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, provides a simple "hello world" tutorial and discusses the differences to regular C. It also discusses all the extensions that are open sourced so far, including components, state machines, exceptions, physical units, requirements tracing and product line variability.

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 out of the box. 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 C 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 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 2.5.x 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 2.5` 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 at least `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`, `gdb`, `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`, `gdb` 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 native code, and Java has to be able to find this native code.

We provide a special plugin which is responsible for loading the native code. This plugin is packaged into a zip file named *spawner* and stored it in the directory `mbeddr.core/tools/spawner`. In order to use the debugger, please unzip this plugin into the `plugins` directory of your MPS installation.

TODO(*How do people get it if they use the downloaded ZIP file? Is it in there?*)

3 Important keyboard shortcuts in MPS

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.

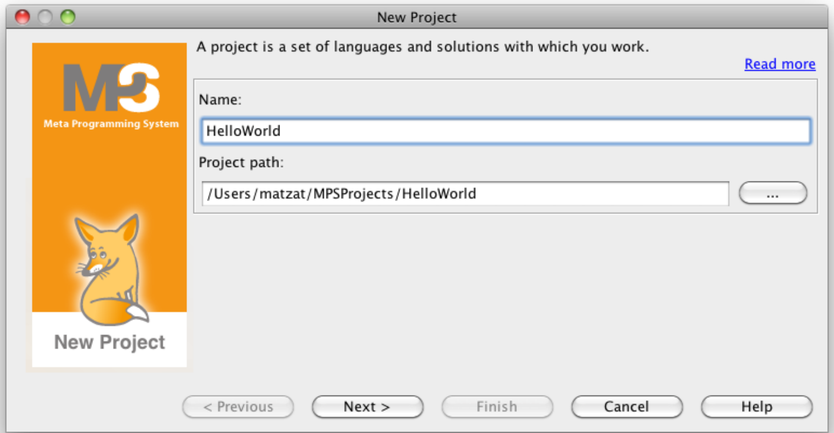
Note: Since MPS 2.5, MPS comes with a productivity guide and actually useful hints and tips at startup. The productivity guide tracks the commands you use and recommends more productive ones. Find it in the `Help` menu.

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.

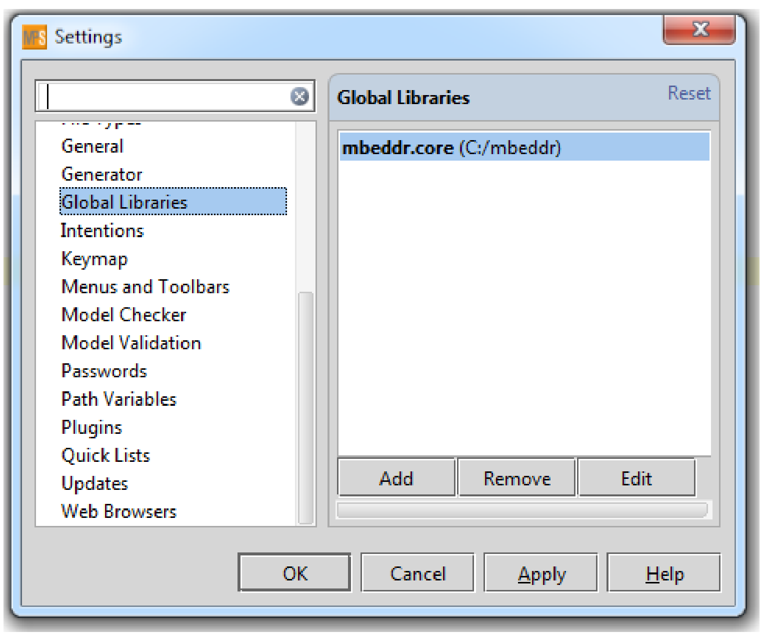
4.1 Create new project

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* → *Settings* and select the *GlobalLibraries* in the IDE settings. Create a library called `mbeddr.core` that points to the root directory of the unzipped mbeddr installation.

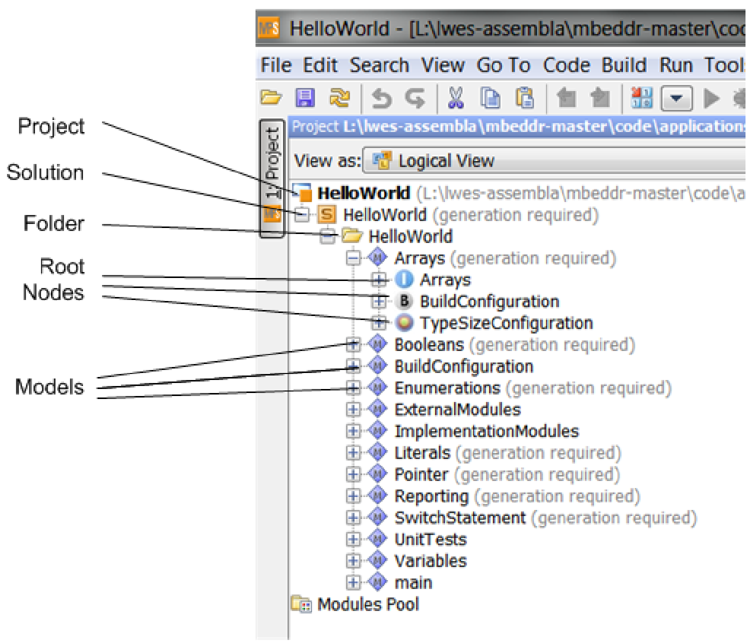
Note: This library must point to the root directory of the checkout so that all languages are below it, including *core* and *mpsutil*.



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

4.2 Project Structure and Settings

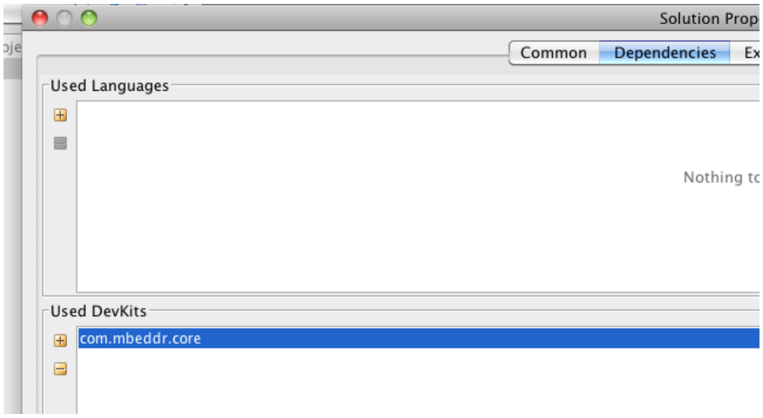
An MPS project is a collection of solutions[4]. A *solution* is an application project that *uses* existing languages. Solutions contain any number of models; models contain root nodes. Physically, models are XML files that store MPS code. They are the relevant version control unit, and the fundamental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



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.

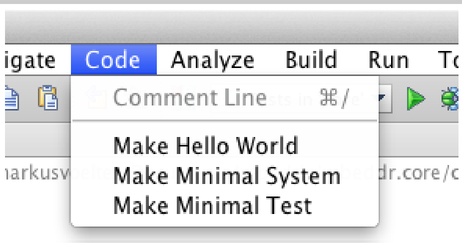
[4] A project can also contain *languages*, but these are only relevant to language implementors. We discuss this aspect of mbeddr in the *Extension Guide*



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

4.3 Create an empty Module

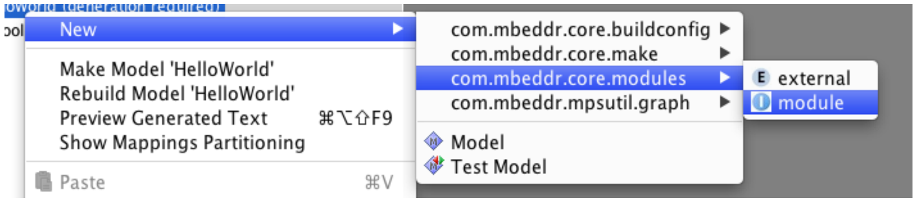
Note: In this tutorial we create the basic building blocks manually. However, there are also a couple of Wizards in the Code menu that create these things manually. See the following figure.



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 create the corresponding `.h` and `.c` files.

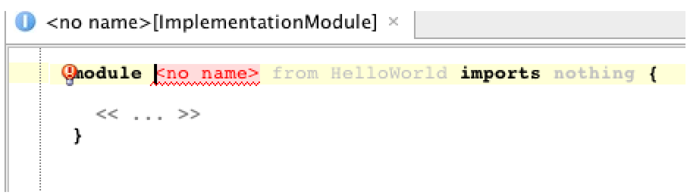
A module can import other modules. The importing module can then access the *exported* contents of imported modules. Module contents can be exported using the `export` intention (available via `Alt-Enter` like any other intention).

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.

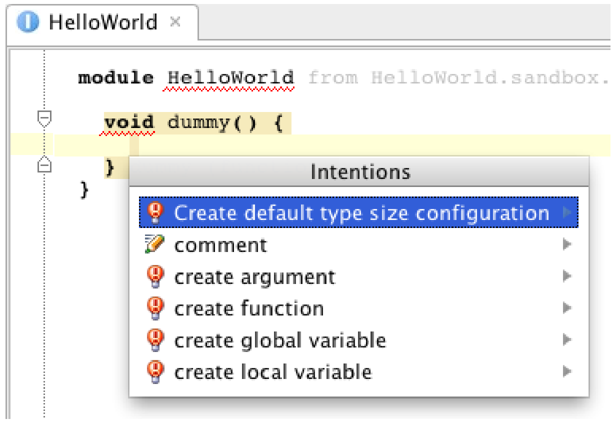
As a result, you will get an empty implementation module. It currently has no name (the name is red and underlined) and only a placeholder `«...»` where top level C constructs such as functions, `struct`s, or `enum`s can be added later.



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. 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, `structs` 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.
- start typing the return type of the function (e.g. `int32_t`) and then enter a name and the opening parentheses[5].
- 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

[5] Entering a type and a name makes it a global variable. As soon as you enter the `(` on the right side of the name, the variable is transformed into a function. This process is called a *right-transformation*.

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:

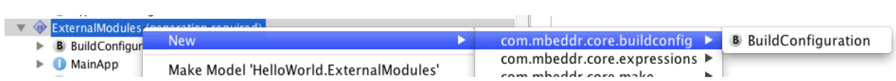
```
message list log {
  INFO hello(string who) active: Hello
}
```

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
  }
}
```

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 figure below).



Initially, it will look as follows:

```
Target Platform:
  <no target>

Configuration Items
  << ... >>

Binaries
  << ... >>
```

You will have to specify three things. 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[6]. 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 language concepts elements are translated. In our case we have to specify the `reporting` configuration. It determines how log messages are output. The `printf` strategy simply prints them to the console, which is fine for our purposes here. Select the placeholder and type `reporting`.

