mbeddr C User Guide

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**.

Contents

Part I

Introduction

1 About this Document

1.1 Who should read this Document

This document is targetted at embedded software developers who want to get started with mbeddr. It provides a hands-on introduction to all aspects of mbeddr. The document does not (yet) discuss building custom extensions, it just discusses how to work with the existing ones.

1.2 Structure of this Document

■ **Part I** starts with an introduction to the idea of and concepts behind mbeddr in Section 2. We also discuss how mbeddr compares to other embedded software development approaches, and discuss when to use mbeddr. You may skip this section if you already get the idea of mbeddr.

■ **Part II** contains a comprehensive tutorial. It starts with a guide to installing mbeddr, provides some tipps about using MPS and then runs through a comprehensive tutorial of using mbeddr. It starts with a Hello World and then adds elements from most of the mbeddr default extensions. If you are new to mbeddr and want to understand what it feels like to use mbeddr, this is the part to work through. This part also contains an set of extensive examples or *extending* mbeddr with new languages or C extensions.

■ **Part III** is the reference documentation. The part has a chapter for each of the major features of mbeddr. In contrast to the tutorial it tried so be complete. It also discusses the differences of mebddr C with regular C. This part is indended to provide a solid reference for all existing extensions. Use this part to look up details when you get stuck.

1.3 Feedback, Support and Contact

While mbeddr is mature enough to be used for real-world software development, it is still new and rough at the edges. To get support in case you run into problems, or if you just share thoughts with the mbeddr developers, we have create a Google Group:

https://groups.google.com/forum/?fromgroups#!forum/mbeddr-discuss

You can also submit issues at the github issue tracker:

https://github.com/mbeddr/mbeddr.core/issues

2 Overview over mbeddr

2.1 Introduction and Motivation for mbeddr

Embedded software is software that is embedded in some kind of mechanical or electronic device, often controlling most of the functionality of that device. Today, embedded software is one of the main innovation drivers and differentiation factors in many kinds of products [7].

Embedded systems are highly diverse, ranging from rather small systems such as refrigerators, vending machines or intelligent sensors over building automation to highly complex and distributed systems such as aerospace or automotive control systems. This diversity is also reflected in the constraints on their respective software development approaches and cost models. For example, flight control software is developed over many years, has a large budget, an expert team and emphasizes safety and reliability. The less sophisticated kinds of embedded systems mentioned above are developed in a few months, often with severe budget constraints and by smaller teams. The requirements for safety and reliability are much less pronounced.

■ **Today's Development Tools** The tools used to develop these systems reflect these differences. Highly safety-critical systems are often developed with tools such as SCADE[1]. Systems that are based on a standardized architecture or middleware, such as AUTOSAR in the automotive domain, are often developed with tools that are specific to the standard (such as Artop[2]). Model-based development and automatic code generation is particularly well suited for systems that are highly structured in terms of state-based behavior or control algorithms. Tools like ASCET-SD[3] or Simulink[4] provide suitable predefined, high-level abstractions (e.g. state machines or data flow diagrams). Using higher-level abstractions leads to more concise programs and simplified fault detection using static analysis and model checking (for example using the Simulink

[1] http://www.esterel-technologies.com/products/scade-suite/
[2] http://www.artop.org/
[3] http://www.etas.com/
[4] http://www.mathworks.com/products/simulink

Design Verifier[5]).

However, the state of the practice [9] is that 80% of companies implement embedded software in C, particularly systems that are *not* safety-critical or implement control algorithms. While C is good at expressing low-level algorithms and produces efficient binaries, its limited support for defining custom abstractions can lead to code that is hard to understand, maintain and extend.

Empirical studies found out that there is a need for tools that are more specific for an application domain yet flexible enough to allow adaptation [11, 13]. Domain-specific languages (DSLs) are increasingly used for embedded software [2, 12, 1]. Studies such as [5] and [13] show that DSLs substantially increase productivity in embedded software development. Examples include Feldspar, [2] a DSL embedded in Haskell for digital signal processing; Hume [12], a DSL for real-time embedded systems as well as [10], which uses DSLs for addressing quality of service concerns in middleware for distributed real-time systems. All these DSLs are *external* DSLs. While they typically generate C code, the DSL program is not syntactically integrated with C. This is useful for some cases, but it is a limitation for others. Extending C to adapt it to a particular problem domain is not new: [15] describes an extension of C for real time applications, [4] proposes an extension for reactive systems, [3] describes an extension for shared memory parallel systems. However, these are all *specific* extensions of C, typically created by invasively changing the C grammar, and they typically do not include IDE support.

