# mbeddr C User's Guide

Marcel Matzat[1], Bernhard Merkle[2], and Markus Voelter[3]

[1] itemis AG
[2] Sick AG
[3] independent/itemis

**Abstract.** mbeddr is a extensible, C-based programming environment optimized for embedded programming based on JetBrains MPS. This document describes the mbeddr stack from a user's perspective. It guides through the installation process of mbeddr.core, provides a simple "hello world" tutorial and discusses the differences to regular C.

This document is part of the
mbeddr project at `http://mbeddr.com`.

# Table of Contents

# 1   Language Use vs. Language Development

This document focuses on the C programmer who wants to exploit the benefits of the extensions to C provided by mbeddr. We assume that you have some knowledge of regular C (such asK&R C, ANSI C or C99). We also assume that you realize some of the shortfalls of C and are "open" to the improvements in mbeddr C. The main point of mbeddr C is the ability to extend it with domain-specific concepts such as state machines, components, or whatever you deem useful in your domain. We have also removed some of the "dangerous" features of C that are often prohibited from use in real world projects.

This document covers mbeddr *core*, which is mainly a subset of the C language for MPS. We do not discuss any extensions of the C language for development in the embedded area, such as e.g. support for statemachines, type safe units etc. These will be discussed in separate documents once they become available.

**Note:** As of now, these extension modules are not yet released. They have to be polished further. They will become available in 2012.

This document does not discuss how to develop new languages or extend existing languages. We refer to the *Extension Guide* instead. It is available from `http://mbeddr.com`.

# 2   Installation

At this point we don't yet provide an all-in-one download package. This is because (a) we didn't get around to building one yet, and (b) as a consequence of a number of open legal issues regarding re-packaging some of the third-party tools used by mbeddr. However, this documentation describes the installation process in detail.

## 2.1   Java

MPS is a Java application. So as the first step, you have to install a Java Development Kit version 1.6 or greater (JDK 1.6). You can get it from here
`http://www.oracle.com/technetwork/java/javase/downloads/index.html`

## 2.2   JetBrains Meta Programming System (MPS)

The mbeddr system is based on JetBrains MPS, an open source language workbench available from `http://www.jetbrains.com/mps/`. MPS is available for Windows, Mac and Linux, and we require the use of the 2.0.4 version. Please make sure you install MPS in a path that does not contains blanks in any of its

directory or file names (not even in the `MPS 2.0` folder). This will simplify some of the command line work you may want to do.

After installing MPS using the platform-specific installer, please open the `bin` folder and edit the `mps.vmoptions` or `mps.exe.vmoptions` file (depending on your platform). To make MPS run smoothly, the `MaxPermSize` setting should be increased to `512m`. It should look like this after the change:

```
-client
-Xss1024k
-ea
-Xmx1200m
-XX:MaxPermSize=512m
-XX:+HeapDumpOnOutOfMemoryError
-Dfile.encoding=UTF-8
```

On some 32bit Windows XP systems we had to reduce the `-Xmx1200m` setting to `768m` to get it to work.

## 2.3 GCC and make

The mbeddr toolkit relies on `gcc` and `make` for compilation (unless you use a different, target-specific build process).

– On Mac you should install XCode to get gcc, make and the associated tools.
– On Linux, these tools should be installed by default.
– On Windows we recommend installing cygwin (`http://www.cygwin.com/`), a Unix-like environment for Windows. When selecting the packages to be installed, make sure `gcc-core` and `make` are included (both of them are in the `Devel` subtree in the selection dialog). The `bin` directory of your cygwin installation has to be added to the system `PATH` variable; either globally, or in the script that starts up MPS (MPS runs `make`, so it has to be visible to MPS). On Windows, the `mps.bat` file in the MPS installation folder would have to be adapted like this:

```
::rem mbeddr depends on Cygwin: gcc, make etc
::rem adapt the following to your cygwin bin path
set PATH=C:\ide\Cygwin\bin;%PATH%
```

## 2.4 Graphviz

MPS supports visualization of models via graphviz, directly embedded in MPS. To use it, you have to install graphviz from `http://graphviz.org`. We use version 2.28. After the installation, you have to put the `bin` directory of graphviz

into the path. Either globally, or by modifying the MPS startup script in the
same way as above:

```
::rem mbeddr depends on graphviz dot
::rem adapt the following to your graphviz bin path
set PATH=C:\ide\graphviz2.28\bin;%PATH%
```

## 2.5 mbeddr

You can get the mbeddr code either via distributions or via the public github
repository.

**Distribution** You can get the mbeddr system via a zip file download from
`http://mbeddr.wordpress.com/getit/` Save the zip file into a folder on
your hard disk and unzip it. Once again, please make sure the path to the
unzipped folder contains no blanks!

**github** The github repo ist a `https://github.com/mbeddr/mbeddr.core`. You
can clone it for your own use, or you can fork it to your own github account
so you can make changes. Contact us if you want to become a committer.

## 2.6 Debugger

The mbeddr debugger is based on `gdb`, which has been installed as part of the
Cygwin install. However, we don't use `gdb` directly; rather we use the Eclipse
CDT debug bridge. This contains a native code, and Java has to be able to find
this native code. Hence, the `java.library.path` property has to be set as part
of the MPS JVM startup. It has to point to a path that depends on your OS
platform.

Open the mps startup script (`.bat` or `.sh`) and look for the line that begins
with

```
set ADDITIONAL_JVM_ARGS=
```

It should be used to set the property, as shown in the following code (keep
it in one line!):

```
set ADDITIONAL_JVM_ARGS=-Djava.library.path=
  "<mbeddr-root>/code/languages/com.mbeddr.core/languages/com.mbeddr.core.debug/lib/spawner/<os>"
```

The `<mbeddr-root>` points to the root directory of the mbeddr languages as
cloned from github or unzipped after download. `<os>` has to be set specifically

for your platform, it can be any of `linux.x86`, `linux.x86_64`, `macosx..x86_64`, `macosx.ppc`, `macosx.x86`, `win32.x86`, `win32.x86_64`. We suggest you take a look at the respective directory structure to better understand what's going on.

The debugger needs a special library to control the `gdb` process. We packaged this library into a MPS plugin named *spawner*. Please download this plugin from bit.ly/K8HPD2 and unzip it into the `plugins` directory of your MPS installation.

## 3 Important keyboard shortcuts in MPS and mbeddr

### 3.1 MPS in general

MPS is a projectional editor. It does not parse text and build an Abstract Syntax Tree (AST). Instead the AST is created directly by user editing actions, and what you see in terms of text (or other notations) is a projection. This has many advantages, but it also means that some of the well-known editing gestures we know from normal text editing don't work. So in this section we explain some keyboard shortcuts that are essential to work with MPS.

Since the very first experience a projectional editor is somewhat different from what you are accustomed to in a text editor, we recommend you watch the following screencast:

> `http://www.youtube.com/watch?v=wgsY3-ZX_fs`

*Entering Code* In MPS you can only enter code that is available from the code completion menu. Using aliases and other "tricks", MPS manages to make this feel *almost* like text editing. Here are some hints though:

– As you start typing, the text you're entering remains red, with a light red background. This means the string you've entered has not yet *bound*.
– Entered text will bind if there is only one thing left in the code completion menu that starts with the substring you've typed so far. An instance of the selected concept will be created and the red goes away.
– As long as text is still red, you can press `Ctrl-Space` to explicitly open the code completion menu, and you can select from those concepts that start with the substring you have typed in so far.
– If you want to go back and enter something different from what the entered text already preselects, press `Ctrl-Space` again. This will show the whole code completion menu.
– Finally, if you're trying to enter something that does not bind at all because the prefix you've typed does not match anything in the code completion menu, there is no point in continuing to type; it won't ever bind. You're probably trying to enter something that is not valid in this place. Maybe you haven't included the language module that provides the concept you have in mind?

*Navigation* Navigation in the source works as usual using the cursor keys or the mouse. References can be followed ("go to definition") either by `Ctrl-Click` or by using `Ctrl-B`.

*Selection* Selection is different. `Ctrl-Up/Down` can be used to select along the tree. For example consider a local variable declaration `int x = 2 + 3 * 4;` with the cursor at the 3. If you now press `Ctrl-Up`, the `3 * 4` will be selected because the `*` is the parent of the 3. Pressing `Ctrl-Up` again selects `2 + 3 * 4`, and the next `Ctrl-Up` selects the whole local variable declaration.

You can also select with `Shift-Up/Down`. This selects siblings in a list. For example, consider a statement list as in a function body ...

```
void aFunction() {
  int x;
  int y;
  int z;
}
```

... and imagine the cursor in the x. You can press `Ctrl-Up` once to select the whole `int x;` and then you can use `Shift-Down` to select the y and z siblings. Note that the screencast mentioned above illustrates these things much clearer.

*Deleting Things* The safest way to delete something is to mark it (using the strategies discussed in the previous paragraph) and the press `Backspace` or `Delete`. In many places you can also simply press `Backspace` behind or `Delete` before the thing you want to delete.

*Intentions* Some editing functionalities are not available via "regular typing", but have to be performed via what's traditionally known as a quick fix. In MPS, those are called intentions. The intentions menu can be shown by pressing `Alt-Enter` while the cursor is on the program element for which the intention menu should be shown (each language concept element has its own set of intentions). For example, module contents in mbeddr can only be set to be `exported` by selecting *export* from the intentions menu. Explore the contents of the intentions menu from time to time to see what's possible.

Note that you can just type the name of an intention once the menu is open, you don't have to use the cursor keys to select from the list. So, for example, to export a module content (function, struct), you type `Alt-Enter`, `ex`, `Enter`.

*Surround-With Intentions* Surround-With intentions are used to surround a selection with another construct. For example, if you select a couple of lines (i.e. a list of statements) in a C program, you can then surround these statements with an `if` or with a `while`. Press `Ctrl-Alt-T` to show the possible surround options at any time. To reemphasize: in contrast to regular intentions which are opened by `Alt-Enter`, surround-with intentions can work on a selection that contains several nodes!

*Refactorings* For many language constructs, refactorings are provided. Refactorings are more important in MPS than in "normal" text editors, because some (actually quite few) editing operations are hard to do manually. Please explore

the refactorings context menu, and take note when we explain refactorings in the user's guide. Unlike intentions, which cannot have a specific keyboard shortcut assigned, refactorings can, and we make use of this feature heavily. The next section introduces some of these.