```
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
}
```

[6] Other target platforms may generate build scripts for other build systems.

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 manually, 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 number `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).

Note: That same Hello World example can be created using the $Code \rightarrow MakeHelloWorld$ menu.

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.

TODO(*Use noindent consistently***)**

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 is not exposed to the programmer. 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 existing, non-mbeddr header files that should be accessible 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 imports nothing {

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

module ModuleUsingTheExportedFunction imports ImplementationModule {

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

External modules mbeddr C code must be able to work with existing code and existing C libraries. So to call existing functions or instantiate `struct`s, 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.

Manually entering the contents of the header file into an external module is tedious and error-prone. mbeddr comes with an automatic import for external header files. Since this process is not as trivial as it may seem, we discuss it extensively in Section 10.

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, is generated into a `make` file which is automatically run as part of the MPS build, resulting in the corresponding executables. The generated code, the `make` file and the executables can be found in the `source_gen` folder of the respective solution (this directory can be changed via the `Generator Output Path` property in the solution properties).

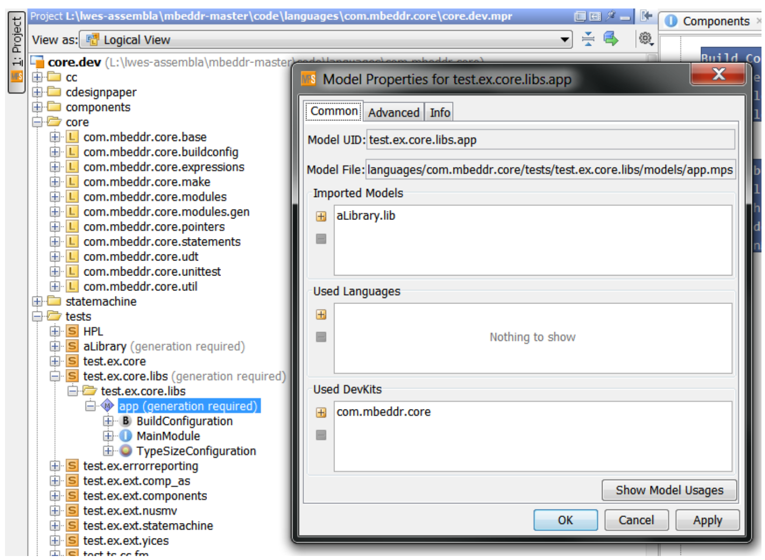
Note: The build language is designed to be extended for integration with other build infrastructures. In that case, other targets (than `desktop`) would be provided by the language that provides integration with a particular build infrastructure.

Libraries Libraries are binaries that are not executable. Specifically, they are `libXXXX.a` files which 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 *must* 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`. 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` and by using the `BCHelper` helper class.
- New platforms can be contributed by extending the `Platform` concept. Users then also have to provide a generator for `BuildConfigurations`.

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 a 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 imports nothing {

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


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

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;
  }
}
```

For executables that contain tests, in the build configuration, the `isTest` 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
```

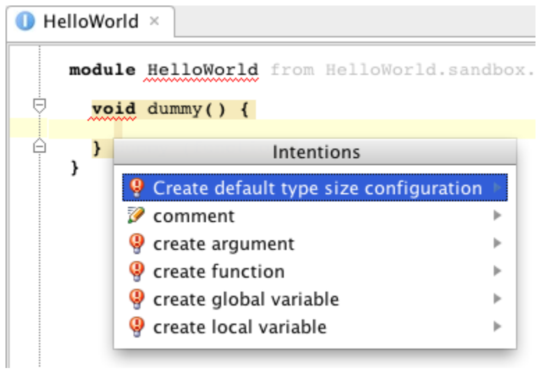
Test cases can of course call arbitrary functions. However, as we have stated earlier, `assert` and `fail` statements must reside in test cases, not in arbitrary functions (this is related to the way the failures are implicitly counted and returned back from a test case). However, functions can be marked as a `test helper` using an intention. `assert` and `fail` can be used within test helper functions. Test helpers must be called *directly* from test cases!

```
exported test case testAddFloat {
  assert(0) 1 + 2 == 3;
  moreStuff(10, 20, 30);
}

test helper
void moreStuff(int8_t x, int8_t y, int8_t z) {
  assert(0) x + y == z;
}
```

TODO(*HERE*)

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. However, 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 size-mapping between the C99 types and the standard primitive types. The `TypeSizeConfiguration` mentioned above can be added with the *Create default type size configuration* (see screenshot below) 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`.



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`

The type size configuration also requires the specification of the size of `size_t` and pointers.

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 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;
  }
}
```

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 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;
  }
}
```

All number literals, including decimal literals are signed by default. A suffix `u` can be added to make them unsigned.

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`, as illustrated in the following code. You can also see the usage of pointer dereference (`*xp`) and assigning an address with `&`.

```
module BasicPointer 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;

  }
  ...
}
```

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 illustrates how to do this. Note that `size_t` is a primitive type, built into mbeddr. It's size is also defined in a `TypeSizeConfiguration`.

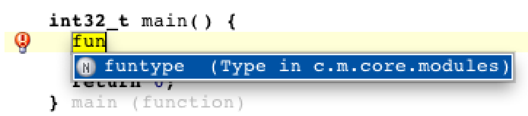
```
external module stdlib resources header : <stdlib.h>
{
  void* malloc(size_t size);
  void free(void* pointer);
}
```

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

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

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. 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 figure below). 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)*.



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`) to distinguish function references 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 imports nothing {

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

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

  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;
  }
}
```

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 imports nothing {

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

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

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

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

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 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;
  }

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

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 enumeration 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

. **TODO(***What is the current state?***)**

```
module EnumerationApp 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;
  }
}
```

5.10 Goto

mbeddr C supports the definition of labels and the `goto` statement. We discourage its use. However, `goto`s are useful for implementing code generators for domain-specific abstractions. This is why they are available.

5.11 Variables

Global variables are identical to regular C. Like all other module contents, they can be `exported`. 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;`).

5.12 Arrays

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

```
module ArrayApplication 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;
  }
}
```

5.13 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 `MessageDefinition`s 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 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;
  }
}
```

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.

5.14 Assembly Code

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

5.15 Comments

In mbeddr we distinguish between commenting out code and adding documentation. The former retains the AST structure of code, but wraps it in a comment. The latter supports adding prose text to program elements.

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.

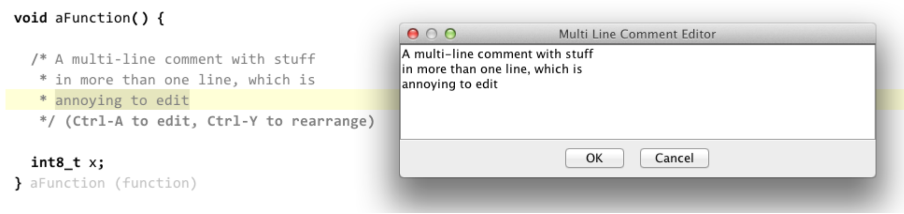
Not all program elements can be commented out (since special support by the language is necessary to make something commentable), only concepts that implement `ICommentable` can be commented. At this time, this is all statements and module contents.

Documentation mbeddr supports comments. There are several kinds of comments that can be used: single line statement comments, multi-line statement comments and element documentations.

Single line statement comments are comments that can be used in statement context. For example, in a function body, you can type `//` and fill in arbitrary text behind. One line only! The following shows as simple comment.

```
void aFunction() {

  // Here is a simple one line comment

  int8_t x;
} aFunction (function)
```

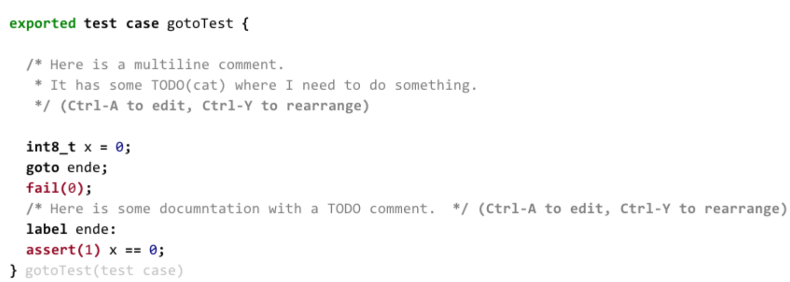
A multi-line comment statement can be created by typing `/*` in statement context. It supports multiple lines. However, since there is no wrapping over into the next line, editing can be cumbersome. To solve this problem you can press `Ctrl-A` anywhere in the multi-line comment to open a dialog with a regular text area. You can edit the text in this text area and when you close it with `OK` the text is transferred back into the actual comment.



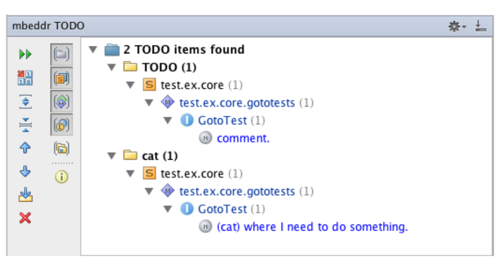
You can also press `Ctrl-Y` on a multi-line comment to automatically rearrange the line contents.

The two statement comments discussed above have two important limitations. The first one is that they are standalone statements and not connected to any program element (other than through their position in the code). Second, they can only be used in statement context. A better solution in many cases is to use the element documentation. It is semantically attached to a program element. To create one, either press `Ctrl-A` or use the `Add Documentation` intention. It looks just like a multi-line comment (and has the same edit dialog for convenient editing), but it is attached to (and hence moves around with) a program element. Currently, all statements and module contents can be annotated with an element documentation.

TODOs mbeddr comes with a special view for collecting and showing TODOs in comments. Anywhere within the comments discussed above, you can write `TODO and then some text` or `TODO(category) and then some text` (see next screenshot):



To find the TODOs anywhere in your project, use the $Tools- > mbeddrTODO$. It shows the TODOs in the form of a Find Usages dialog.



The view has many view options (try them!). The most important one is the top left one. Pressing it enables categories. In this case the TODOs are grouped by what has been specified by `category` in the `TODO(category)` format.

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/MPS2.5/
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:

```
~/Documents/mbeddr/mbeddr.core/code/applications/HelloWorld (master)$ ant
Buildfile: /Users/markusvoelter/Documents/mbeddr/mbeddr.core/code/applications/HelloWorld/build.xml

init:
   [delete] Deleting directory /Users/markusvoelter/temp/mpscache
    [mkdir] Created dir: /Users/markusvoelter/temp/mpscache

build:
[mps.generate] Loaded project MPS Project [file=.../HelloWorld/HelloWorld.mpr]
[mps.generate] Generating in strict mode, parallel generation = on
[mps.generate] Generating:
[mps.generate]     MPS Project [file=.../HelloWorld/HelloWorld.mpr]

BUILD SUCCESSFUL
Total time: 41 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/.MPS25/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.