■ **The mbeddr Approach** mbeddr is fundamentally different. While it builds heavily on domain-specific abstractions, mbeddr is an *open framework* and tool for defining *modular* extensions of C, based on the underlying MPS language workbench (Fig. 2.1 shows an example). In contrast to essentially all other embedded development tools we are aware of, third parties can use *the same* mechanisms for building their own extensions that were used to implement C and the existing extensions. Third parties are *not* at a disadvantage from having to use limited second-class language extension constructs. This is a fundamental shift in the design of tools.

mbeddr also directly integrates formal analyses based on the domain-specific extensions. Even though formal analysis tools for C programs exist (e. g., deductive verification and abstract interpretation with different plugins for Frama-C[6]; model checkers like SLAM[7] and BLAST[8], or commercial tools such as the Escher C Verifier[9] or Klocwork[10]), they

[5] http://www.mathworks.com/products/sldesignverifier
[6] http://frama-c.com
[7] http://research.microsoft.com/en-us/projects/slam/
[8] http://mtc.epfl.ch/software-tools/blast/index-epfl.php
[9] http://www.eschertech.com/products/ecv.php
[10] http://www.klocwork.com/

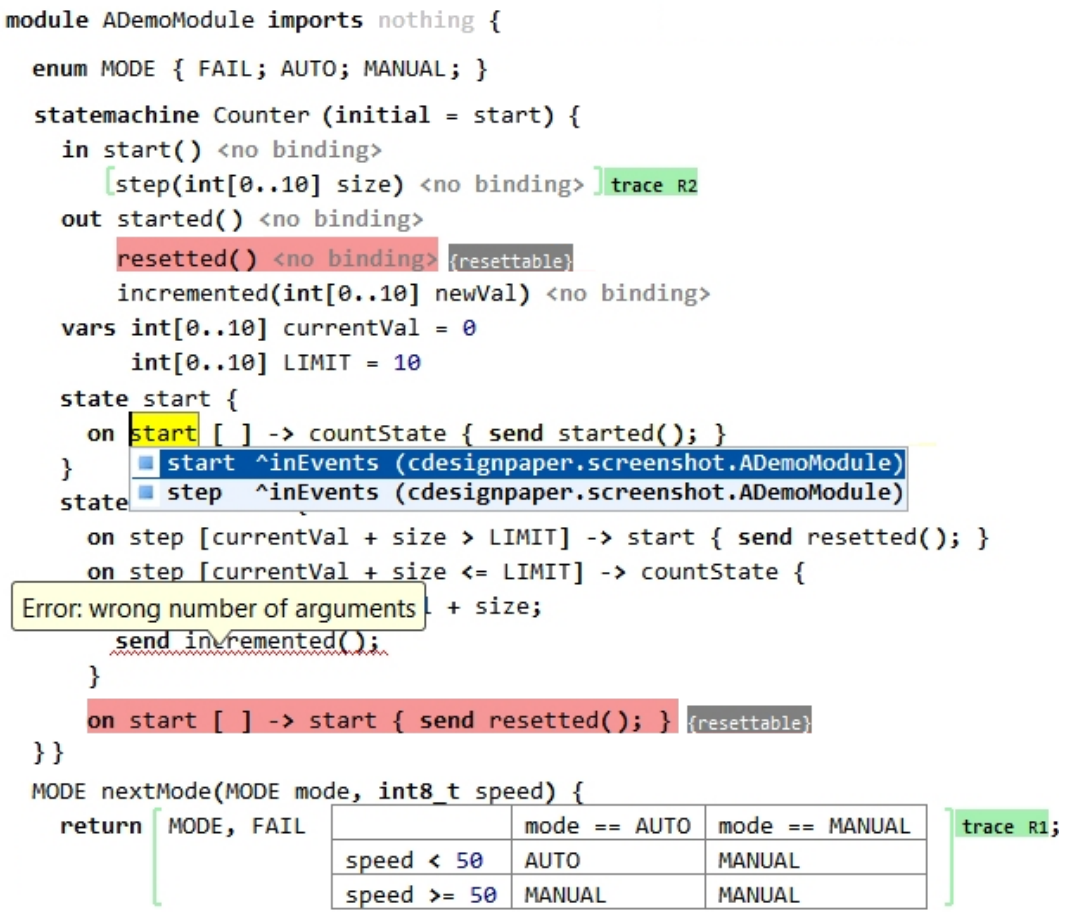


Figure 2.1: An mbeddr example program using five separate but integrated languages. It contains a module with an **enum**, a state machine (**Counter**) and a function (**nextMode**) that contains a decision table. Inside both of them developers can write regular C code. The IDE provides code completion for all language extensions (see the **start**/**stop** suggestions) as well as static error validation (**Error...** hover). The green **trace** annotations are traces to requirements that can be attached to arbitrary program elements. The red parts with the **{resettable}** next to them are presence conditions: the respective elements are only in a program variant if the configuration feature **resettable** is selected.

are considered by many practitioners as hard-to-use expert tools and are often avoided. This is because verifying *domain-level* properties (as opposed to low level properties of the code such as read-before-write errors) requires complex code annotations (e. g., Frama-C) or the use of tool-specific property specification languages. It is also difficult to re-interpret the analysis results on the domain level [14]. mbeddr makes formal analyses more accessible by relying on high-level, domain-specific language constructs (such as state machines or decision tables), making it easier to specify verification properties and interpret results.