## 3.2 mbeddr specific shortcuts

*Documentation* Many program elements can be documented. Examples include statements, functions, global variables or structs. A documentation is basically free text associated with a program element.

We distinguish documentation from commenting out code (explained below). A documentation is shown as a grey comment above the commented element (the documentation is really attached to the element, and not just written into a line above it — a subtle but important difference!)

```
// Here is some documentation for the function
int8_t main(string[ ] args, int8_t argc) {
  // ... and here is some doc for the report statement
  report(0) HelloWorldMessages.hello() on/if;
  return 0;
} main (function)
```

Documentation can be added using the *Add Documentation* intention, or by using `Ctrl-Alt-D` on the respective element.

Note that comments are only shown if they are turned on. You can use the context menu on any program element and select *ToggleDocs* to enable/disable display of comments. As soon as a comment is added, comment display is automatically turned on.

*Commenting out Code* Code that is commented out retains its syntax highlighting, but is shaded with a grey background.

```
// // Here is some documentation for the function
int8_t main(string[ ] args, int8_t argc) {
  // ... and here is some doc for the report statement
  report(0) HelloWorldMessages.hello() on/if;
  return 0;
}
```

Code can be commented out by pressing `Ctrl-Alt-C` (this is technically a refactoring, so this feature is also availale from the refactorings context menu). This also works for lists of elements. Commented out code can be commented back in by pressing `Ctrl-Alt-C` on the comment itself (the //) or the commented element.

Commenting out code is a bit different than in regular, textual systems because code that is commented out is still "live": it is still stored as a tree, code completion still works in it, it may still be shown in $FindReferences$, and refactorings may affect the code. We are not sure if this is a desirable feature and we are looking for your feedback. Of course, the code is not executed. All commented program elements are removed during code generation.

**Note:** The current implementation of comments is still a little bit of a hack since we are waiting for some direct support by MPS. For example, errors should not be shown in commented code, and we are sure other quirks will arise as we continue using mbeddr.

Not all program elements can be commented out (since special support by the language is necessary to make something commentable), only concepts that implement `ICommentable` can be commented. At this time, this is all statements and module contents.

## 4 Hello World Example

For this tutorial we assume that you know how to use the C programming language. We also assume that you have have installed MPS, gcc/make, graphviz and the mbeddr.core distribution. This has been disussed in the previous section.

### 4.1 Create new project

Start up MPS and create a new project. Call the project `HelloWorld` and store it in a directory without blanks in the path. Let the wizard create a solution, but no language.
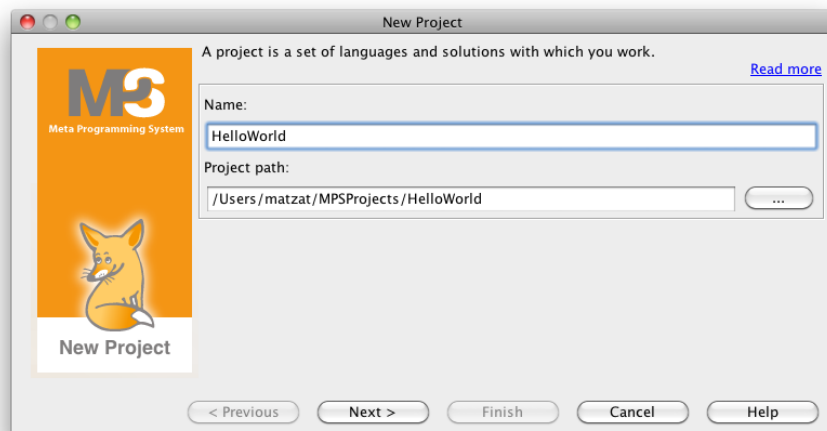
We now have to make the project aware of the *mbeddr.core* languages installed via the distribution. Go to the $File \rightarrow Settings$ and select the $Global\,Libraries$ in the IDE settings. Create a library called `mbeddr.core` that points to the root directory of the unzipped mbeddr installation.

**Note: This library must point to the root directory of the checkout so that all languages are below it, including *core* and *mpsutil*.**
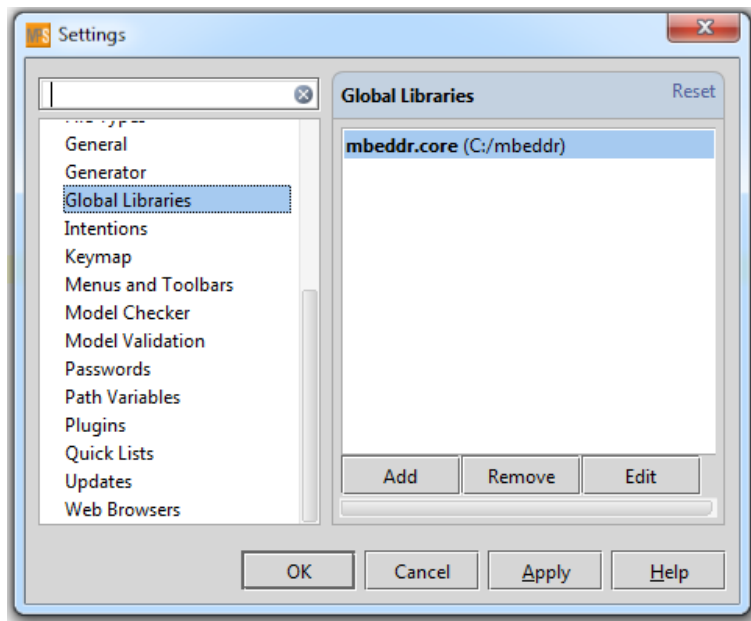


Notice that this is a settings and have to be performed only once before your first application project.

## 4.2   Project Structure and Settings

An MPS project is a collection of solutions[4]. A *solution* is an application project that *uses* existing languages. Solutions contain any number of models; models contain root nodes. Physically, models are XML files that store MPS programs. They are the relevant version control unit, and the fundamental unit of configuration.

---

[4]  A project can also contain *languages*, but these are only relevant to language implementors. We discuss this aspect of mbeddr in the *Extension Guide*
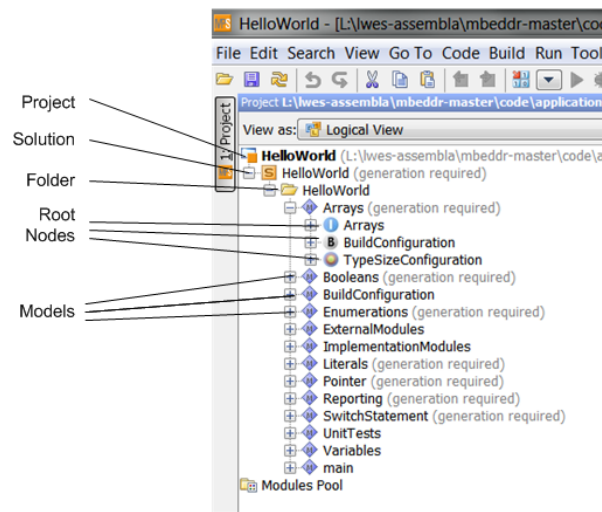
In the solution, create a new model with the name `main`, prefixed with the solution's name: select *New* → *Model* from the solution's context menu. No sterotype.

A model has to be configured with the languages that should be used to write the program in the model. In our case we need all the `mbeddr.core` languages. We have provided a *devkit* for these languages. A devkit is essentially a set of languages, used to simplify the import settings. As you create the model, the model properties dialog should open automatically. In the *Used Devkits* section, select the + button and add the `com.mbeddr.core` devkit.
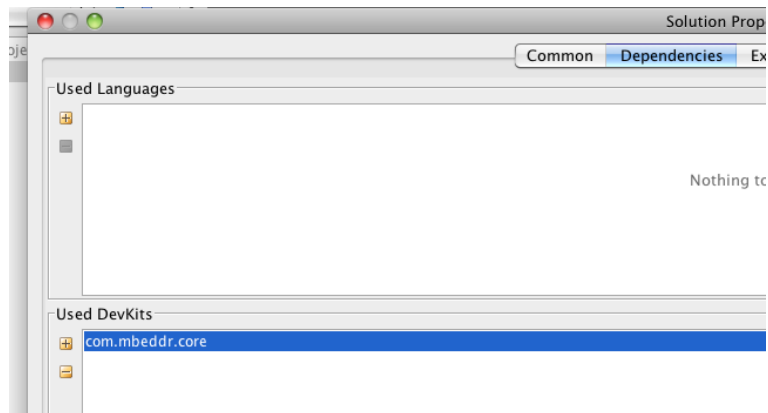


This concludes the configuration and setup of your project. You can now start writing C code.

### 4.3 Create an empty Module

The top level concept in mbeddr C programs are *modules*. Modules act as namespaces and as the unit of encapsulation. So the first step is to create an empty module. The mbeddr.core C language does not use the artificial separation between `.h` and `.c` files you know it from classical C. Instead mbeddr C uses the aforementioned module concept. During code generation we then create the corresponding `.h` and `.c` files.

A module can import other modules. The importing module can then access the *exported* contents of imported modules.

So to get started, we create a new `implementation module` using the model's context menu as shown in the following screenshot:
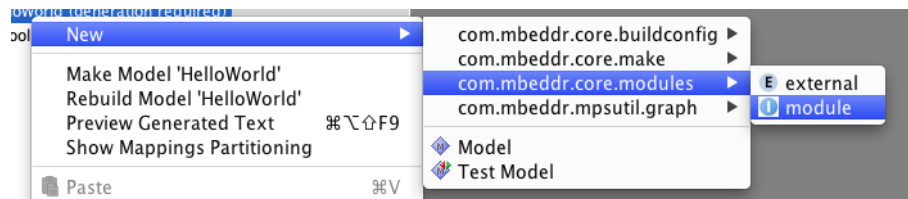


Note: This operation, as well as almost all others, can be performed with the keyboard as well. Take a look at *File → Settings → Keymap* to find out or change keyboard mappings.