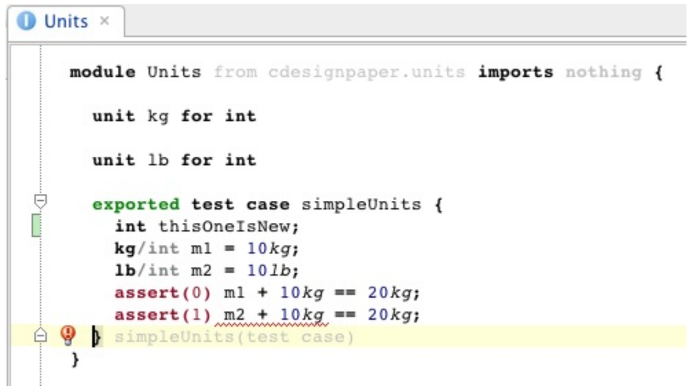


Fig. 1. A new variable has been added to the program and the gutter shows the green markup

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. 1 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. 2) 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.

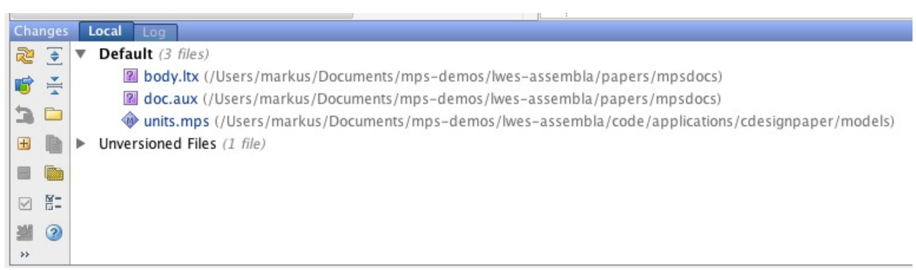


Fig. 2. A new variable has been added to the program and the gutter shows the green markup

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 The Graph Viewer

mbeddr ships with a graph viewer embedded into MPS. It can be used to visualize graphviz files. 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 (explained in Section 11).

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.

9 Debugging

TODO(*Domenik, can you please check this for accuracy?*)

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.

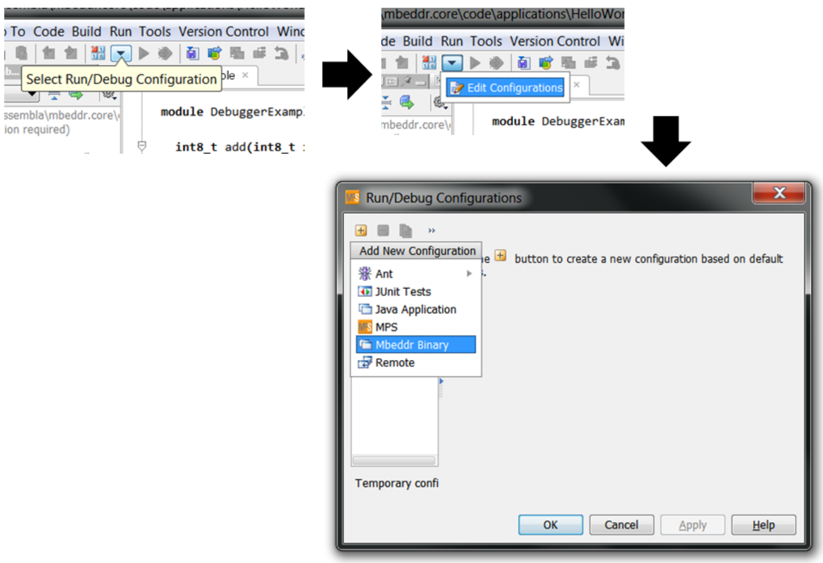
Note: We assume the default configuration, where we use `gdb` as the debug backed. Debugging on some target device is not yet supported.

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.

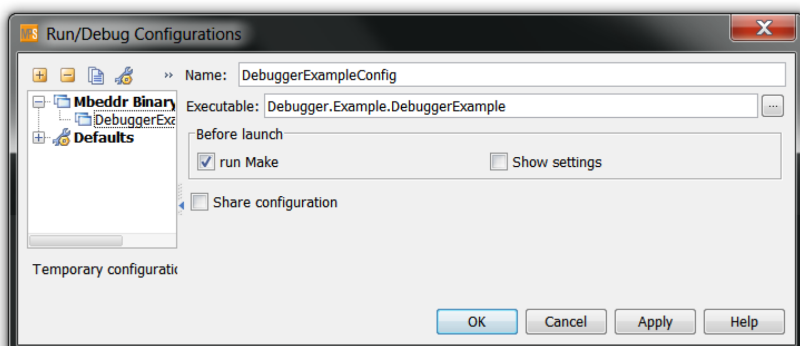
```
module DebuggerExample from Debugger.Example imports nothing {

  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;
  }
}
```

9.1 Creating a Debug Configuration

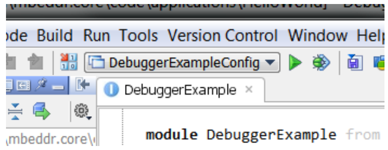
We start out by creating a new debug configuration as shown in the figure below:



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).



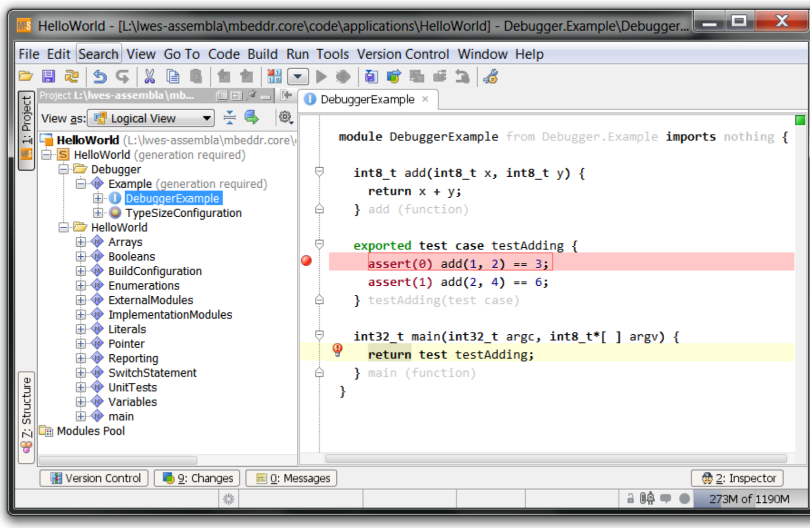
The launch configurations drop down at the top of the MPS screen should now show this new configuration:



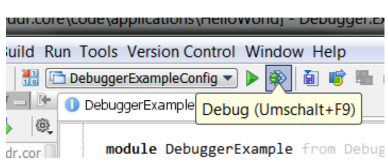
9.2 Running a Debug Session

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`.

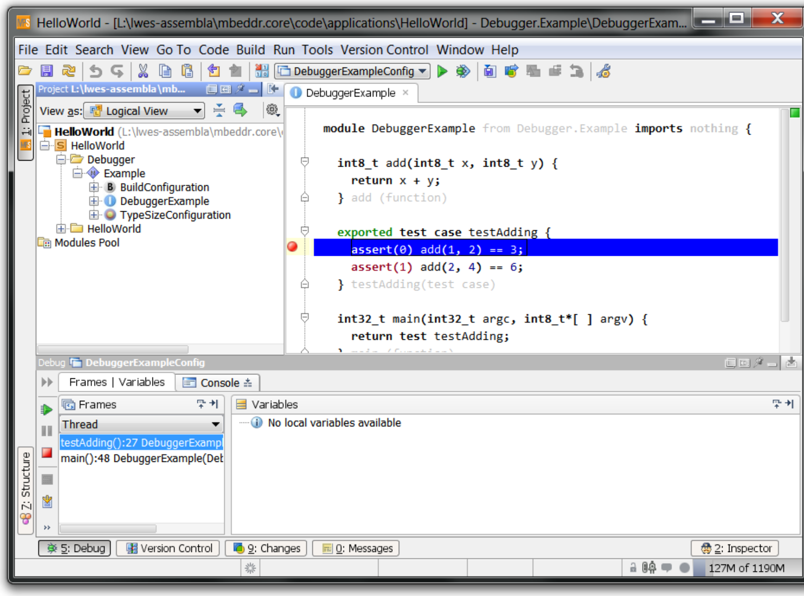
Now set a breakpoint in the first line of the test case. You can set breakpoints by clicking into the gutter of the editor. The result should look like this:



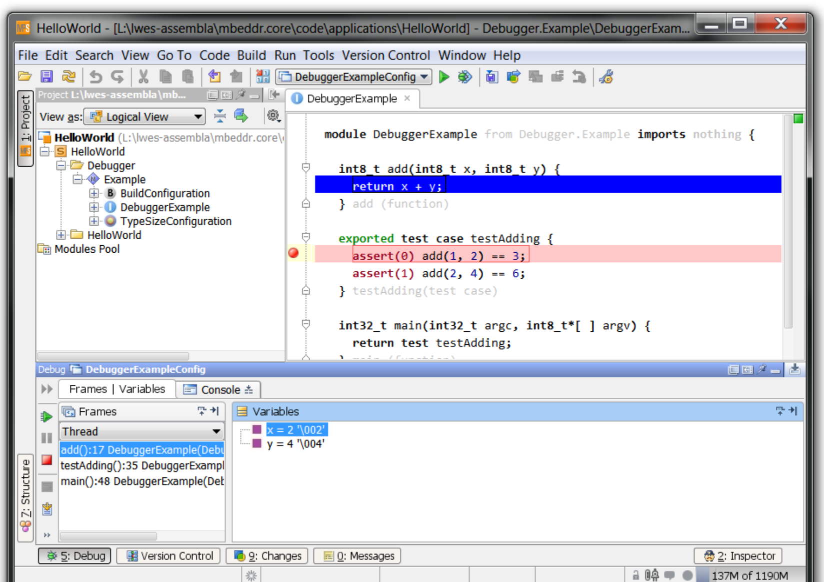
Next, run the previously created debug configuration by pressing `Shift-F9` or by selecting the debugger button in the MPS title bar (see next figure).



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`.



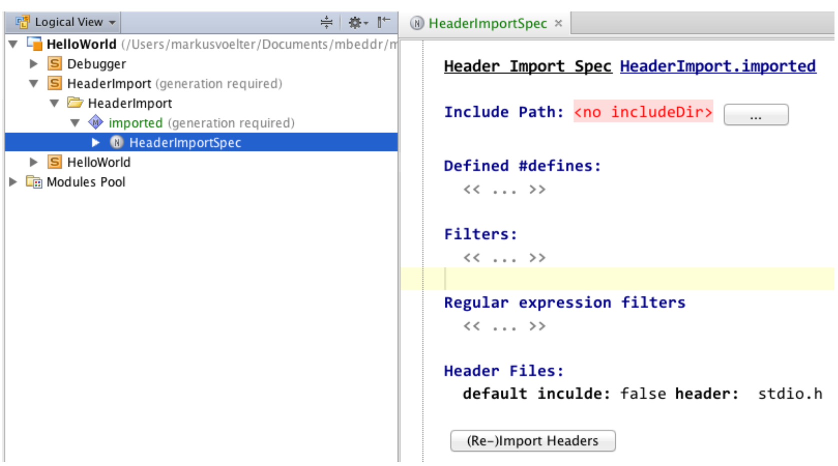
10 Importing existing Header Files

To be able to call into existing libraries you have to be able to access the contents defined in their header files. As we have discussed in Section 5.2, the way to do this is to create an external module. However, typing the contents into MPS is obviously not productive. This section explains how to automatically import them.

