouldn't it be easier to just use (static) variables in my code templates?

A: You could do that. However, there are a number of reasons why you shouldn't do it that way:

- When used at the right granularity, state variables provide an easy way to debug a function block. State variables can also be hidden from the user if you do not want them to be visible.
- Default values for state variables can easily be configured within EasyLab. This is not possible for custom variables in code templates.
- When using state variables, EasyLab takes care about state variable (memory) management. It ensures that state variables are allocated correctly and that their values are reset when the correspondig parent program (e.g., the parent SDF of the function block) is restarted. You could do that manually, but it's error-prone.
- When the user creates multiple instances of your function blocks, static state variables will refer to the same address and state will be shared between function blocks. This is most probably not what you want and hence this design pattern is not supported by EasyLab (i.e., it will not work in all situations, even if it works

for a specific feature). We highly discourage users from using static variables in code templates.

Q: Can I use static variables in code templates?

A: We highly discourage the usage of static variables in code templates (at least for those that could be instantiated multiple times). For example, using static variables for the internal state of functions blocks is a bad idea, because function blocks can be instantiated multiple times and hence the state of the function blocks will be shared. Note that even if this sounds like an interesting approach for sharing data, it violates EasyLab's design pattern of modularity and hence *must not be used*.

Q: Is it possible to use multiple threads in plugins?

A: Yes. Some device simulation plugins will require the creation of threads to allow to efficiently deal with asynchronous data. For unit simulation plugins, multi-threading may be meaningful on multicore systems when the data to be processed can be easily split into disjoint parts. Note that appropriate synchronization mechanisms are needed between threads and that you should always use the same thread for communicating with EasyLab's plugin API.

Multi-threading may also be interesting for dialog plugins, although we discourage using extra threads, because dialogs should only be needed to be updated when either `update()` is called or when the user interacts with them, which is all possible without the creation of separate threads. Asynchronous behavior is better off in the respective *simulation* plugin. In case such behavior is really needed, a `QTimer` should be used instead of threads.

A Localization

This section will be added later on. Please contact us if you need help.

List of Tables

1 Availability of connectors and state variables in code templates 19
2 Overview of callback methods in a unit simulation plugin 19
3 Overview of data access methods in a unit simulation plugin 19
4 Overview of callback methods in a unit dialog plugin 24
5 Overview of data access methods in a unit dialog plugin 24

List of Listings

1 Unit type definition in XML, `unit.xml` 12
2 Unit type definition in XML, type template specification (compact form) . . 13
3 Unit type definition in XML, type template specification (explicit form) . . . 14
4 Unit type definition in XML, connector specification 15
5 Unit type definition in XML, state variable specification 15
6 Unit type definition in XML, plugin reference specification 16
7 Unit type definition in XML, code generation specification 17
8 Unit type definition in XML, code template 17
9 Unit type definition in XML, code reference 18
10 External code template, `code/holdelement.c` 18
11 External code template, `code/holdelement.c` (alternative solution) 18
12 Unit simulation plugin, `plugin/simulation/main.cpp` 20
13 Unit simulation plugin, `plugin/simulation/HoldElement.h` 21
14 Unit simulation plugin, `plugin/simulation/HoldElement.cpp` 21
15 Unit simulation plugin, `plugin/simulation/CMakeLists.txt` 22
16 Unit simulation plugin, CMake output 22
17 Unit dialog plugin, `plugin/dialog/main.cpp` 25
18 Unit dialog plugin, `plugin/gui/HoldElementDialog.h` 26
19 Unit dialog plugin, `plugin/gui/HoldElementDialog.cpp` 27
20 Unit dialog plugin, `plugin/gui/CMakeLists.txt` 28
21 Unit dialog plugin, CMake output 29
22 Example for raising an exception during simulation 31