As a result, you will get an empty implementation module. It currently has no name (the name is red and underlined) and only a placeholder «...» where top level C constructs such as functions, `struct`s, or `enum`s can be added later.
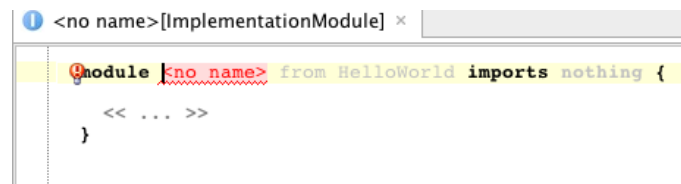


Next, specify `HelloWorld` as the name for the implementation module.

```
module HelloWorld from HelloWorld.main imports nothing {
  << ... >>
}
```

The module name is still underlined in red because of a missing type size configuration. The `TypeSizeConfiguration` specifies the sizes of the primitive types (such as `int` or `long`) for the particular target platform. mbeddr C provides a *default* type size configuration, which can be added to a module via an intention

*Create default type size configuration* on the module in the editor. To invoke intentions please use `Alt-Enter`. You may have to press `F5` to make the red underline go away. For more details on type size configurations see chapter 5.5.
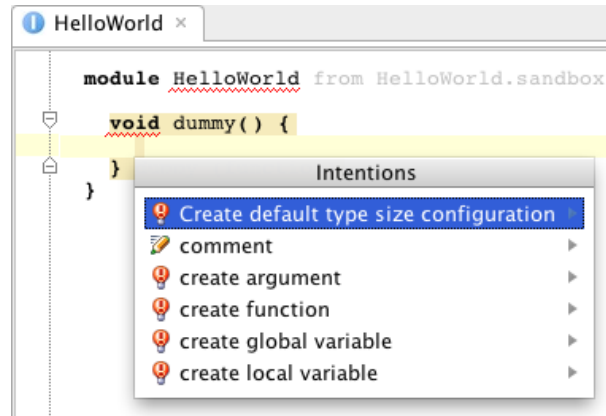


## 4.4  Writing the Program

Within the module you can now add contents such as functions, `struct`s or global variables. Let's enter a `main` function so we can run the program later. You can enter a `main` function in one of the the following ways:

- create a new function instance by typing `function` at the placeholder in the module, and then specify the name and arguments.
- simply start typing the return type of the function e.g. `int32_t` in this case, and then entering the name.
- specifically for the main function, you can also just type `main` (it will set up the correct signature automatically)

At this point, we are ready to implement the Hello World program. Our aim is to simply output a log message and return 0. To add a return value, move the cursor into the function body and type `return 0`.

```
module HelloWorld from HelloWorld.main imports nothing {
  int32_t main() {
    return 0;
  }
}
```

To print the message we could use `printf` or some other `stdio` function. However, in embedded systems there is often no `printf` or the target platform has no display available, so we use a special language extension for logging. It will be translated in a suitable way, depending on the available facilities on the target platform. Also, specific log messages can be deactivated in which case

they are completely removed from the program. Below our main function we create a new **message list** (just type **message** followed by return) and give it the name **log**.

Within the message list, hit **return** or type **message** to create a new message. Change the type from **ERROR** to **INFO** with the help of code completion. Specify the name **hello**. Add a message property by hitting **return** between the parentheses. The type should be a **string** and the name should be **who**. Specify **Hello** as the value of the **message text** property. The resulting message should look like this:

```
message list log {
  INFO hello(string who) active: Hello
}
```

Now you are ready to use the message list and its messages from your main function. Insert a **report()** statement in the main function, specify the message list **log** and select the message **hello**. Pass the string **"World"** as parameter.

```
module HelloWorld from HelloWorld.main imports nothing {

  int32_t main(int8_t argc, string[ ] args) {
    report(0) log.hello("World") on/if;
    return 0;
  } main (function)


  message list log {
    INFO hello(string who) active: Hello
  }
}
```
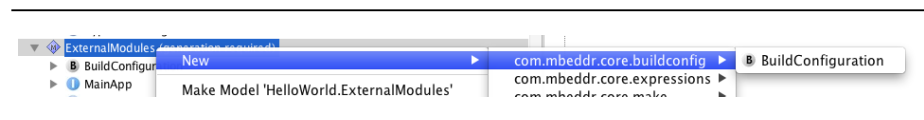


**Fig. 1.** Add a BuildConfiguration Model

### 4.5 Build Configuration

We have to create one additional element, the **BuildConfiguration**. This specifies which modules should be compiled into an executable or library, as well as other aspects related to creating an executable. Depending on the selected target

platform, a `BuildConfiguration` will automatically generate a corresponding `make` file. In the main model, create a new instance of `BuildConfiguration` (via the model's context menu, see Fig. 1). Initially, it will look as follows:

```
Target Platform:
  <no target>

Configuration Items
  << ... >>

Binaries
  << ... >>
```

You will have to specify three aspects. First you have to select the target platform. For our tests, we use the `desktop` platform that generates a `make` file that can be compiled with the normal `gcc` compiler. Other target platforms may generate build scripts for other build systems. The `desktop` target contains some useful defaults, e.g. the `gcc` compiler and its options.

```
Target Platform:
  desktop
    compiler: gcc
    compiler options: -std=c99
    debug options: <no debug options>
```

Next, we have to address the configuration items. These are additional configuration data that define how various program elements are translated. In our case we have to specify the reporting configuration. Select the placeholder and type `reporting`. It determines how log messages are output. The `printf` strategy simply prints them to the console, which is fine for our purposes here.

```
Configuration Items
  reporting: printf
```

Finally, in the `Binaries` section, we create a new `exectuable` and call it `HelloWorld`. In the program's body, add a reference to the `HelloWorld` implementation module we've created before. The code should look like this:

```
executable main isTest: false {
  used libraries
    << ... >>
  included modules
    HelloWorld
}
```

### 4.6 Building and Executing the Program

Press `Ctrl-F9` (or `Cmd-F9` on the Mac) to rebuild the solution. In the
`HelloWorld/solutions/HelloWorld/source_gen/HelloWorld/main` directory
you should now have at least the following files (there may be others, but those
are not important now):

```
Makefile
HelloWorld.c
HelloWorld.h
```

The files should be already compiled as part of the mbeddr C build facet (i.e.
`make` is run by MPS automatically). Alternatively, to compile the files manu-
ally, open a command prompt (must be a cygwin prompt on Windows!) in this
directory and type `make`. The output should look like the following:

```
\$ make
rm -rf ./bin
mkdir -p ./bin
gcc -c -o bin/HelloWorld.o HelloWorld.c -std=c99
```

This builds the executable file `HelloWorld.exe` or `HelloWorld` (depending
on your platform), and running it should show the following output:

```
\$ ./HelloWorld.exe
hello: Hello @HelloWorld:main:0
  world = World
```

Note the output of the log statement in the program (report statement num-
ber 0 in function `main` in module `HelloWorld`; take a look back at the source
code: the index of the statement (here: 0) is also output in the program source).

### 4.7 The Graph Viewer

The graph viewer allows to render all graphviz files in the current solution. It
simply scans the `source_gen` directory recursively for `.gv` files and shows them
in the tree. Underscores in the file name are used as hierarchies in the tree. Users
can create their own transformation to graphviz graphs, and they will be shown
in the tree. A special `graph` language is available for this transformation (will
be explained later).

By default, each build configuration results in a diagram that shows the
dependencies between the modules. To see it in the graph viewer,

– Open an implementation module in the editor

- select *Tools− > OpenGraphViewer* from the menu
- in the graph viewer, click through the tree until you find a leaf node representing a diagram

The diagram should open in the lower pane of the graph viewer. You can zoom (mouse wheel) and move around (press mouse button and move). More interestingly, you can also click on a node, and the MPS editor selects the respective node.

This concludes our hello world example. In the next section we will examine important differences between mbeddr C and regular C.

## 5   Differences to regular C

This section describes the differences between *mbeddr C* and regular C99. All examples shown in this chapter can be found in the *HelloWorld* project that is available for download together with the *mbeddr.core* distribution.

### 5.1   Preprocessor

mbeddr C does not support the preprocessor. Instead we provide first class concepts for the various use cases of the C preprocessor. This avoids some of the chaos that can be created by misusing the preprocessor and provides much better analyzability. We will provide examples later.

The major consequence of not having a preprocessor is that the separation between header and implementation file does not work anymore. mbeddr provides `modules` instead.

### 5.2   Modules

While we *generate* header files, we don't *expose* them to the user in MPS. Instead, we have defined modules as the top-level concept. Modules also act as a kind of namespace. Module contents can be exported, in which case, if a module is imported by another module, the exported contents can be used by the importing module.

We distinguish between *implementation modules* which contain actual implementation code, and *external modules* which act as proxies for pre-existing header files that we want to be able to use from within mbeddr C programs.

*Implementation Modules* The following example shows an implementation module (`ImplementationModule`) with an exported function. You can toggle the *exported* flag with the intention *Toggle Export*. The second module (`ModuleUsingTheExportedFunction`) imports the `ImplementationModule` with the `imports` keyword in the module

header. An importing module can access all exported contents defined in imported modules.

```
module ImplementationModule from HelloWorld.ImplementationModules
  imports nothing {

  exported int32_t add(int32_t i, int32_t j) {
    return i + j;
  } add (function)
}

module ModuleUsingTheExportedFunction from HelloWorld.ImplementationModules
  imports ImplementationModule {

  int32_t main(int8_t argc, string[ ] args) {
    int32_t result = add(10, 15);
    return 0;
  } main (function)
}
```

*External modules* mbeddr C code must be able to work with existing code and existing C libraries. So to call existing functions or instantiate `structs`, we use the following approach:

- We identify existing external header files and the corresponding object or library files.
- We create an *external module* to represent those; the external module specifies the `.h` file and the object/library files it represents.
- In the external module we add the contents of the existing `.h` files we want to make accessible to the mbeddr C program.
- We can now import the external module into any implementation module from which we want to be able to call into the external code
- The generator generates the necessary `#include` statements, and the corresponding build configuration.

**Note:** In the future we will provide a mechanism to automatically import existing header files into an external module. As of now, the relevant signatures etc. have to be typed in manually.