Note: MPS provides a built-in way for accessing external code. This mechanism is called stubs. However, we decided not to use it since importing C code is much more sophisticated than Java, and all kinds of configurations have to be performed that cannot be easily integrated into MPS' stub mechanism.

10.1 An Example

In this section we import the `stdio` header file. The code for this example is also in the Hello World project, in the `HeaderImport` solution. We first create a new, empty model. In the model we create a `HeaderImportSpec` object (from the `cstubs` language). Initially it looks as follows:



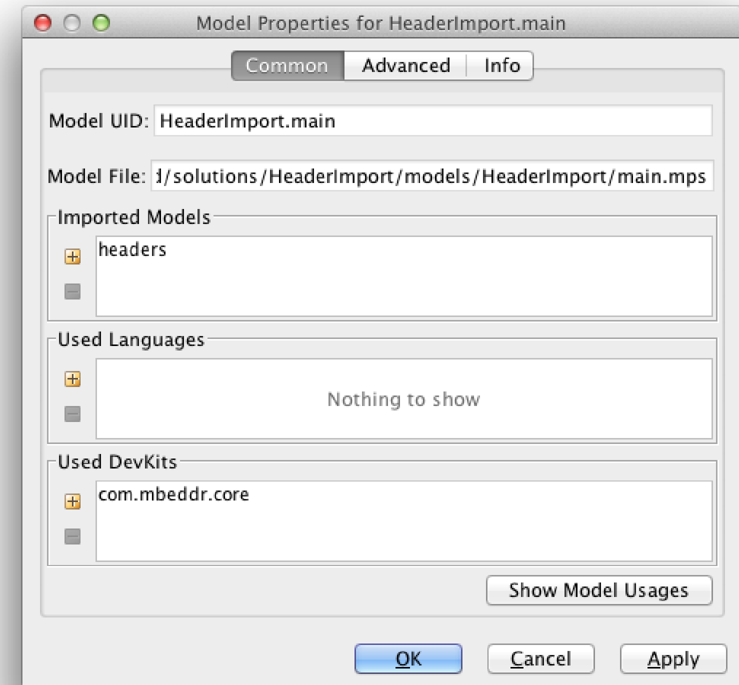
In the `path` property we set the path to a directory that contains the set of header files we want to import. Note that you can use the `...` button to bring up a directory selection dialog. You can also use MPS' path variables in the path specification, using the familiar `${the.var.name}` notation[7].

Once you have specified the path, you can press the `(Re-)import Headers` button. This will import the headers in the specified directory.

If you then open the resulting external module in MPS, you will see a number of errors. They result from the fact that no type size configuration is in you model yet. If you add one, the errors should go away.

[7] Note that the example application ships with `stdio.h` in the solution's `inc` directory.

Let us now create a program that uses the imported header. Create a new model and open its properties. Add the model that contains the imported header files to its `Imported Models` section:



You can now create a minimal program (*Code* → *MakeMinimalSystem*) in the *importing* model. In the `imports` section of the resulting implementation module you can now add `stdio`. In the main function you can now use, for example, `printf`:

```
module UsingIO imports stdio {

  int32_t main(int32_t argc, int8_t*[] argv) {
    printf("Hello World");
    return 0;
  } main (function)
}
```

To make the program run, you also have to add the `stdio` external module to the build configuration:

```
executable UsingIO isTest: false {
  used libraries
    << ... >>
  included modules
    UsingIO (HeaderImport/HeaderImport.main)
    stdio (HeaderImport/HeaderImport.imported)
}
```

You can now rebuild the solution and run the generated `make` file from the command line:

```
HelloWorld/solutions/HeaderImport/source_gen/HeaderImport/main (master)$ make
rm -rf ./bin
mkdir -p ./bin
gcc    -c -o bin/UsingIO.o UsingIO.c -std=c99
gcc -o UsingIO bin/UsingIO.o   -std=c99
```

Finally, you can run `UsingIO`. It should print `Hello World` onto the command line.

Note that this header file was special in the sense that it doesn't require to link some library or object file that contains the implementation for the functions defined in the header file. This is because it is part of the standard library. If you were to import arbitrary other header files, you may have to add a `linkable` to the external module's `resources`.

10.2 Tweaking the Import

Importing header files is not as simple as it may seem initially. In this section we discuss some of the things you can do when importing headers that "don't look right".

Defined #defines : Header files can express product line variability using `ifdef`s. These use preprocessor constants as in `#ifdef SOMETHING`. To import a header file correctly you may have to define a number of these constants. This can be done in this section:

```
Defined #defines:
  #define SOMETHING = <no value>
  #define SOMETHINGELSE = 10
```

Mappings Some header files use platform-specific directives that cannot be parsed by the Eclipse CDT parser that underlies the header file import. These must be removed (or changed) before parsing the file. The header file importer comes with a preprocessor that can remove or change such unparsaeble code[8].

[8] Note that this is not a problem in the final system, since the C code generated from mbeddr will include the actual header file, not a generated version of the external module. So the final system (which is assumed to be processed by a compiler that understands the platform-specific stuff) will see these things unchanged.

Regular Expression Mappings This is the same as in the previous paragraph, except that the a regular expression can be used.

Header Files **TODO**(*somebody explain please what this does.*)

10.3 Limitations

Importing header files is not as simple as it may seem (heard that before :-)?), and our current importer still has some limitations.

#ifdef Variability At this point we cannot yet map product line variability expressed with `#ifdefs` onto the product line variability mechanism of mbeddr. Hence, as discussed above, *only a particular variant* can be imported, which is why we have the *Defined #defines* section above.

Note: Note that we will be working on improving this situation.

Complex Expressions We cannot parse complex expressions used in `#defines` at this point. Assume the following example:

```
#define SOMETHING 10 + SOME_OTHER_DEFINE + 3
```

Since we cannot parse such expressions, we represent them as an opaque string in the external module, like this:

```
exported #define SOMETHING = (void) 10 + SOME_OTHER_DEFINE + 3
```

Since we cannot parse the expression, MPS cannot calculate the type, so `SOMETHING` is typed to be `void`. Of course, if you reference `SOMETHING` from application code, the type check will fail (e.g. if you write `SOMETHING + 3` it won't work since you cannot add `void` to an integer). To solve this problem, you have to change the type manually in the imported external module:

```
exported #define SOMETHING = (int8_t) 10 + SOME_OTHER_DEFINE + 3
```

11 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.

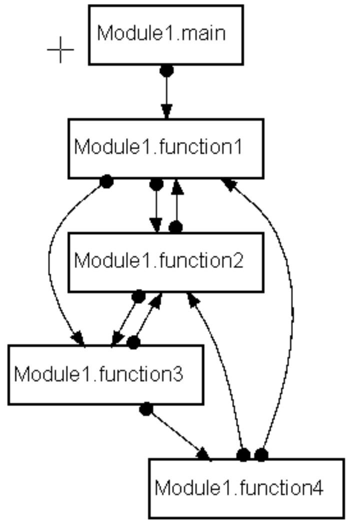
Note: This section assumes that you have a basic understanding of MPS generators.

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:



11.1 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).

11.2 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>;
                }
```

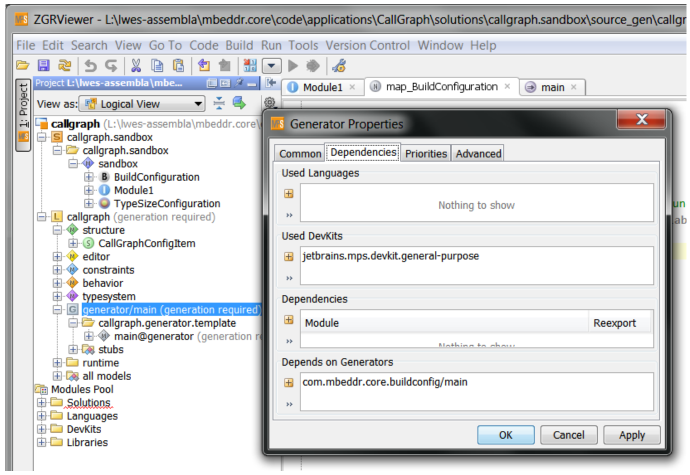
We make the edge bidirectional (<->), we specify the ID to be the ID of the function call node (over which we currently iterate) and then we specify a `dot` tail arrow style and a `normal` head arrow style. Finally, we specify the target; we use the same `function` node defined above as the target, and 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("\\.", "_");
```

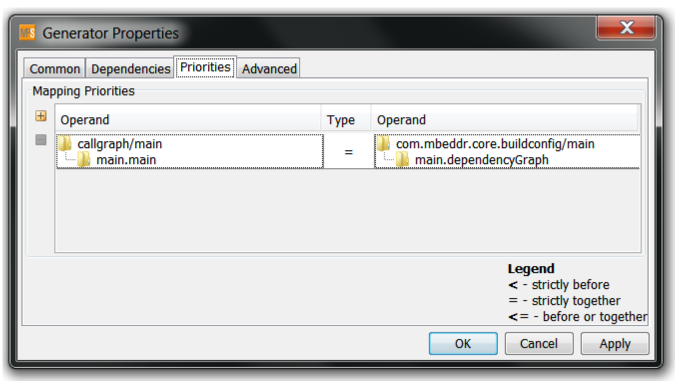
That's all we need to do in the generator template.

11.3 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`.



11.4 Making the Generation Optional

mbeddr comes with a generic configuration framework. `BuildConfigurations` 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`

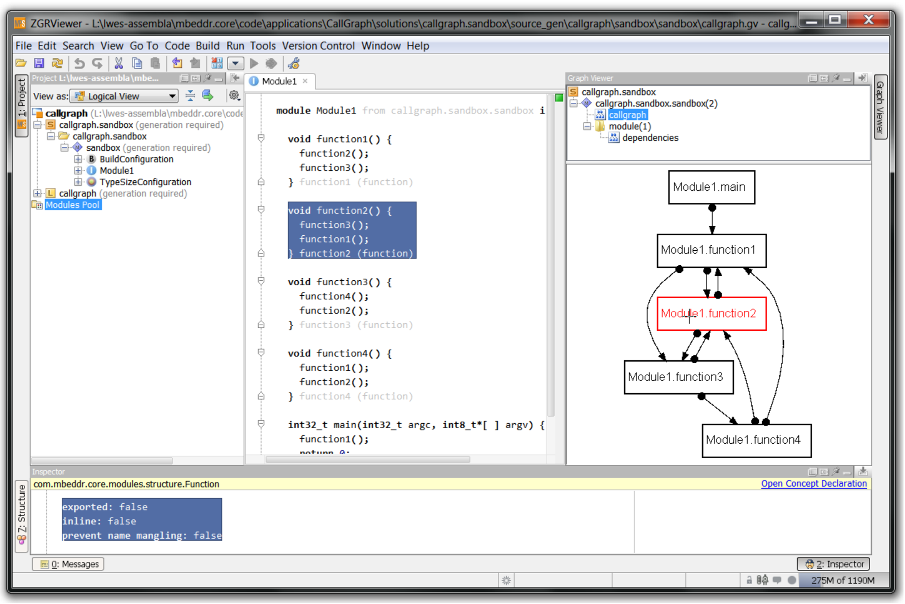
```
concept CallGraphConfigItem extends    BaseConcept
                            implements IConfigurationItem

  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
```

11.5 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.