Language extension is much more powerful than the alternatives available to C programmers today. In contrast to libraries, language extensions can lead to low runtime overhead because they are statically translated to C. They also provide extensions to the type system and the IDE. Macros can have similarly low overhead, but extensions of the type system and the IDE are not supported. Also, since macros are text transformations, all kinds of maintainability problems can result from excessive use of macros. Nevertheless, many macro libraries, such as Protothreads [8] (which implements lightweight threads), SynchronousC [18] and PRET-C[16] (both adding constructs for deterministic concurrency and preemption), are good candidates for abstractions that could be reified as language extensions based on mbeddr.

In private communication with the authors, a potential user from a major vendor of heating systems told us that he likes mbeddr because it is right in the middle between programming in C and high-level, rigid modeling tools, with the added benefit of extensibility. He argued that mbeddr lets him add language-level support for the abstractions relevant for his platform ("I can build my own AUTOSAR-like infrastructure for the heating systems domain with very little effort"). This is a nice summary of how mbeddr is to be used.

2.2 The mbeddr Stack

As Fig. 2.2 shows, mbeddr can be seen as a matrix. On the horizontal axis it is separated into an implementation concern (left) and an analysis concern (right). On the vertical axis it consists of a number of layers.

At the center is the MPS language workbench. On top of MPS, mbeddr ships with a number of core languages. On the implementation side the core language is C. On the analysis side, the core comprises languages that represent different analysis formalisms, currently SMT (satisfiability modulo theories) solving [17] and model checking [6]. The next layer up consists of default extensions. On the implementation side mbeddr ships

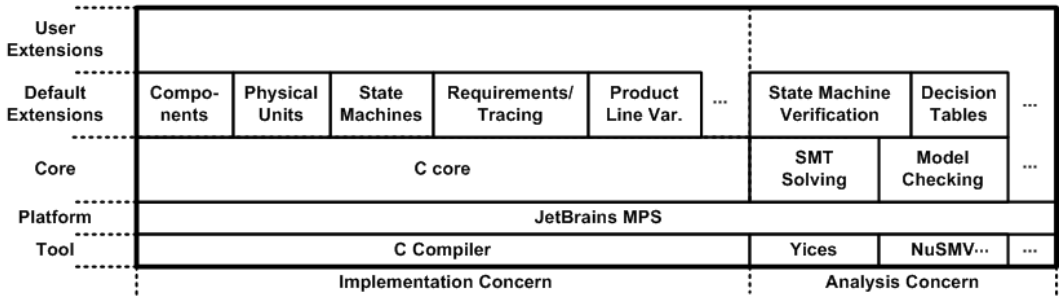


Figure 2.2: The mbeddr stack at a glance. Details are explained in the running text (Section 2.2).

C extensions for interfaces and components, physical units, state machines plus various smaller ones, such as decision tables (an example is at the bottom of Fig. 2.1). These build on top of C and also translate back to C during generation. On the analysis side the default extensions include support for model checking state machines and for checking the completeness and determinism of decision tables. Below the common platform JetBrains MPS, mbeddr integrates existing tools: a C compiler for the implementation side (`gcc` by default, but it can be exchanged), as well as the NuSMV[11] model checker and Yices[12] and CVC[13] SMT solvers. On top of the default extensions, users can develop their own application level DSLs. These typically rely on the core and default extensions either by directly extending (and translating back to) languages from those layers or by embedding subsets of the languages from these layers into new application level DSLs.

[11] `http://nusmv.fbk.eu`
[12] `http://yices.csl.sri.com`
[13] `http://www.cs.nyu.edu/acsys/cvc3`

Part II

Tutorials

3 Installation and Setup

However, this section describes the installation of mbeddr in detail, including Java and MPS itself.

> **Note:** We are working on a all-in-one distribution that includes MPS and mbeddr. Stay tuned.

3.1 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`

3.2 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.

3.3 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%
```

3.4 Graphviz

MPS supports visualization of models via PlantUML (**http://plantuml.sourceforge.net**), directly embedded in MPS. To use it, you have to install graphviz from **http://graphviz.org**. We use version 2.30. 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%
```

On Windows, you also have to have an environment variable **GRAPHVIZ_DOT** that points to the **dot.exe** file supplied with graphviz.