The following code shows the external module `STDIO`. In the `resources` section, you have to provide the path to the resources associated with this external module. You can add `headers` and `linkables` (`.o` or `.a` files) here. Since `gcc` knows what to link when `<stdio.h>` is included, we don't have to specify a linkable here.

```
external module STDIO resources header : <stdio.h>
  // external module contents are exported by default
{
  void printf(string format, ...);
}
```

To call methods from external modules, you have to import the external module into your implementation module with `imports STDIO`. You can add now call the `printf` function defined in the external module.

```
module MainApp from HelloWorld.ExternalModules imports STDIO {

  int32_t main() {
    printf("Dies ist ein stdout.printf Text: %s\n", "Noch einer");
    return 0;
  } main (function)
}
```

## 5.3    Build configuration

The `BuildConfiguration` specifies how a model should be translated and which modules should be compiled into an executable. Typically it will be generated into a `make` file that performs the compilation. We have discussed the basics as part of the Hello World in Section 4.5. We won't repeat the basics here.

The main part of the build configuration supports the definition of binaries. Binaries are either executables or libraries.

*Executables* An executable binds together a set of modules and compiles them into an executable. Exactly one module in a executable shall have a main function.

The build configuration, if it uses the `desktop` target, results in a `make` file which is automatically invoked as part of the MPS build, resulting in the corresponding executable binaries. The generated code, the make file and the executables can be found in the `source_gen` folder of the respective solution.

**Note:** The build language is designed to be extended for integration with other build infrastructures. In that case, other targets (than `desktop`) would be provided by the language that provides integration with a particular build infrastructure.

Below is the build configuration of the `ExternalModules` example. It defines one executable `Application`. It consist of the modules `MainApp` and `STDIO`.

```
executable Application isTest: false {
  used libraries
    << ... >>
  included modules
    MainApp
    STDIO
}
```

*Libraries* Libraries are binaries that are not executable. Specifically, they are `libXXXX.a` files. They can be linked into executables. A library will typically reside in its own MPS model (and hence in its own `source_gen` directory).

To create a library, create a build configuration with a `static library`:

```
static library MathLib {
  MyFirstModule
  MyOtherModule
}
```

Running the resulting make file will create a `libMathLib.a`.

Using the library for inclusion in an executable (which *has* to be in a different MPS model!) requires the following three steps:

- You have to import the model. Open the properties of the model that contains the code that *uses* the library, and add the model that *contains* the library to the `Imported Models` (Fig. 2). This is necessary so that MPS can see the nodes defined in that model.
- In the implementation module that wants to use the functionality defined in the library, import the corresponding module(s) from the library. The importing module will see all the exported contents in the imported module (this is just like any other inter-module dependency).
- finally, in the build configuration of the executable that *uses* the library, the used library has to be specified in the `used libraries` section.

```
executable AnExe isTest: true {
  used libraries
    MathLib
  included modules
    MainModule
}
```

**Extending the Build Process** The build configuration is built in a way it is easily extensible. We will discuss details in the extension guide, but here are a couple of hints:

- New configuration items can be contributed by implementing the `IConfigurationItem` interface. They are expected to be used from transformation code. It can find the relevant items by querying the current model for a root of type `IConfigurationContainer`.
- New platforms can be contributed by extending the `Platform` concept. Users then also have to provide a generator for `BuildConfigurations`.
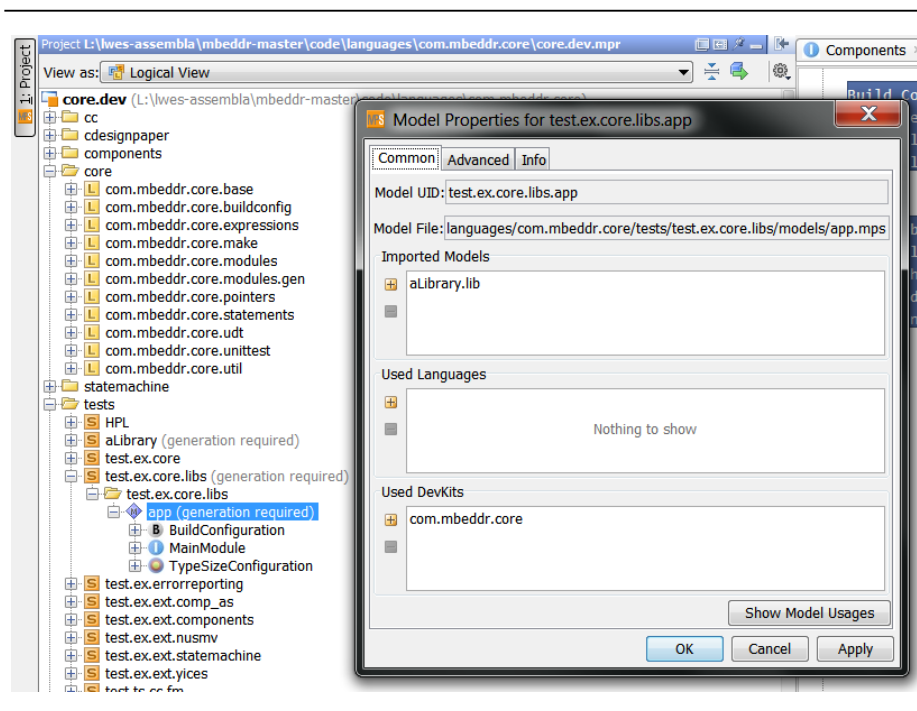
**Fig. 2.** Importing a model that contains a library

## 5.4 Unit tests

Unit Tests are supported as first class citizens by mbeddr C. A `TestCase` implements `IModuleContent`, so it can be used in implementation modules alongside with functions, `structs` or global variables. To assert the correctness of a result you have to use the `assert` statement followed by an Boolean expression (note that `assert` can just be used *only* inside test cases). A `fail` statement is also available — it fails the test unconditionally.

```
module AddTest from HelloWorld.UnitTests imports nothing {

  exported test case testAddInt {
    assert(0) 1 + 2 == 3;
    assert(1) -1 + 1 == 1;
  } testAddInt(test case)


  exported test case testAddFloat {
    float f1 = 5.0;
    float f2 = 10.5;
    assert(0) f1 + f2 == 15.5;
  } testAddFloat(test case)
}
```

The next piece of code shows a main function that executes the test cases imported from the `AddTest` module. The `test` expression supports invocations of test cases; it also evaluates to the number of failed assertions. By returning this value from `main`, we get an exit code $!= 0$ in the case a test failed.

```
module TestSuite from HelloWorld.UnitTests imports AddTest {
  int32_t main() {
    return test testAddInt, testAddFloat;
  } main (function)
}
```

In the build configuration, the `isTest: true` flag can be set to true; this adds a `test` target to the make file, so you can call `make test` on the command line in the `source_gen` folder to run the tests.

The example above contains a failing assertion `assert(1) -1 + 1 == 1;`. Below is the console output after running `make test` in the generated source folder for the solution:

```
runningTest: running test @AddTest:test_testAddInt:0
FAILED: ***FAILED*** @AddTest:test_testAddInt:2
  testID = 1
runningTest: running test @AddTest:test_testAddFloat:0
make: *** [test] Error 1
```

If you change the assertion to `assert(1) -1 + 1 == 0;`, rebuild with `Ctrl-F9` and rerun `make test` you will get the following output, which has no errors:

```
runningTest: running test @AddTest:test_testAddInt:0
runningTest: running test @AddTest:test_testAddFloat:0
```

## 5.5   Primitive Numeric Datatypes

The standard C data types (`int`, `long`, etc.) have different sizes on different platforms. This makes them non-portable. C99 provides another set of primitive data types with clearly defined sizes (`int8_t`, `int16_t`). In mbeddr C you *have* to use the C99 types, resulting in more portable programs. To be able to work with existing header files, the system has to know how the C99 types relate to the standard primitive types. This is the purpose of the `TypeSizeConfiguration`. It establishes a mapping between the C99 types and the standard primitive types.

The `TypeSizeConfiguration` mentioned above can be added with the *Create default type size configuration* (Fig. 3) on modules, or by creating one through the *New* menu on models. Every model has to contain exactly one type size configuration. To fill an existing empty type size configuration with the default values, you can use an intention on the `TypeSizeConfiguration`.
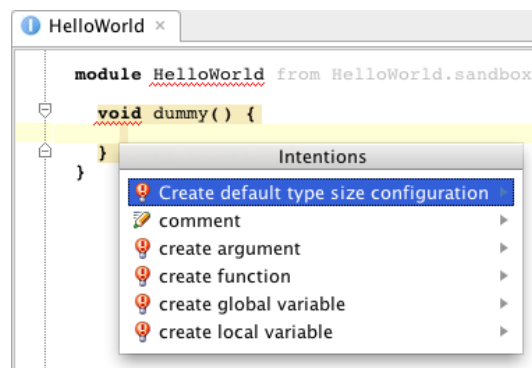


**Fig. 3.** Create default `TypeSizeConfiguration`

*Integral Types*  The following integral types are not allowed in implementation modules, and can only be used in external modules for compatibility: `char`, `short`, `int`, `long`, `long long`, as well as their unsigned counterparts. The following list shows the default mapping of the C99 types:

- int8_t → char
- int16_t → short
- int32_t → int
- int64_t → long long
- uint8_t → unsigned char
- uint16_t → unsigned short
- uint32_t → unsigned int
- uint64_t → unsigned long long

*Floating Point Types* The size of floating point types can alsoi be speficied, e.g. if they differ from the IEEE754 sizes.

- float → 32
- double → 64
- long double → 128

## 5.6 Booleans

We have introduced a specific `boolean` datatype, including the `true` and `false` literals. Integers cannot be used interchangably with Boolean values. We do provide a (red, ugly) cast operator between integers and booleans for reasons of interop with legacy code. The following example shows the usage of the Boolean data type.

```
module BooleanDatatype from HelloWorld.BooleanDatatype imports nothing {
  exported test case booleanTest {
    boolean b = false;
    assert(0) b == false;
    if ( !b ) { b = true; } if
    assert(1) b == true;
    assert(2) int2bool<1> == true;
  } booleanTest(test case)
}
```

## 5.7 Literals

mbeddr C supports special literals for hex, octal and binary numbers. The type of the literal is the smallest possible signed integer type (int8_t, ..., int64_t) that can represent the number.

```
module LiteralsApp from HelloWorld.Literals imports nothing {

  exported test case testLiterals {
    int32_t intFromHex = hex<aff12>;
    assert(0) intFromHex == 720658;

    int32_t intFromOct = oct<334477>;
    assert(1) intFromOct == 112959;
```

```
        int32_t intFromBin = bin<100110011>;
        assert(2) intFromBin == 307;
    } testLiterals(test case)
}
```

## 5.8   Pointers

C supports two styles of specifying pointer types: `int *pointer2int` and `int*`
`pointer2int`. In mbeddr C, only the latter is supported: pointer-ness is a characteristic of a type, not of a variable.