12 Physical Units

Physical Units are new types that, in addition to specifying their actual data type, also specify a physical unit (see Fig. 3). New literals are introduced to support specifying values for these types that include the physical unit. The typing rules for the existing operators (+, * or >) are overridden to perform the

```
                         -1
derived unit mps = m s      for speed
convertible unit kmh for speed
conversion kmh -> mps = val * 0.27

int8_t/mps/ calculateSpeed(int8_t/m/ length, int8_t/s/ time) {
  int8_t/mps/ s = length / time;
  if ( s > 100 mps ) { s = [100 kmh → mps]; }
  return s;
}
```

Fig. 3. The *units* extension comes with the sevel SI units predefined and lets users define arbitrary derived units (such as the `mps` in the example). It is also possible to defined convertible units that require a numeric conversion factor to get back to SI units. Type checks ensure that the values associated with unit literals use the correct unit and perform unit computations (as in speed equals length divided by time). Errors are recorded if incompatible are used together (e.g. if we were to add length and time). To support this feature, the typing rules for the existing operators (such as `+` or `/`) have to be overridden.

correct type checks for types with units. The type system also performs unit computations to, for example, handle an `speed = length/time` correctly.
To use physical units use the `com.mbeddr.physicalunits` devkit in your model.

12.1 Basic SI Units in C programs

Once the devkit is included, types can be annotated with physical units. We have defined the seven SI units in mbeddr:

```
int8_t/m/ length;
int8_t/s/ time;
```

It is also possible to define composite units. To add additional components press enter after the first one (the `m` in the example below):

```
          -1
int8_t/m s   / speed;
```

To change the exponent, use intentions on the unit. Types with units can also be `typedef`'ed to make using them more convenient:

```
                  -1
typedef int8_t/m s   / as speed_t;
```

If you want to assign a value to the variable, you have to use literals with units, otherwise you will get compile errors:

```
int8_t/m/ length = 3 m;
```

Note that units are computed correctly:

```
speed_t aSpeed = 10 m / 5 s;
speed_t anotherSpeed = 10 m + 5 s; // error; adding apples and oranges
```

Of course, the type system is fully aware of the types and "pulls them through":

```
int8_t/m/ someLength;
int8_t/m/ result = 10 m + someLength;
```

To get values "into" and out of the units world, you can use the `stripunit` and `introduceunit` expressions:

```
int8_t/m/ someLength = 10 m;
int8_t justSomeValue = stripunit[someLength];
int8_t/m/ someLength = introduceunit[justSomeValue -> m];
```

12.2 Derived Units

A derived unit is one that combines several SI units. For example, $N = kg\frac{m}{s^2}$ or $mps = \frac{m}{s}$. Such derived units can be defined with the units extension as well.

To define derived units, create a `UnitContainer` in your model. There you can define derived units (Fig. 4).

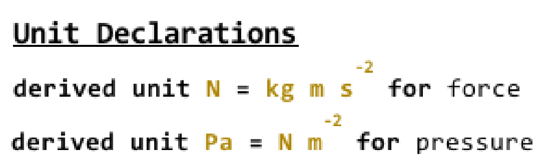


Fig. 4. Derived units are combinations of other units.

Unit computations in C programs work as expected; compatibility is checked by reducing both to-be-compared units to their SI base units.

12.3 Convertible Units

The derived units discussed in the previous section are still within the SI system and require no value conversions. For convertible units, this is different. They can be declared in the unit container as well; but they need conversion rules to be usable (Fig. 5).

```
Unit Declarations
convertible unit F for temperature
convertible unit C for temperature

Conversion Rules
conversion F -> C = val * 9 / 5 + 32
conversion C -> F = (val - 32) * 5 / 9
```

Fig. 5. Convertible units require the definition of conversion rules.

Typically, a convertible unit is a non-SI unit, and the conversion rules map it back to the SI world. Notice how in the conversion rule the `val` keyword represents the value in the original unit.

Conversions in the C programs do not happen automatically, since such a conversion produced runtime overhead. Instead, an explicit conversion has to be used which relies on the conversion rule defined in the units container. A conversion can be added with an intention.

```
int8_t/F/ tempInF = 10 F;
int8_t/C/ tempInC = [tempInF -> C];
```

12.4 Extension with new Units

The units extension can also be "misused" to work with other type annotations such as money, time or coordinate systems. To achieve this, you have to define new elementary unit declarations in a languge extension. Here is some example code with coordinate systems:

```
int8_t/#global/ aGlobalVariable = 10#global;
int8_t/K/ aTemp = aGlobalVariable; // error; #global != K
aGlobalVariable = 10K; // error; #global != K
aGlobalVariable = 230#local; // error; #global != #local
```

In the example above, `#global` and `#local` are two different coordinate systems; technically, both are subtypes of `ElementaryUnitDeclaration`:

```
concept GlobalCoords extends ElementaryUnitDeclaration
                     implements <none>
  concept properties:
    alias = #global
```

Even though these are not technically *convertible* units, we can still define conversion rules:

```
Conversion Rules
  conversion #global -> #local = val + 20
  conversion #local -> #global = val - 20
```

The C code can then use conversions as shown above:

```
int8_t/#local/ aLocalVariable = 20#local;
aGlobalVariable = [aLocalVariable -> #global];
```

The coordinate systems extension is especially useful if combined with a new type and literal, for example, for vectors:

```
intvec/#global/ globalVector = (10,20)#global;
intvec/#local/ localVector = (20,20)#global;
```

13 Components

Modularization supports the divide-and-conquer approach, where a big problem is broken down in to a set of smaller problems that are easier to understand and solve. To make modules reusable in different contexts, modules should define a contract that prescribes how it must be used by client modules. Separating the module contract from the implementation also supports different implementations of the same contract.

Object oriented programming, as well as component-based development exploit this notion. However, C does not support any form of modularization beyond separating sets of functions, `enum`s, `typedef`s etc. into different `.c` and `.h` files. mbeddr, in contrast supports a rich component model.

13.1 Basic Interfaces and Components

To use the components in your C programs, please include the `com.mbeddr.components` devkit.

Interfaces An interface is essentially a set of operation signatures, similar to function prototypes in C. `query` marks functions as not performing any state changes; they are assumed to be invokable any number of times without side effects (something we do not verify automatically at this time).

```
exported interface DriveTrain  {
  void driveForwardFor(uint8_t speed, uint32_t ms)
  void driveBackwardFor(uint8_t speed, uint32_t ms)
  void driveContinouslyForward(uint8_t speed)
  void driveContinouslyBackward(uint8_t speed)
  void stop()
  query uint8_t currentSpeed()
}
```

Components Components can provide and require interfaces via *ports*. A *provided* port means that the component implements the provided interface's operations, and clients can invoke them. These invocations happen via required ports. A *required* port expresses an expectation of a component to be able to call operations on the port's interface. The example below shows a component `RobotChassis` that provides the `DriveTrain` interface shown above, and requires two instances of `EcRobot_Motor`.

```
exported component RobotChassis {

    provides DriveTrain dt
    requires EcRobot_Motor motorLeft
    requires EcRobot_Motor motorRight

    void dt_driveForwardFor(uint8_t speed, uint32_t ms) <- op dt.driveForwardFor {
      motorLeft.set_speed(((int8_t) speed));
      motorRight.set_speed(((int8_t) speed));
      ...
    }
    ...
}
```

Note how the `dt_driveForwardFor` runnable implements the operation `driveForwardFor` from the `dt` provided port. The two signatures are automatically synchronized. Inside components, the operations on required ports can be invoked in the familiar dot notation.

Components can be instantiated. Each component instance generally must get all its required ports connected to provided ports provided by other instances. However, a required port may be marked as `optional` (this is toggled via an intention), in which case, for a given instance, the required port may *not* be connected. Invocations on this required port make no sense in this case, which is why code invoking operations on an optional port must be wrapped in a `with port (optionalReqPort) { .. }` statement. The body of the `with port` is not executed if the port is not connected. The IDE reports an error at editing time if an invocation on an optional port is *not* wrapped this way.

```
exported component RobotChassis {

    provides DriveTrain dt
    requires EcRobot_Motor motorLeft
    requires EcRobot_Motor motorRight
    requires optional ILogging log

    void dt_driveForwardFor(uint8_t speed, uint32_t ms) <- op dt.driveForwardFor {
      with port(log) {
        // other stuff
        log.error(...)
        // more other stuff
      }
    }
    ...
}
```

Components can have fields. These get values for each of the component instances created:

```
exported component OrienterImpl extends nothing {
  ports:
    ...
  contents:
    field int16\_t[5] headingBuffer
    field int8\_t headingIndex

    void orienter\_orientTowards(int16\_t heading, uint8\_t speed, DIRECTION dir) <- ... {
      headingIndex = heading;
    }
}
```

Fields can be marked as `init` fields (via an intention). In this case, when a component is instantiated, a value for the field has to be specified. We will show this below.

We have seen above how a component runnable is tied to the invocation of an operation on a provided port (using `<- op port.operation`). This triggering mechanism can also be used for other events, for example, to react to component instantiation. This effectively supports constructors:

```
exported component OrienterImpl extends nothing {
  contents:
    void init() <- on init {
      compass.initAbsolute();
      compass.heading();
    }
}
```

Instantiation A key difference of mbeddr components compared to C++ classes is that mbeddr component instances are assumed to be allocated and connected during program startup (embedded software typically allocates all memory at program startup to avoid failing during execution), not at arbitrary points in the execution of a program (as in C++ classes). The following piece of code shows an instance configuration:

```
exported instance configuration defaultInstances extends nothing {
  instance RobotChassis chassis
  instance EcRobot_Motor_Impl motorLeft(motorAddress = NXT_PORT_B)
  instance EcRobot_Motor_Impl motorRight(motorAddress = NXT_PORT_C)
  connect chassis.motorLeft to motorLeft.motor
  connect chassis.motorRight to motorRight.motor
}
```

It allocates two instances of the `EcRobot_Motor_Impl` component (each with a different value for its `motorAddress` init parameter) as well as a single instance of `RobotChassis`. The `chassis`' required ports are connected to the provided ports of the two motors. Note that an `instance configuration` just *defines* instances and port connections. The actual *allocation and initialization* of the underlying data structures happens separately in the startup code of the application, for example, in a `main` function:

```
int32_t main(int32_t argc, int8_t*[ ] argv) {
  initialize defaultInstances;
  ...
}
```

To be able to call "into" component instances from regular, non-component C code, adapters can be used. They can be defined inside of instance configurations as well as outside of them. Here is an example:

```
exported instance configuration instances extends nothing {
  instances:
    instance EcRobot_Display_Impl i_display
  connectors:
    ...
  adapter:
    adapt i_display.displayPort as disp
}

void main() {
  initialize instances;
  disp.show("some message");
}
```

Transformation Configuration Components must be able to work potentially with various off-the-shelf middleware solutions such as AUTOSAR. In this case, components will have to be translated differently. Consequently, the transformation for components has to be configured. This happens — like any other configuration — in the `BuildConfiguration`:

```
Configuration Items
  components: no middleware
              wire statically: false
```

The default configuration uses the `no middleware` generator, where components are transformed to plain C function. mbeddr components support polymorphic invocations in the sense that a required port only specifies an *interface*, not the implementing *component*. This way, different implementations can be connected to the same required port (we implement this via a function pointer in the generated C code). This is roughly similar to C++ classes. However, to optimize performance, the generators can also be configured to connect instances statically. In this case, an invocation on a required port is implemented as a direct function call, avoiding additional overhead. This optimization can be performed globally or specifically for a single port. Polymorphism is not supported in this case — users trade flexibility for performance. To do this, select `wire statically: true`. You then have to reference the instance configuration you intend to use:

```
components: no middleware
            wire statically: true instance config: instances
```

If you connect a specific component port to different provided ports in different instances of this component, an error will be reported.