*Pointer Arithmetics*  For pointer arithmetics you have to use an explicit type
conversion `pointer2int` and `int2pointer`. For more details, look at the following example. You also see the usage of pointer dereference (`*xp`) and assigning
an address with `&`.

```
module BasicPointer from HelloWorld.Pointer imports stdlib {

  exported test case testBasicPointer {

    int32_t x = 10;
    int32_t* xp = &x;
    assert(0) *xp == 10;

    int32_t[ ] anArray = {4, 5};
    int32_t* ap = anArray;
    assert(1) *ap == 4;

    // pointer arithmetic
    ap = int2pointer<pointer2int<ap> + 1>;
    assert(2) *ap == 5;

  } testBasicPointer(test case)
  ...
}
```

Memory allocation works the same way as in regular C except that you need
an external module to call functions such as `malloc` from `stdlib`. The next
example shows how to do this. Note that `size_t` is a primitive type, built into
mbeddr. It's size is also defined in a `TypeSizeConfiguration`.

```
external module stdlib resources header : <stdlib.h>
{
  void* malloc(size_t size);
  void free(void* pointer);
}
```

You have to include the external module `stdlib` in your implementation
module with `imports stdlib`. You can then call `malloc` or `free`:

```
module BasicPointer from HelloWorld.Pointer imports stdlib {
  ...
  exported test case mallocTest {
    int8_t* mem = ((int8_t*) malloc(sizeof int8_t));
    *mem = 10;
    assert(0) *mem == 10;
    free(mem);
  } mallocTest(test case)
}
```

*Function Pointers* In regular C, you define a function pointer type like this: `int (*pt2Function) (int, int)`. The first part is the return type, followed by the name und a comma separated argument type list. The pointer asterisk is added before the name. This is a rather ugly notation; we've cleaned it up in mbeddr C.

In mbeddr, we have introduced the notion of function types and function references. These are syntactically different from pointers (of course they are mapped to function pointers in the generated C code). We have also introduced lambdas (i.e. closures without their own state).

For function types you first define the argument list and then the return type, separeted by `=>` (a little bit like Scala). Here is an example: `(int32_t, int32_t)=>(int32_t)` You can enter a fuction type by using the `funtype` alias (see Fig. 4). Function types are types, so they can be used in function signatures, local variables or `typedef`s, just like any other type (see example *HelloWorld.Pointer.FunctionPointerAsTypes*).
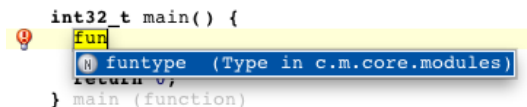


**Fig. 4.** Add a function pointer with code completion

*Values of type* `funtype` are either references to functions or lambdas. In regular C, you have use the address operator to obtain a function pointer (`&function`). In mbeddr C, you use the `:` operator (as in `:someFunction`) do distinguish function referneces from regular pointer stuff. Of course the type and values have to be compatible; for function types this means that the signature must be the same. The following example shows the use of function references:

```
module FunctionPointer from HelloWorld.Pointer imports nothing {
```

```
  int32_t add(int32_t a, int32_t b) {
    return a + b;
  } add (function)


  int32_t minus(int32_t a, int32_t b) {
    return a - b;
  } minus (function)

  exported test case testFunctionPointer {
    // function pointer signature
    (int32_t, int32_t)=>(int32_t) pt2Function;

    // assign "add"
    pt2Function = :add;
    assert(0) pt2Function(20, 10) == 30;

    // assign "minus"
    pt2Function = :minus;
    assert(1) pt2Function(20, 10) == 10;
  } testFunctionPointer(test case)
}
```

Function types can be used like any other type. This is illustrated in the next example. The typedef `typedef (int3_t, int32_t)=>(int32_t) as ftype;` defines a new function type. The type `ftype` is the first parameter in the `doOperation` function. You can easily call the function `doOperation(:add, 20, 10)` and put any suitable function reference as the first parameter.

```
module FunctionPointerAsTypes from HelloWorld.Pointer imports nothing {

  typedef (int32_t, int32_t)=>(int32_t) as ftype;

  int32_t add(int32_t a, int32_t b) {
    return a + b;
  } add (function)

  exported test case testFunctionPointer {
    // call "add"
    assert(0) doOperation(:add, 20, 10) == 30;
  } testFunctionPointer(test case)

  int32_t doOperation(ftype operation, int32_t firstOp, int32_t secondOp) {
    return operation(firstOp, secondOp);
  } doOperation (function)
}
```

Lambdas are also supported. Lambdas are essentially functions without a name. They are defined as a value and can be assigned to variables or passed to a function. The syntax for a lambda is `[arg1, arg2, ...|an-expression-using-args]`. The following is an example:

```
module Lambdas from HelloWorld.Pointer imports nothing {

  typedef (int32_t, int32_t)=>(int32_t) as ftype;
```

```
    exported test case testFunctionPointer {
      assert(0) doOperation([a, b|a + b;], 20, 10) == 30;
    } testFunctionPointer(test case)

    int32_t doOperation(ftype operation, int32_t firstOp, int32_t secondOp) {
      return operation(firstOp, secondOp);
    } doOperation (function)
}
```

## 5.9   Enumerations

The mbeddr C language also provides enumeration support, comparable to to
C99. There is one difference compared to regular C99. In mbeddr C an enumer-
ation is not an integer type. This means, you can't do any arithmetic operations
with enumerations.

**Note:** We may add a way to cast enums to ints later if it turns out that "enum
arithmetics" are necessary

.

```
module EnumerationApp from HelloWorld.Enumerations imports nothing {

  enum SEASON { SPRING; SUMMER; AUTUMN; WINTER; }

  exported test case testEnumeration {
    SEASON season = SPRING;
    assert(0) season != WINTER;
    season = WINTER;
    assert(1) season == WINTER;
  } testEnumeration(test case)
}
```

## 5.10   Goto

There is no `goto` in mbeddr C.

**Note:** This may change :-)

## 5.11   Switch statement

In the `switch` statement, we don't use the annoying fall through semantics.
Only one `case` within the `switch` will ever be executed, since we automatically

generate a break statement into the generated C code. You can also add an `default` statement which will be executed if no other case match.

The next example shows a `switch` statement with integers and enumeration as the switched expression.

```
module SwitchStatement from HelloWorld.SwitchStatement imports nothing {

  var int32_t globalState;

  enum DAY { MONDAY; THUESDAY; WEDNESDAY; }

  exported test case testSwitchCase {
    globalState = -1;

    // Switch with int
    callSwitch(0);
    assert(0) globalState == 20;

    callSwitch(1);
    assert(1) globalState == 0;

    callSwitch(2);
    assert(2) globalState == 10;


    // Switch with day
    callSwitchWithEnumeration(MONDAY);
    assert(3) globalState == 1;

    callSwitchWithEnumeration(WEDNESDAY);
    assert(4) globalState == 3;

    callSwitchWithEnumeration(THUESDAY);
    assert(5) globalState == 2;
  } testSwitchCase(test case)


  void callSwitch(int32_t state) {
    switch ( state ) {
      case 1: { globalState = 0; break; }
      case 2: { globalState = 10; break; }
      default: { globalState = 20; break; }
    } switch
  } callSwtich (function)


  void callSwitchWithEnumeration(DAY day) {
    switch ( day ) {
      case MONDAY: { globalState = 1; break; }
      case THUESDAY: { globalState = 2; break; }
      case WEDNESDAY: { globalState = 3; break; }
    } switch
  } callSwitchWithEnumeration (function)
}
```

## 5.12 Variables

**Global variables** Global variables start with the keyword `var`. In every other respect they are identical to regular C. Like all other module contents, it can be `exported`.

```
module GlobalVariables from HelloWorld.Variables imports nothing {

  var int32_t globalInt32;

  exported test case testGlobalVariables {
    setGlobalVar(10);
    assert(0) globalInt32 == 10;
    setGlobalVar(20);
    assert(1) globalInt32 == 20;
    return;
  } testGlobalVariables(test case)

  void setGlobalVar(int32_t globalVarValue) {
    globalInt32 = globalVarValue;
  } setGlobalVar (function)
}
```

**Local variables** At this point a local variable declaration can only declare one variable at a time; otherwise is it is just like in C (so you cannot write `int a,b;`).

## 5.13 Arrays

Array brackets must show up after the type, not the variable name. The following example shows the usage of arrays in mbeddr C, which also supports multi-dimensional arrays. Their usage is equivalent to regular C.

```
module ArrayApplication from HelloWorld.Arrays imports nothing {

  exported test case arrayTest {
    int32_t[3] array = {1, 2, 3};

    assert(0) array[0] == 1;

    int8_t[2][2] array2 = {{1, 2}, {3, 4}};
    assert(1) array2[1][1] == 4;
  } arrayTest(test case)
}
```

## 5.14 Reporting

Reporting or logging is provided as a special concept. It's designed as a platform-independent reporting system. With the current generator and the `desktop` setting in the build configuration, `report` statements are generated into a `printf`. For other target platforms, other translations will be supported in the future, for example, by storing the message into some kind of error memory.

If you want to use reporting in your module, you first have to define a `message list` in a module. Inside, you can add `MessageDefinitions` with three different severities: `ERROR` (default), `INFO` and `WARN`.

Every message definition has a name (acts as an identifier to reference a message in a report statement), a severity, a string message and any number of additional arguments. Currently, only integer values and strings are allowed.

A `report` statement references a message from a message list and supplies values for all arguments defined by the message. The following example shows an example (`active` refers to the fact that these messages have not been disabled; use the corresponding intentions on the messages to enable/disable each message).

```
module Reporting from HelloWorld.Reporting imports nothing {

  message list demo {
    INFO programStarted() active: Program has just started running
    ERROR noArgumentPassedIn(int16_t actualArgCount) active:
        No argument has been passed in, although an arg is expected
  }

  int32_t main(int8_t argc, string[ ] args) {
    report(0) demo.programStarted();
    report(1) demo.noArgumentPassedIn(argc) on argc == 0;
    return 0;
  } main (function)
}
```

Note how the first report statement outputs the message in all cases. The second one only outputs the message if a condition is met.

Report statements can be disabled; this removes all the code from the program, so no overhead is entailed. Intentions on the message definition support enabling and disabling messages. It is also possible to enable/disable groups of messages by using intentions on the message list.

**Note:** At this time there is no way of enabling/disabling messages at runtime. This will be added in the future.

### 5.15   Assembly Code

At this point we are not able to write inline assembler. We will enable this feature in the future.

## 6   Command Line Generation

mbeddr C models can be generated to C code from the command line using `ant`. The `HelloWorld` project comes with an example ant file: in the project root directory, you can find a `build.xml` ant file:

```
<project name="HelloWorld" default="build">

  <property file="build.properties"/>

  <taskdef resource="jetbrains/mps/build/ant/antlib.xml"
           classpath="\${mps.home}/languages/generate.ant.task.jar"/>

  <target name="build">
    <mps.generate loglevel="info" fork="true" failonerror="true">
      <jvmargs id="myargs">
       <arg value="-Xmx512m"/>
      </jvmargs>
      <project file="\${mbeddr.home}/code/applications/HelloWorld/HelloWorld.mpr"/>
      <library name="mbeddr" dir="\${mbeddr.home}/code/languages"/>
    </mps.generate>
  </target>

</project>
```

It uses the `mps.generate` task provided with MPS. All the code is boilerplate, except these two lines:

```
<project file="\${mbeddr.home}/code/applications/HelloWorld/HelloWorld.mpr"/>
<library name="mbeddr" dir="\${mbeddr.home}/code/languages"/>
```

The first line specifies the project whose contents should be generated. We point to the `HelloWorld.mpr` project in our case. If you only want to generate parts of a project (only some solutions or models), take a look at this article: http://confluence.jetbrains.net/display/MPSD2/HowTo+-+MPS+and+ant

The second line points to the directory that contains all the languages used by the to-be-generated project.

To make it work, you also have to provide a `build.properties` file to define two path variables:

```
mps.home=/some/path/to/MPS2.0/
mbeddr.home=/the/path/to/mbeddr/
```

Assuming you have installed **ant**, you can simply type **ant** at the command prompt in the directory that contains the `build.xml` file. Unfortunately, generation takes quite some time to execute (50 seconds on my machine). However, most of the time is startup and loading all the languages, so having a bigger program won't make much of a difference. The output should look like this:

```
L:\lwes-assembla\mbeddr\code\applications\HelloWorld>ant
Buildfile: build.xml

build:
[mps.generate] Build number MPS Build.MPS-20.7460
[mps.generate] Loaded project MPSProject file: L:\lwes-assembla\mbeddr\code\applications\HelloWorld\HelloWorld.mpr
```

```
[mps.generate] Per-root generation set to false
[mps.generate] Generating:
[mps.generate]     MPSProject file: L:\lwes-assembla\mbeddr\code\applications\HelloWorld\HelloWorld.mpr

BUILD SUCCESSFUL
Total time: 57 seconds
```

You can now run `make` to build the executable.

# 7 Version Control - working with MPS, mbeddr and git

This section explains how to use git with MPS. It assumes a basic knowledge of git and the git command line. The section focuses on the integration with MPS. We will use the git command line for all of those operations that are not MPS-specific.

We assume the following setup: you work on your local machine with a clone of an existing git repository. It is connected to one upstream repository by the name of `origin`.

## 7.1 Preliminaries

**VCS Granularity** MPS reuses the version control integration from the IDEA platform. Consequently, the granularity of version control is the file. This is quite natural for project files and the like, but for MPS models it can be confusing at the beginning. Keep in mind that each *model*, living in solutions or languages, is represented as an XML file, so it is these files that are handled by the version control system.

**The MPS Merge Driver** MPS comes with a special merge driver for git (as well as for SVN) that makes sure MPS models are merged correctly. This merge driver has to be configured in the local git settings. In the MPS version control menu there is an entry *Install Version Control AddOn*. Make sure you execute this menu entry before proceeding any further. As a result, your `.gitconfig` should contain an entry such as this one:

```
[merge "mps"]
name = MPS merge driver
driver = "\"/Users/markus/.MPS20/config/mps-merger.sh\" %O %A %B %L"
```

**The .gitignore** For all projects, the `.iws` file should be added to `.gitignore`, since this contains the local configuration of your project and should not be shared with others.

Regarding the (temporary Java source) files generated by MPS, two approaches are possible: they can be checked in or not. Not checking them in means that some of the version control operations get simpler because there is less "stuff" to deal with. Checking them in has the advantage that no complete rebuild of these files is necessary after updating your code from the VCS, so this results in a faster workflow.

If you decide *not* to check in temporary Java source files, the following directories and files should be added to the `.gitignore` in your local repo:

- For languages: `source_gen`, `source_gen.caches` and `classes_gen`
- For solutions, if those are Java/BaseLanguage solutions, then the same applies as for languages. If these are other solutions to which the MPS-integrated Java build does not apply, then `source_gen` and `source_gen.caches` should be added, plus whatever else your own build process creates in terms of temporary files.

Make sure the `.history` files are *not* added to the `gitignore`! These are important for MPS-internal refactorings.

**MPS' caches and Branching** MPS keeps all kinds of project-related data in various caches. These caches are outside the project directory and are hence not checked into the VCS. This is good. But it has one problem: If you change the branch, your source files change, while the caches are still in the *old* state. This leads to all kinds of problems. So, as a rule, whenever you change a branch (that is not just trivially different from the one you have used so far), make sure you select `File -> Invalidate Caches`, restart and rebuild your project.

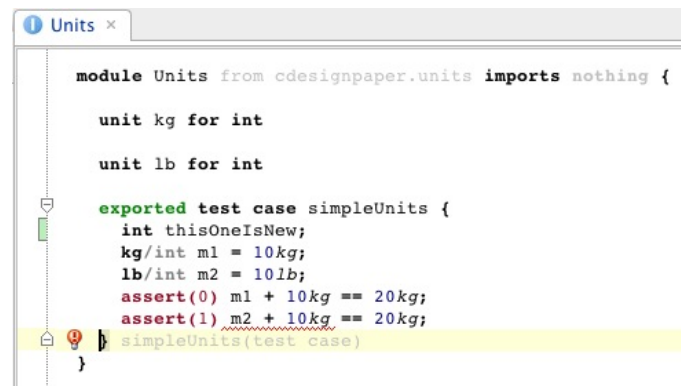Depending on the degree of change, this may also be advisable after pulling from the remote repository.



**Fig. 5.** A new variable has been added to the program and the gutter shows the green markup

## 7.2 Committing Your Work

In git you can always commit locally. Typically, commits will happen quite often, on a fine grained level. I like to do these from within MPS. Fig. 5 shows a program where I have just added a new variable. This is highlighted with the green bar in the gutter. Right-Clicking on the green bar allows you to rever this change to the latest checked in state.

In addition you can use the `Changes` view (from the `Window -> Tool Windows` menu) to look at the set of changed files. In my case (Fig. 6) it is basically one `.mps` file (plus two files realted to writing this document :-)). This `.mps` file contains the test case to which I have added the new variable.
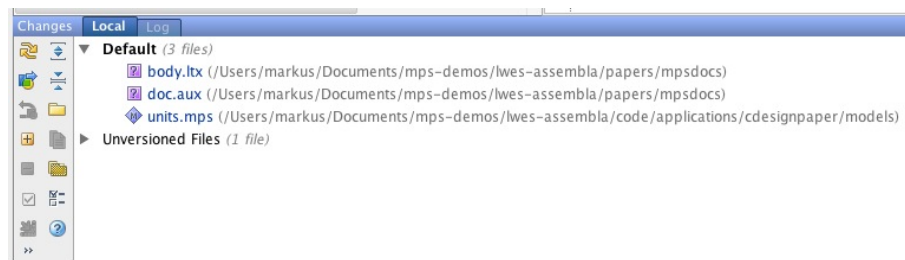


**Fig. 6.** A new variable has been added to the program and the gutter shows the green markup

To commit your work, you can now select `Version Control -> Commit Changes`. The resulting dialog, again, shows you all the changes you have made and you can choose which one to include in your commit. After committing, your `git status` will look something like this and you are ready to push:

```
Markus-Voelters-MacBook:lwes-assembla markus$ git status
# On branch demo
# Your branch is ahead of 'assembla/demo' by 1 commit.
#
nothing to commit (working directory clean)
Markus-Voelters-MacBook-Air:lwes-assembla markus$
```

## 7.3 Pulling and Merging

Pulling (or merging) from a remote repository or another branch is when you potentially get merge conflicts. I usually perform all these operations from the command line. If you run into merge conflicts, they should be resolved from within MPS. After the pull or merge, the `Changes` view will highlight conflicting files in red. You can right-click onto it and select the `Git -> Merge Tool` option.

This will bring up a merge tool on the level of the projectional editor to resolve the conflict. Please take a look at the screencast at

http://www.youtube.com/watch?v=gc9oCAnUx7I

to see this process in action.

The process described above and in the video works well for MPS model files. However, you may also get conflicts in project, language or solution files. These are XML files, but cannot be edited with the projectional editor. Also, if one of these files has conflicts and contains the < < < < and > > > > merge markers, then MPS cannot open these files anymore because the XML parser stumbles over these merge markers.

I have found the following two approaches to work:

– You can either perform merges or pulls while the project is closed in MPS. Conflicts in project, language and solution files should then be resolved with an external merge tool such as *WinMerge* before attempting to open the project again in MPS.
– Alternatively you can merge or pull while the project is open (so the XML files are already parsed). You can then identify those conflicing files via the `Changes` view and merge them on XML-level with the MPS merge tool. After merging a project file, MPS prompts you that the file has been changed on disk and suggests to reload it. You should do this.

Please also keep in mind my remark about invalidating caches above.

## 7.4 A personal Process with git

Many people have described their way of working with git regarding branching, rebasing and merging. In principle each of these will work with MPS, when taking account what has been discussed above. Here is the process I use.

To develop a feature, I create a feature branch with

```
git branch newFeature
git checkout newFeature
```

I then immediately push this new branch to the remote repository as a backup, and to allow other people to contribute to the branch. I use

```
git push -u origin newFeature
```

Using the `-u` parameter sets up the branch for remote tracking.