You can also make the decision to wire statically (without polymorphism) on a port-by-port basis, directly in the component (via an intention):

```
requires EcRobot_Compass compass restricted to OrienterImpl.orienter
```

In this case you specify not just the interface but also the particular component and port. This allows the generator to directly refer to the implementing C function — without any overhead.

13.2 Contracts

An additional difference to C++ classes is that mbeddr interfaces support contracts. Operations can specify pre and post conditions, as well as sequencing constraints. Here is the interface from above, but with contract specifications:

```
exported interface DriveTrain {
  void driveForwardFor(uint8_t speed, uint32_t ms)
    pre(0) speed < 100
    post(1) currentSpeed() == 0
    protocol init -> init
  void driveContinouslyForward(uint8_t speed)
    pre(0) speed <= 100
    post(1) currentSpeed() == speed
    protocol init -> forward
  void accelerateBy(uint8_t speed)
    post(1) currentSpeed() == old(currentSpeed()) + speed
    protocol forward -> forward
  query uint8_t currentSpeed()
}
```

The first operation, `driveForwardFor`, requires the `speed` parameter to be below 100. After the operation finishes, `currentSpeed` is zero (notice how the `query` operation `currentSpeed()` is called as part of the post condition). The protocol specifies that, in order to call the operation, the protocol has to be in the `init` state. The post condition for `driveContinouslyForward` expresses that after executing this method the current speed will be the one passed into the operation — in other words, it keeps driving. This is also reflected by the protocol specification which expresses that the protocol will be in the `forward` state. The `accelerateBy` operation can only be called legally while the protocol is in the `forward` state, and it remains in this state. The post condition shows how the value of a `query` operation *before* the execution of the function can be accessed.

The contract is specified on the *interface*. However, the code that checks the contract is generated into the components (i.e. the implementations of the interface operations). The contracts are then checked at runtime.

To add a pre- or postcondition or a protocol, use an intention on the operation. In the inspector, you will have to provide a reference to a message definition that will be `report`ed in case the condition or the protocol fails.

13.3 Mocks

Mocks are used in tests to verify that a component sees a specific behavior at its ports. In mbeddr, a mock verifies the behavior of one specific interface only. Let us look at an example. Here is an interface and a `struct` used by that interface:

```
exported struct DataPacket {
  int8_t size;
  int8_t* data;
};

exported c/s interface PersistenceProvider {
  boolean isReady()
  void store(DataPacket* data)
  void flush()
}
```

This interfaces assumes that clients first call the `isReady` operation, call `store` several times and then call `flush`. We could specify this behavior via contracts shown above. However, we may also want to test that a specific *client* behaves correctly. Here is an example client:

```
exported c/s interface Driver {
  void run()
}

exported component Client extends nothing {
  ports:
    requires PersistenceProvider pers
    provides Driver d
  contents:
    void Driver_run() <- op d.run {
      DataPacket p;
      if ( pers.isReady() ) {
        pers.store(&p);
        pers.flush();
      }
    }
}
```

As we can see, this client does indeed behave correctly. However, if we wanted to write a test to see if it does, we could use a mock to verify it. Here is a mock implememtation for the `PersistenceProvider` interface that verifies this behavior:

```
mock component PersistenceMock {
  report messages: true
  ports:
    provides PersistenceProvider pp
  expectations:
    sequence {
      0: pp.isReady return false;
      1: pp.isReady return true;
      2: pp.store {
          0: parameter data: data != null
        }
      3: pp.flush
    }
    total no. of calls is 4
}
```

It specifies that it expects four invocations in total; the first one should be `isReady` and the mock returns `false`. We then expect `isReady` to be called again, then we expect `store` to be called with a `data` argument not being `null`, and then we expect `flush` to be called.

Here is a test case that uses an instance of the `Client` shown above. It also uses an instance of the `PersistenceMock`:

```
exported test case runTest {
  client.run();
  client.run();
  validate mock mock report messages.mockDidntValidate();
}
```

Notice the `validate mock` statement. If the `mock` instance of `PersistenceMock` has seen anything else but the expected behavior, the `validate mock` statement will fail — and hence the test case will fail.

14 State Machines

State machines can be used to represent state-based behavior in a structural way (they can also be analyzed via model checking, however, we do not discuss this in this section). They are module contents and can be added to arbitrary models. Use the devkit `com.mbeddr.statemachines`.

14.1 Hello State Machine

The following is the simplest possible state machine. It has one state and one in event. Whenever it receives the `reset` event, it goes back to its single state `start`. In other words, it does nothing.

```
statemachine WrappingCounter {
  in events
    reset()
  states ( initial = start )
    state start {
      on reset [ ] -> start {  }
    }
}
```

However, it is a valid state machine and can be used to illustrate some concepts. State machines have in events. These can be "injected" into the state machine from a C program. State machines have one or more states, and one of them must be the initial state. A state may have transitions; a transition reacts to an in event and then points to the new state.

Events can have arguments; they are declared along with the event. Notice how the `int` type requires the specification of bounds. This is to simplify model checking.

```
in events:
  reset()
  increment(int[0..10] delta)
```

A state machine can also have local variables. While these could in principle be handled via separate states, local variables are more scalable.

```
local variables
  int[0..100] current = 0
  int[0..100] LIMIT = 100
  int[0..100] steps = 0
```

We are now ready to write a somewhat more sensible `WrappingCounter` state machine. Whenever we enter the `start` state, we set the `current` variable and the `steps` variable to zero (entry and exit actions can be added to a state via an intention). We have two transitions: if the `increment` event arrives, we go to the `increasing` state, incrementing the `current` value by the `delta`, passed in via the event. Notice how in transition actions we can access the arguments of the event that triggered the respective transition.

```
state start {
  entry {
    current = 0;
    steps = 0;
  }
  on increment [ ] -> increasing { current = current + delta; }
  on reset [ ] -> start {  }
}
```

Let us look at the `increasing` state. There we also react to the `increment` event. But we use guard conditions to determine which transition fires. We can also access event arguments in the guard condition.

```
state increasing {
  entry { steps++; }
  on increment [current + delta <= LIMIT] -> increasing { current = current + delta; }
  on increment [current + delta > LIMIT] -> start { current = 0; }
  on reset [ ] -> start {  }
}
```

14.2 Integrating with C code

State machines can be instantiated. They act as a type, so they can be used in local variables, arguments or in global variables:

```
var WrappingCounter wc;
```

The `trigger` statement is used to "inject" events from regular C code. Notice how we pass in the argument to the `increment` event.

```
void someFunctionCalledByADriver(int8_t ticks) {
  trigger(wc, increment(ticks));
}
```

State machines can also have out events. These are a means to define abstractly some kind of interaction with the environment. Currently, they can be bound to a C function:

```
out events
  wrapped(int[0..100] steps) -> wrapped
```

These out events can then be fired from an action in the state machine, for example, in the exit action of the `increasing` state:

```
exit { send wrapped(steps); }
```

14.3 The complete WrappingCounter state machine

```
module WrappingCounterModule imports nothing {

  statemachine WrappingCounter {
    in events
      increment(int[0..10] delta)
      reset()
    out events
      wrapped(int[0..100] steps) -> wrapped
    local variables
      int[0..100] current = 0
      int[0..100] LIMIT = 100
      int[0..100] steps = 0
    states ( initial = start )
      state start {
        entry {
          current = 0;
          steps = 0;
        }
        on increment [ ] -> increasing { current = current + delta; }
        on reset [ ] -> start {  }
      }
      state increasing {
        entry { steps++; }
```

```
        on increment [current + delta <= LIMIT] -> increasing { current = current + delta; }
        on increment [current + delta > LIMIT] -> start { current = 0; }
        on reset [ ] -> start {  }
        exit { send wrapped(steps); }
      }
  }

  void wrapped(int8\_t steps) {
    // do something
  }

  var WrappingCounter wc;

  void someFunctionCalledByADriver(int8_t ticks) {
    trigger(wc, increment(ticks));
  }
}
```

14.4 Testing State Machines

In addition to model checking (discussed in the chapter on analyses) state machines can also be checked regularly. The following piece of test code checks if the state machine works correctly:

```
exported test case testTheWrapper {
  trigger(wc, reset);
  assert(0) isInState(wc, start);
  trigger(wc, increment(5));
  assert(1) isInState(wc, increasing);
  assert(2) wc.current == 5;
  assert(3) wc.steps == 1;
}
```

Notice that in order to be able to access the variables `current` and `steps` from outside the state machine, these variables have to be marked as `readable` (via an intention).

There is also a shorthand for checking if a state machine reacts correctly to events in terms of the new current state:

```
exported test case testTheWrapper {
  ...
  test statemachine wc {
    reset -> start
    increment(5) -> increasing
    increment(90) -> increasing
    increment(10) -> start
  }
}
```

15 Exceptions

To be documented later.

16 Requirements

16.1 Overview

The requirements package supports the collection of requirements and traceability from arbitrary code back to the requirements.

16.2 Specifying Requirements

Requirements can be collected in instances of `RequirementsModule`, a root concept defined by the `com.mbeddr.cc.requirements` language. An example is shown in Fig. 16.2. Each requirement has an ID, a short prose summary, and a kind. (`functional`, `timing`). The kind, however, is more than just a text; each kind comes with its own additional specifications. For example, a `timing` requirement requires users to enter a `timing` specification. This way, additional formal data can be collected for each kind of requirement.