I then work locally on the branch, committing changes in a fine-grained way. I regularly push the branch to the remote repo. In less regular intervals I pull in the changes from the master branch to make sure I don't diverge too far from what happens on the master. I use merge for this:

```
git checkout master
git pull                    // this makes sure the master is current
git checkout myFeature
git merge master
```

Alternatively you can also use

```
git fetch
git checkout myFeature
git merge origin/master
```

This is the time when conflicts occur and have to be handled. In repeat this process until my feature is finished. I then merge my changes back on the master:

```
git checkout master
git pull                         // this makes sure the master is current
git merge --squash myFeature
```

Notice the `-squash` option. This allows me to "package" all of the commits that I have created on my local branch into a single commit with a meaningful comment such as "initial version of myFeature finished".

# 8   Debugging

mbeddr comes with a Debugger for core C. The debugger can also debug the standard extensions and is extensible for user-defined extensions. In this section we describe how to debug C programs.

**Note:** We assume the default configuration, where we use `gdb` as the debug backed. Debugging on some target device is currently not yet supported.

The Hello World project contains a model called `Debugger.Example` that contains a simple program that makes use of a number of C extensions. We will use this program to illustrate debugging.

```
module DebuggerExample from Debugger.Example imports nothing {

  int8_t add(int8_t x, int8_t y) {
    return x + y;
  }

  exported test case testAdding {
    assert(0) add(1, 2) == 3;
    assert(1) add(2, 4) == 6;
```
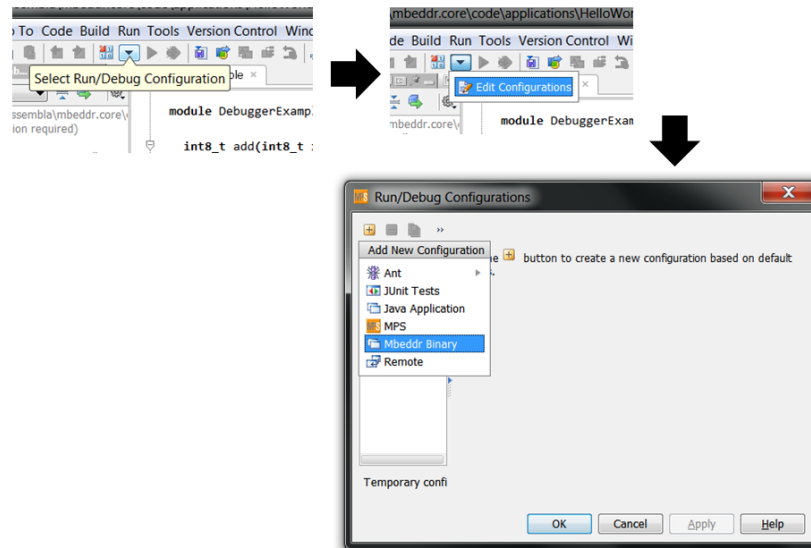
```
    }

  int32_t main(int32_t argc, int8_t*[ ] argv) {
    return test testAdding;
  }
}
```
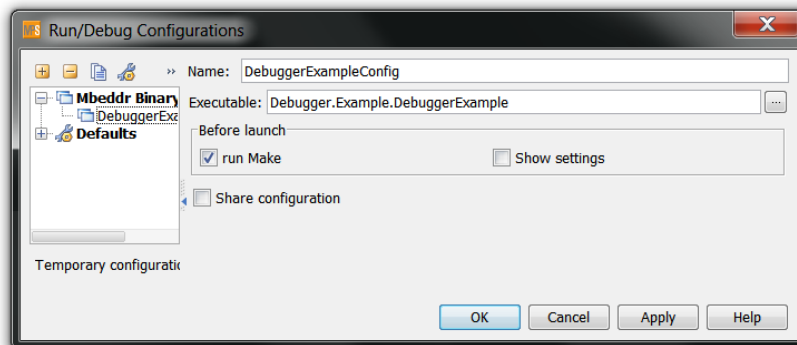
## 8.1 Creating a Debug Configuration

We start out by creating a new debug configuration as shown in the figure below:



In the resulting dialog, name the new configuration `DebuggerExampleConfig` and select the `Debugger.Example.DebuggerExample` executable via the `...` button (this executable is defined in the build configuration of the debugger example).

The launch configurations drop down at the top of the MPS screen should now show this new configuration:
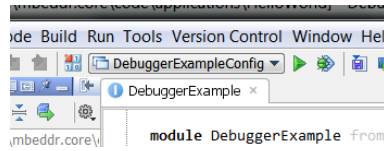


## 8.2 Running a Debug Session

Before we can run the debugger, we have to make sure the C code for the program has been generated. So select `Rebuild` from the context menu of the `Debugger.Example` model or press `Ctrl-F9`.

Now set a breakpoint in the first line of the test case. You can set breakpoints by clicking into the gutter of the editor. The result should look like this:
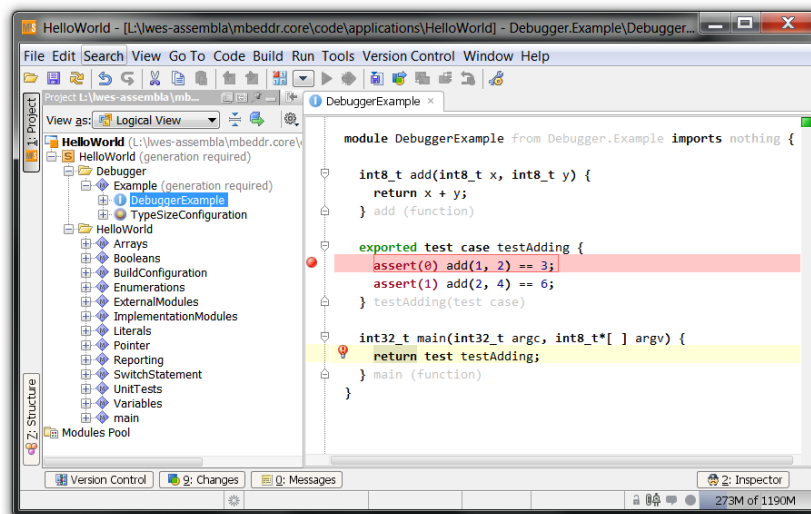


Next, run the previously created debug configuration by pressing `Shift-F9` or by selecting the debugger button in the MPS title bar (see next figure).
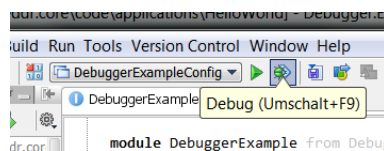


The debugger should start up and stop at the breakpoint we had set before.
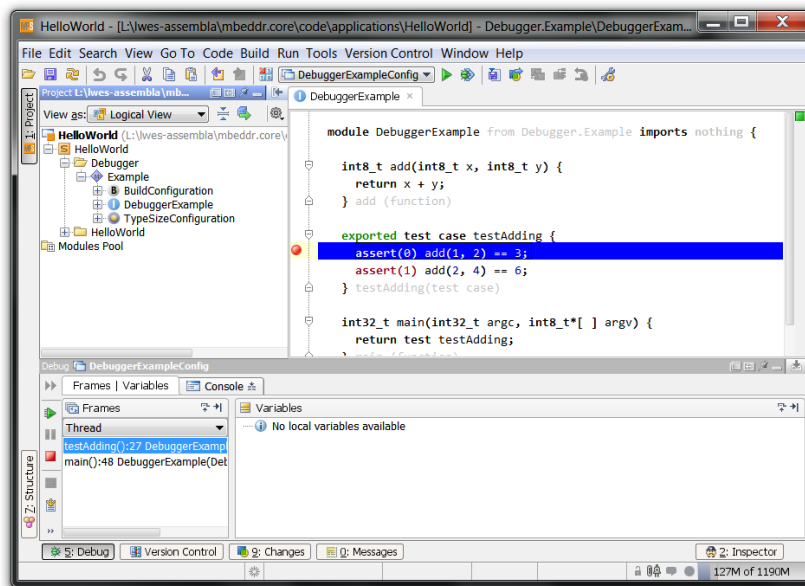
Next, press **F8** to *step over* the current line and then press **F7** to *step into* the **add** function on the second line of the test case. Once inside the function, you can see the nested stack frames as well as the local variables x and y.
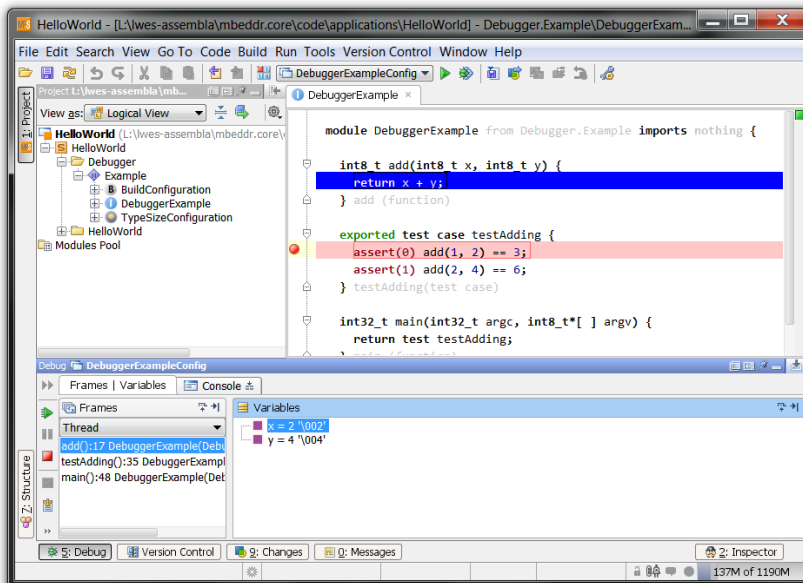
# 9   Graphs

This section explains how you can create custom graphs from your models. mbeddr comes with the `com.mbeddr.mpsutil.graph` language. It is an MPS language you can use to describe graphs. These graphs are then automatically translated into a `.gv` file, which is then picked up by the graphview for rendering.

**Note:** This section assumes that you have a basic understanding of MPS generators.

The example code for this chapter can be found in `code/applications/Callgraph`.

A call graph shows the *call* relationships between functions. Here is an example program:

```
module Module1 {

  void function1() {
    function2();
    function3();
  }

  void function2() {
    function3();
    function1();
  }

  void function3() {
    function4();
    function2();
  }

  void function4() {
    function1();
    function2();
  }

  int32_t main(int32_t argc, int8_t*[ ] argv) {
    function1();
    return 0;
  }
}
```

Here is the resulting call graph diagram we are going to create in this chapter:
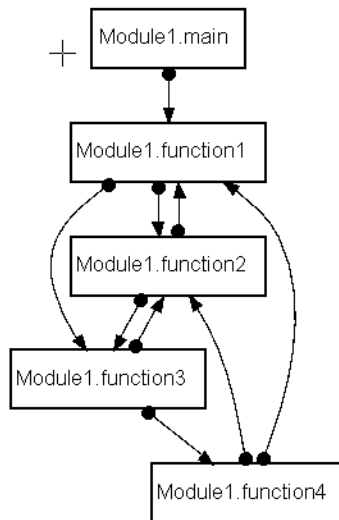
## 9.1 Setting up a Language

To define the transformation to the `graph` language we have to define our own language, even though we're not going to define any new *language* constructs, but just a generator. MPS still considers this a language.

We create a new language `callgraph`. It has to extend `com.mbeddr.core.modules` (because it defines functions, and we want to create a callgraph between functions), `com.mbeddr.core.buildconfig` (because we're going to hook the creation of the graph to the build configuration) and `com.mbeddr.core.base` (because we'll need one specific concept from this language, as we'll see later).

## 9.2 Creating the Generator

In this new language we now create a new generator. It contains one root mapping rule that maps a `BuildConfiguration` to a graph.

```
root mapping rules:
  ┌ concept          BuildConfiguration ┐ --> map_BuildConfiguration
  │ inheritors        false             │
  │ condition         <always>          │
  └ keep input root   true              ┘
```

The `map_BuildConfiguration` template creates the actual graph. It is a root template that uses a `Graph` from the `com.mbeddr.mpsutil.graph` language. To make this available, the generator model has to `use` this language. Here is the empty `Graph` node:

```
root template
input BuildConfiguration

graph $[map_BuildConfiguration] {
nodes:
  << ... >>
edges:
  << ... >>
}
```

If we generated this, it would create an empty and useless graph. So we now have to create a new `Node` for each function in each module in the executable created by the build configuration from which we generate the graph. So we first create a `node` object (that's a `node` from the `graph` language):

```
node function id someID (shape: rect t:"function") style normal
```

We specify an arbitrary name (`function`), an arbitrary ID (`someID`), we use a `rect` as the shape, a `normal` style and a label that is also arbitrary (`"function"`). Then we attach a `LOOP` macro. It is used to iterate over all relevant functions using the following `LOOP` code:

```
sequence<node<Binary>> allExecutables =
  node.binaries.where({~it => it.isInstanceOf(Executable);}).
  ofType<node<Executable>>; sequence<node<Module>>
allModules = allExecutables.
  selectMany({~it => it.referencedModules.module; });
allModules.selectMany({~it => it.descendants<concept = Function>; });
```

We then use a property macro on the function name that uses the qualified name of the function as the name of the node:

```
node.qualifiedName().replaceAll("\\.", "_");
```

We put the function's internal node ID as the ID of the created graph node using another property macro:

```
node.adapter.getId();
```

We use another property macro for the label of the node that also contains the qualified name of the function. Next up, we have to create an out edge for each function call in that function. So we add an out edge to the template graph node and `LOOP` over all functions:

```
root template
input BuildConfiguration

graph $[map_BuildConfiguration] {
nodes:
  $LOOP$ node $[function] id $[id] (shape: rect t:"$[function]") style bold {
           $LOOP$ out edge <--> id $[call] label <no label> linestyle: solid
                   from:<no sourcePort>
                   to ->$[function]:<no targetPort>
                   arrows tail dot head normal
        }
edges:
  << ... >>
}
```

Here is the expression we use in the `LOOP` macro:

```
comment       : <none>
mapping label : <no label>
mapped nodes  : (node, genContext, operationContext)->sequence<node<>> {
                    node.descendants<concept = FunctionCall>;
                }
```

We make the edge bidirectional (`<->`), we specify the ID to be the ID of the function call node (over which we currently iterate) and then we specify a `dot` tail arrow style and a `normal` head arrow style. Finally, we specify the target; we use the same `function` node defined above as the target, and then use a reference macro (`->`) to "rewire" the edge to its actual target node. Here is the expression in the reference macro:

```
node.function.qualifiedName().replaceAll("\\.", "_");
```

That's all we need to do in the generator template.

## 9.3   Generator Priorities

We have to make sure that our generator runs at the right time during the overall, multistep transformation process. To make our lives simple, we simple have it run at the same time as the generator that create the module dependencies. It runs as part of every transformation by default. So we open the generator properties of the generator we just wrote and specify a dependency to the `com.mbeddr.core.buildconfig` generator (it is the one creating the module dependency graph).
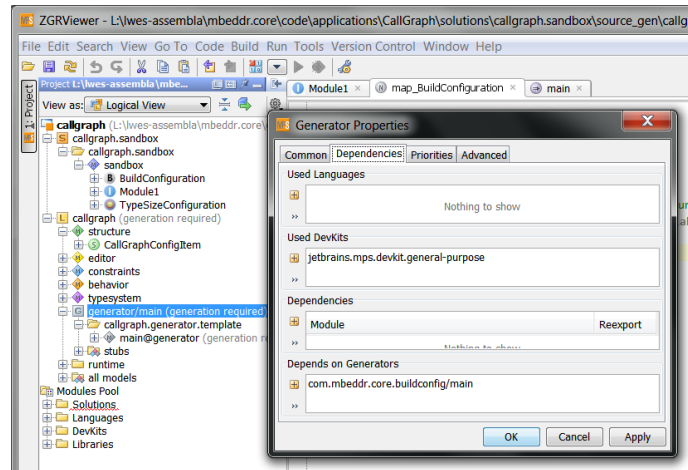
We then swap over to the next tab and specify that our generator runs at the same time (=) as the `com.mbeddr.core.buildconfig/main.dependencyGraph`.
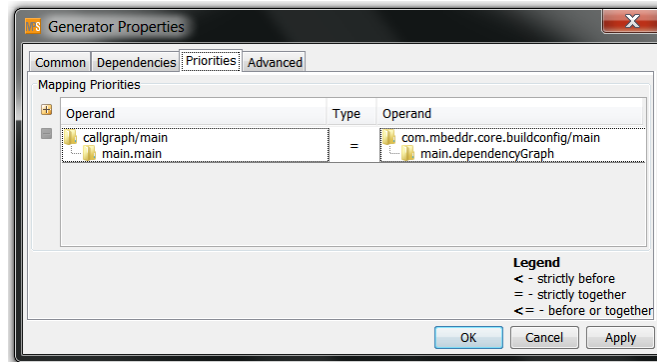


## 9.4 Making the Generation Optional

mbeddr comes with a generic configuration framework. `BuildConfiguration`s contain so-called configuration items:

```
Build System:
  desktop
    compiler: gcc
    compiler options: -std=c99
    debug options: -g

Configuration Items
  reporting: printf

Binaries
  executable Dummy isTest: false {
    used libraries
      << ... >>
    included modules
      Module1 (callgraph.sandbox/callgraph.sandbox.sandbox)
  }
```

We create a configuration item; we only generate the graph if the `callgraph` item is configured.

```
Configuration Items
  reporting: printf
  callgraph
```

To make enable this feature, we have to do two things. In our `callgraph` language we create a new language concept called `CallGraphConfigItem`. It has to implement the `IConfigurationItem` interface for this to work. The editor for the concept is simply the constant `callgraph`

```
concept CallGraphConfigItem extends      BaseConcept
                            implements IConfigurationItem


  instance can be root: false


  properties:
  << ... >>


  children:
  << ... >>


  references:
  << ... >>


  concept properties:
  alias = callgraph
```

Finally, we have to make sure the transformation only runs if this item is present. To do this, we go back to the mapping configuration and to the root mapping rule we created earlier. We make this conditional on the presence of the configuration item:

```
root mapping rules:
 ┌concept          BuildConfiguration                                                          ┐-->
 │inheritors       false                                                                       │
 │condition        (node, genContext, operationContext)->boolean {                             │
 │                   node.model.roots(BuildConfiguration).configurationItems.                  │
 │                       any({~it => it.isInstanceOf(CallGraphConfigItem); });                 │
 │                 }                                                                            │
 └keep input root  true                                                                        ┘
        map_BuildConfiguration
```

## 9.5    Wrap Up

This finishes our implementation of the callgraph. You can now create some kind of example program (including a `BuildConfiguration` with the `callgraph` item). After rebuilding this example program, you should be able to open the graph viewer and see the new diagram:
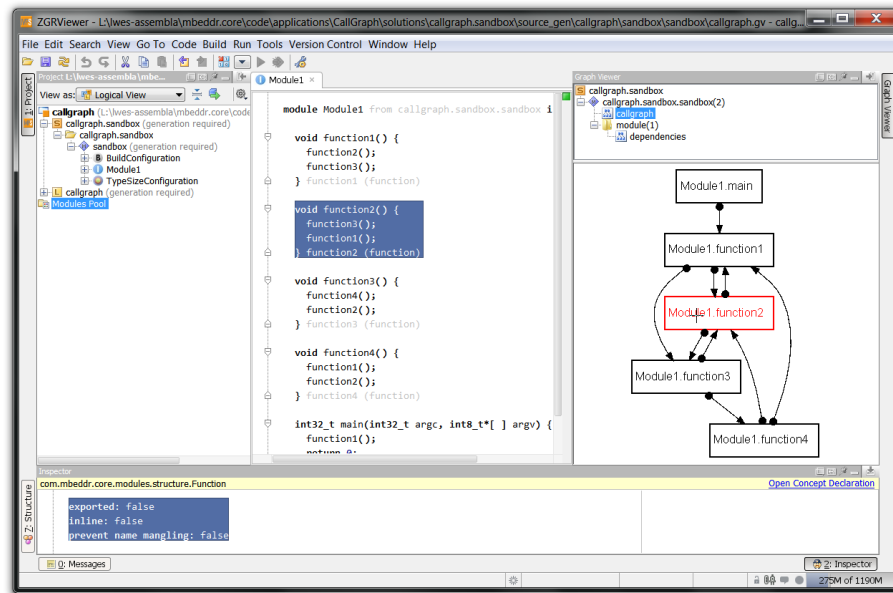


You can click on the boxes and select the function. You can also click on either end of the arrows and highlight the "outgoing" function calls.