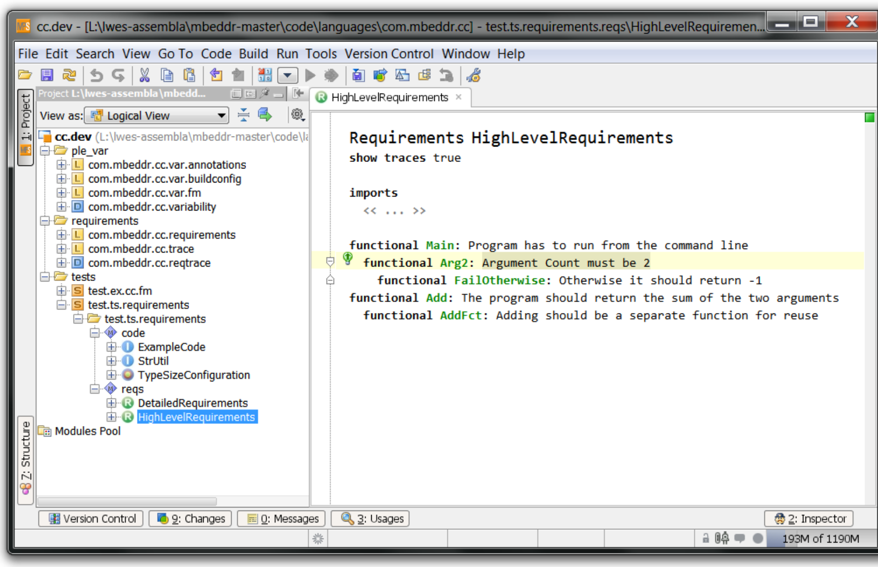


Fig. 6. Working with Requirements

In addition to the summary information discussed above, requirements can also contains details. The details editor can be opened on a requirement with an intention or with `Ctrl-Shift-D`. Fig. 16.2 shows an example.

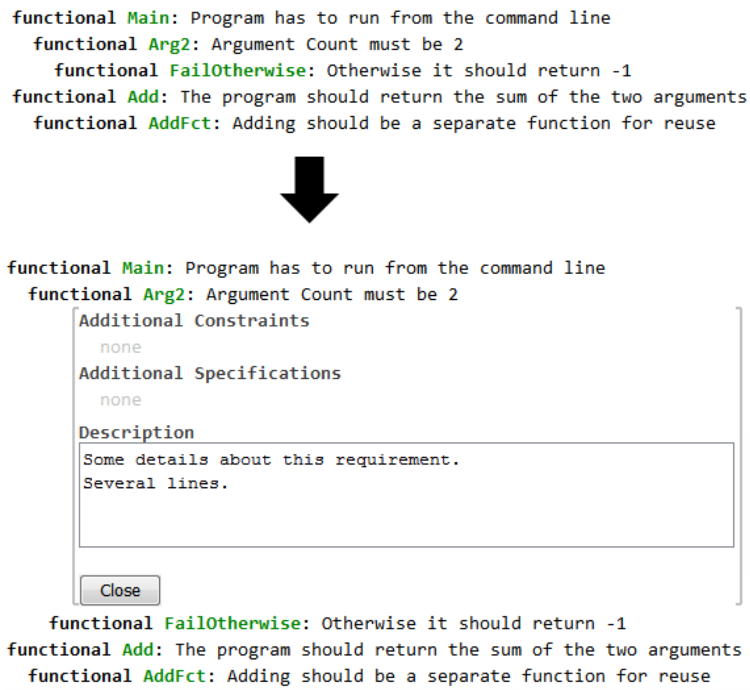


Fig. 7. Requirements editor expanded

In the details, a requirement can be described with additional prose, with the kind-specific formal descriptions as well as with additional constraints among requirements. The default hierarchical structure represents refinement: child requirements refine the parent requirements. In addition, each requirement can have typed links relative to other requirement, such as the `conflicts with` shown in Fig. 16.2.

Requirements modules can import other reqirements modules using the `import` section. This way, large sets of requirements can be modularized.

Extending the Requirements Language To extend the requirements framework, create a new language that extends `com.mbeddr.cc.requirements`. Then, use this language in the project that manages your requirements.

A new Link To create a new link, create a concept that extends `RequirementsLink`. It base class already comes with a pointer to the target requirement. Just define the concept and an alias.

A new Kind To create a new requirements kind, extend `RequirementsKind` and define an alias.

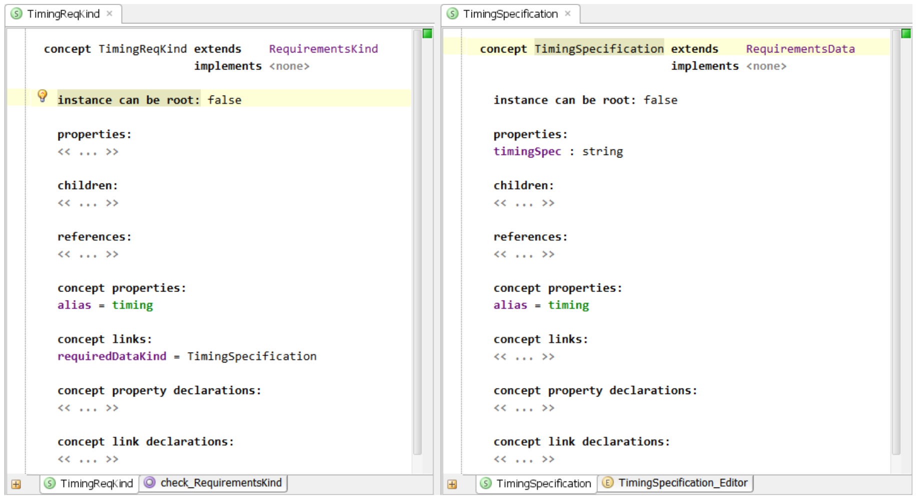


Fig. 8. Defining additional required specifications for a requirements kind.

A new Additional Specification Defining new additional specifications (such as the `timing` specification mentioned above) happens in two steps. First you have to create a new concept that extends `RequirementsData`. It should contain any additional structure you need (this can be a complete MPS DSL, or just a set of pointers to other nodes). You should also define an alias. The second step requires enforcing that a certain requirements kind also requires that particular additional specification. In the respective kind, use the `requiredDataKind` concept link to point to the concept whose instance is required. Fig. 16.2 shows an example.

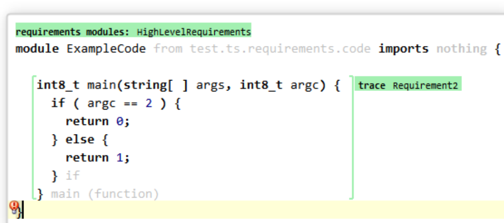


Fig. 9. Code with requirements traces

16.3 Tracing

Tracing establishes links between implementation artifacts (i.e. arbitrary MPS nodes) and requirements. The trace facilities are implemented in the `com.mbeddr.cc.trace` language. Fig. 16.2 shows an example of requirements traces.

Traces can be attached to any MPS node using an intention. However, for this to work, the root owning the current node has to have a reference to a requirements module. This can be added using an intention. Only the requirements in the referenced modules can be referred to from a trace.

There is a second way to attach a trace to a program element: go to the target requirement and copy it (`Ctrl-C`). Then select one or more program nodes and press `Ctrl-Shift-R`. This will attach a trace from each of these elements to the copied requirement.

A trace can have a kind. By default, the kind is `trace`. However, the kind can be changed (`Ctrl-Space` on the `trace` keyword.)

Extending the Trace Facilities New trace kinds can be added by creating a new language that extends `com.mbeddr.cc.trace`, using that language from your application code, and defining a new concept that extends `TraceKind`.

16.4 Other Traceables

The tracing framework cannot just trace to requirements, but to any concept that implements `ITraceTarget`. For example, a functional model may be used as a trace target for implementation artifacts. In this section we explain briefly how new trace targets can be implemented. We suggest you also take a look at implementation of `com.mbeddr.cc.requirements`, since this uses the same facilities.

- Concepts that should act as a trace target must implement the `ITraceTarget` interface (e.g. `Requirement`)
- The root concept that contains the `ITraceTarget`s must implement the `ITraceTargetProvider` and implement the method `allTraceTargets`.
- In addition, you have to create a concept that extends `TraceTargetProvider-RefAttr`. It references `ITraceTargetProvider`s.

Here is how the whole system works: You attach a `TraceAnnotation` to a program element. It contains a set of `TranceTargetRef`s which in turn reference `ITraceTarget`s. To find the candidate trace targets, the scoping rule of `TranceTargetRef` ascends the tree to the current root and checks if it has something in the `traceTargetProviderAttr` attribute. That would have to be a subtype of `TraceTargetProviderRefAttr`. It then follows the `refs` references to a set of `ITraceTargetProvider` and asks those for the candidate `ITraceTarget`.

16.5 Evaluating the Traces in Reverse

The traces can be evaluated in reverse order. For example, Fig. 16.5 (left) shows how requirements can be color-coded to reflect their state. The color codes must be updated explicitly (may take a while) by the `Update Trace Stats` intention on the requirement module.

In addition, MPS Find Usages functionality has been enhanced for requirements. If the user executes Find Usages for requirements, the various kinds of traces are listed separately in the result (Fig. 16.5, right).

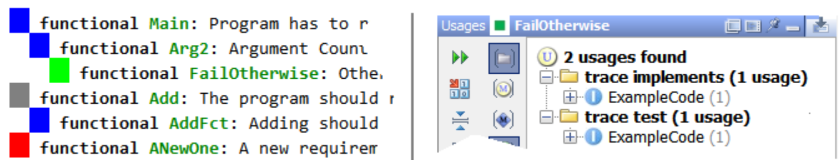


Fig. 10. Left: The requirements can be color coded to reflect whether they are traced at all (grey), implemented (blue) and testes (green). Untraced requirements are red. **Right:** The Find Usages dialog shows the different kinds of traces as separate categories.

17 Variability

17.1 Overview

Product line engineering involves the coordinated construction of several related, but different products. Each product is typically referred to as a *variant*. The product variants within a product line have a lot in common, but also exhibit a set of well-defined differences. Managing these differences over the sets of products in a product line is non trivial. This document explains how to do it in the context of mbeddr.

A devkit `com.mbeddr.cc.variability` is defined that comprises the following three languages:

- `com.mbeddr.cc.var.fm` supports the definition of feature models. Feature models are a well-known formalism for expressing variability on a high-level, independent of the realization of the variability in software.
- `com.mbeddr.cc.var.annotations` allows the connection of implementation artifacts (any MPS model) to feature models as a way of mapping the high-level variability to implementation code. This is done by attaching presence conditions to program elements (this is mbeddr's replacement for `#ifdefs`).
- `com.mbeddr.cc.var.buildconfig` ties the processing of annotations into mbeddr's build process.

17.2 Feature Models and Configurations

Defining a Feature Model Feature models express configuration options and the constraints between them. They are usually represented with a graphical notation as shown in Fig. 17.2.

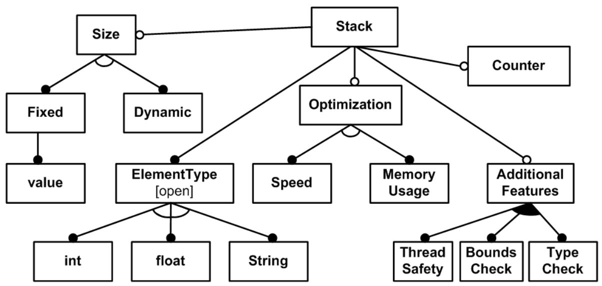


Fig. 11. An example feature model

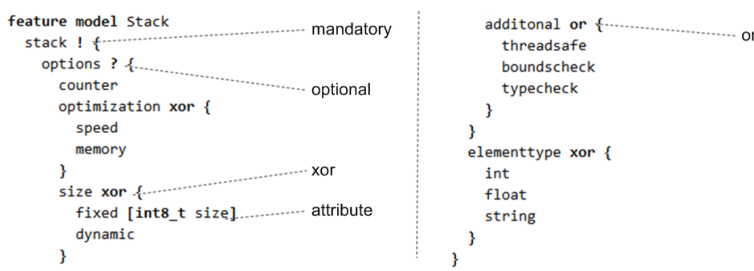


Fig. 12. An example feature model in mbeddr

The following four kinds of constraints are supported between the features in a feature model:

- **mandatory**, the filled circle: mandatory features have to be in each product variant. In the above example, each `Stack` has to have the feature `ElementType`.
- **optional**, the hollow circle: optional features may or may not be in a product variant. In the example, `Counter` and `Optimization` are examples of optional features.
- **or**, the filled arc: a product variant may include zero, one or any number of the features grouped into a an or group. For example, a product may include any number of features from `ThreadSafety`, `BoundsCheck` and `TypeCheck`.

– **xor**, the hollow arc: a product variant must include exactly one of the features grouped into a xor group. In the example, the `ElementType` must either `int`, `float`, or `String`.

Fig. 17.2 shows the textual notation for feature models used in mbeddr. Note how the constraint affects all children! We had to introduce the intermediate feature `options` to separate the mandatory stuff from the optional stuff. Features can have configuration attributes (of any type!). Children and attributes can be added to a feature via an intention. You can also use a surround intention to wrap a new feature around an existing one.

Defining Configurations The point of a feature model is to define and constrain the configuration space of a product. If a product configuration would just be expressed by a bunch of boolean variables, the configuration space would grow quickly, with 2^n, where n is the number of boolean config switches. With feature models, constraints are expressed over the features, defining what are valid configurations. This limits the space explosion and allows interesting analyses that will be provided in later releases.

Let us now look at how to define a product variant as a set of selected features. Fig. 17.2 shows two examples:

```
configuration model SimpleStack configures Stack
  stack {
    options {
      counter
      size {
        fixed [size = 10]
      }
    }
    elementtype {
      int
    }
  }
```

```
configuration model DynamicStack configures Stack
  stack {
    options {
      optimization {
        speed
      }
      size {
        dynamic
      }
      additonal {
        boundscheck
        threadsafe
      }
    }
    elementtype {
      float
    }
  }
```

Fig. 13. Two valid configurations of the feature model

Note that, if you create invalid configurations by selecting feature combinations that are prohibited by the constraints expressed in the feature model, errors will be shown.

Feature models and configurations can be defined in a root concept called `VariabilitySupport`. It lives in the `com.mbeddr.cc.var.fm` language.

17.3 Presence Conditions

A presence condition is an annotation on a program element that specifies, under which conditions the program element is part of a product variant. To do so, the presence condition contains a boolean expression over the configuration features.

For example the two `report` statements and the message list in Fig. 17.3 are only in the program, if the `logging` feature is selected. `logging` is a feature defined in the feature model `FM` that is referenced by this root node.

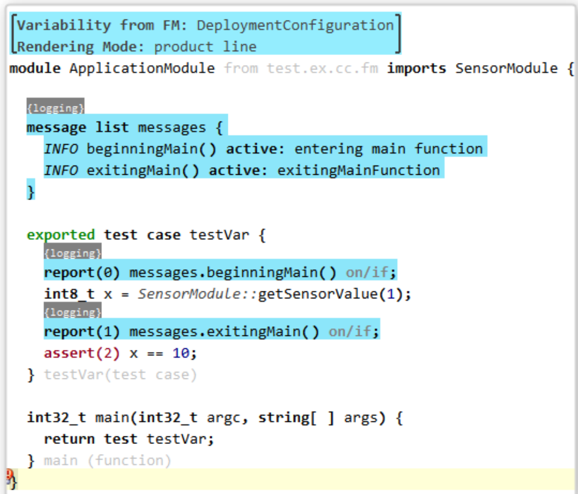


Fig. 14. C program code with presence conditions

To use presence conditions, the root note (here: an implementation module) as to have a `FeatureModelConfiguration` annotation. It can be added via an intention if the `com.mbeddr.cc.var.annotations` language is used in the respective model. The annotation points to the feature model whose features the respective presence conditions should be able to reference. The configuration and the presence conditions are attached via intentions.

An existing presence condition can be *pulled up* to a suitable parent element by pressing `Ctrl-Shift-P` on the presence condition.

The background color of an annotated note is computed from the expression. Several annotated nodes that use the same expression will have the same color (an idea borrowed from Christian Kaestner's CIDE).

17.4 Replacements

A presence condition is basically like an `#ifdef`: the node to which it is attached will *not* be in the resulting system if the presence condition is *false*. But sometimes you want to *replace* something with something else if a certain feature condition is met. You can use replacements for that. Fig. 17.4 shows an example.

A replacement replaces the node to which it is attached with an alternative node if the condition is *true*. In the example in Fig. 17.4 the function call and the `10` are both replaced with a `42`. Note that you'll get an error if you try to replace a node with something that is not structurally compatible, or has the wrong type.

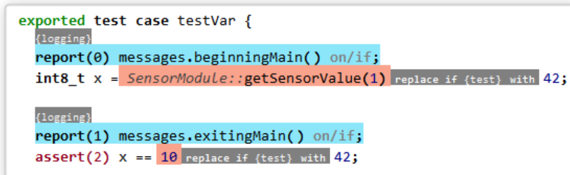


Fig. 15. C program code with a conditional replacement

17.5 Attribute Injection

We have seen that features can have attributes and configurations specify values for these attributes. These values can be injected into programs. The attributes of those features that are used in ancestors of the current node are in scope and can be used. A type check is performed and errors are reported if the type is not compatible.

Note: At this time, this can only be done for expressions. This may be generalized later.

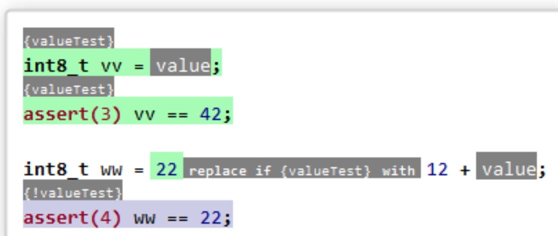


Fig. 16. C program code with an attribute injection (`value` is the name of an attribute of the `valueTest` feature)

17.6 Projection Magic

It is possible to show and edit the program as a product line (with the annotations), undecorated (with no annotations) as well as in a given variant. Fig. 17.6 shows an example. Note that due to a limitation in MPS, it is currently not possible to show the values of attributes directly in the program in variant mode. The projection mode can be changed in the configuration annotation on the root node.

17.7 Building Variable Systems

Building variable systems is a little bit tricky. The problem is that you will want to build different variants at the same time. To make the C build simple, each variant should live in its own directory. If you want to generate into different

Fig. 17. C program rendered in product line mode and in two variants

directories with MPS, you need different models. This results in the following setup:

- You create one or more models that contains your product line artifacts, i.e. the program code, the feature models and the configurations.
- Then, for each variant you want to build, you create yet another model. This model imports the product line model (the one above). In this model you will have a build configuration that determines the variant that will be built.

Figure Fig. 17.7 shows an example of such as setup with one product line model (`fm`) and two variant models `v_debug` and `v_production`.

Each of the product models contains a `BuildConfiguration` that specifies the structure of the binary. In addition, the `BuildConfiguration` has an `VariabilityTransformationConfiguration` attached to it. It determines which configuration should be used for each feature model (`Debug` in the example). You can attach on of these to a build config using an intention, but you need the `com.mbeddr.cc.var.buildconfig` language for that.

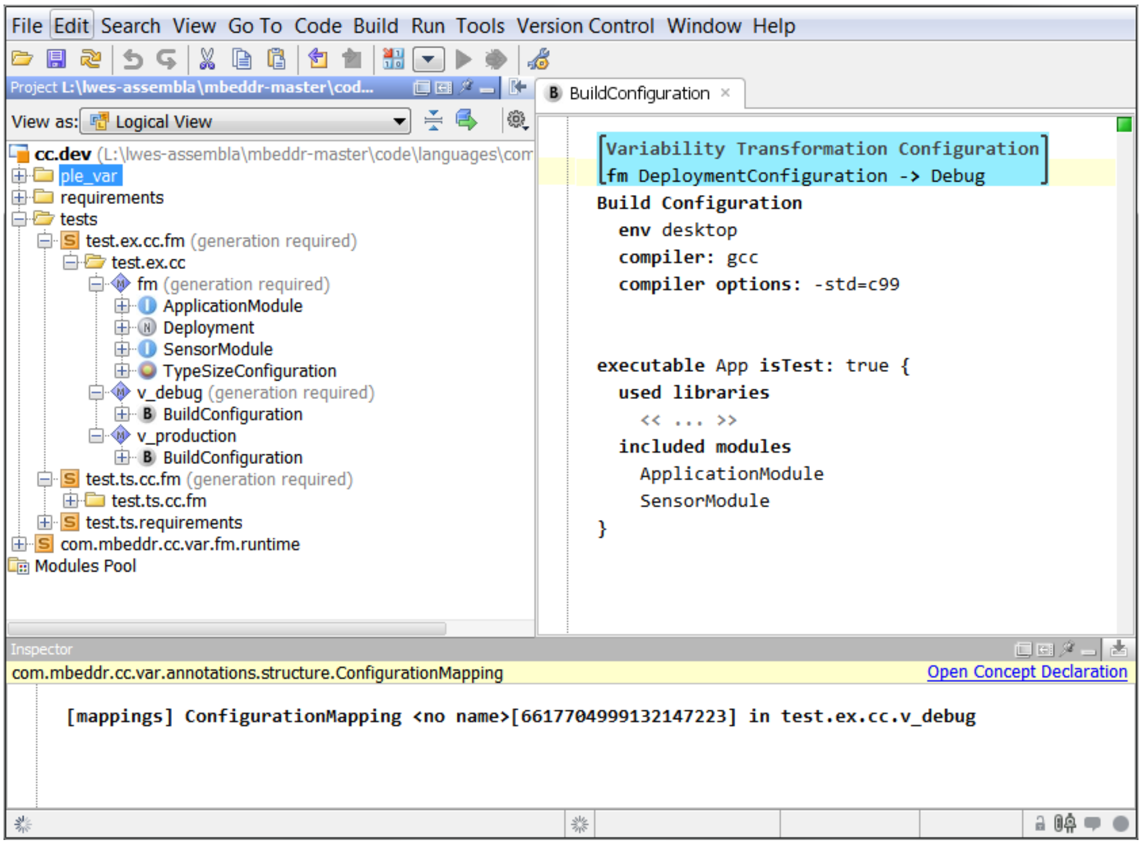


Fig. 18. Model setup and transformation configuration for generating several variants at the same time.