3.5 mbeddr

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

■ **Installing the Distribution** We recommend installing the distribution, because this is a much simpler process than working from the github sources. The distribution can be downloaded from the **mbeddr.com** website:

http://mbeddr.wordpress.com/getit/

The ZIP file you can get there contains this user guide, an early version of the extension guide as well as a set of plugins for MPS. Please take all the folders inside the **plugins** directory in the ZIP file and copy them into the **plugins** directory under MPS[1]. So, for example, after copying, there should be a **MPS_DIR/plugins/mbeddr.core** directory.

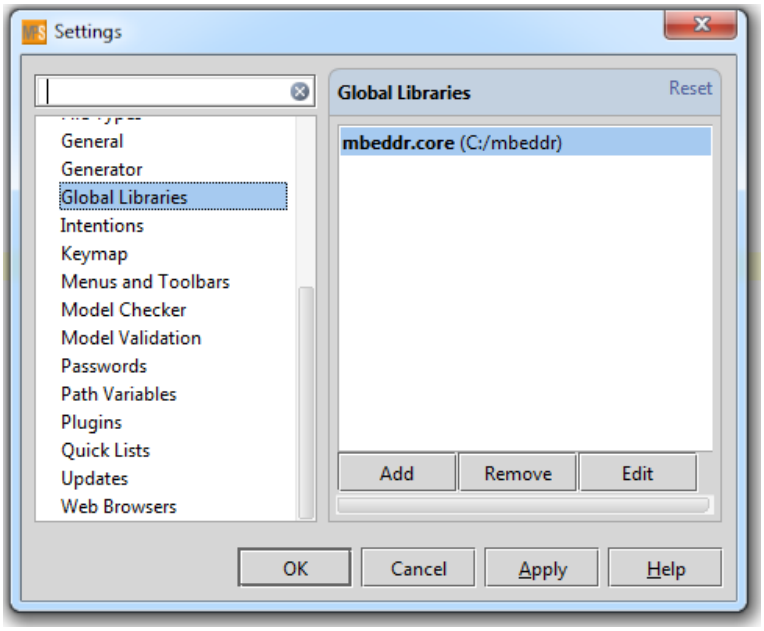
■ **Installing From github** The github repo ist a **https://github.com/mbeddr/mbeddr.core**. Here are the steps to get things running:

- Clone the repository onto your local machine.
- Inside the cloned repository, there is a **tools** folder. This folder contains a **mbeddr.spawner** folder. This folder (not just its contents) must be copied into the **plugins** directory inside MPS (note that there is also a **plugin** directory in MPS, without the **s**!).
- Rebuild all the languages. We provide a shell script for this purpose (**.bat** and **.sh**) called **rebuildLanguagesAfterCheckout**. It is in the **code/languages** directory. For this to work, you have to configure all **build.properties** files to point to your MPS install dir and the mbeddr root dir (**You have to use forward shlashes as directory separators even on Windows!**). The easiest way to find all the places where this has to be adapted is to search for all **build.properties.example**

[1]Note that there is also a **plugin** (without the **s**) directory under MPS!

under the mbeddr root mbeddr directory; remove the `.example` and fill in the respective directories.

- We now have to make the project aware of the **mbeddr.core** languages in the github checkout. Go to the *File* → *Settings* (on the Mac it is under *MPS* → *Preferences*) and select the *GlobalLibraries* in the IDE settings. Create a library called **mbeddr.core** that points to the **code** directory of the unzipped mbeddr installation. The library must point to the **code** 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.

- 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. The Eclipse code is packaged into a special MPS plugin which you have to install into MPS. To do so, please move (not just copy!) the **spawner** folder from the **tools** directory of mbeddr into the **plugins** directory of MPS (again, make sure it ends up the **plugins** directory, not in **plugin**).

You are now ready to build your first project with mbeddr.

3.6 Additional Verification Tools

To be able to use the verifications, you have to install a set of external tools:

- You have to install **NuSMV** from **http://nusmv.fbk.eu/** to be able to verify state machines.
- You have to install **Yices 1** from **http://yices.csl.sri.com/** to be able to verify decision tables and feature models.
- You have to install **CBMC** from **http://www.cprover.org/cbmc/** to be able to verify component protocols and contracts.

For each of these tools, the exectuables have to be on the path! You can verify that they are by simply typing **nusmv**, **yices** and/or **cbmc** somewhere in a console. The respective tools should start.

4 Writing Code in MPS

You can now start writing mbeddr code. Section 5.1 contains a tutorial that shows how to write the simplest possible program, the ubiquitous Hello, World. We suggest to take a look at this section.

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?

- Some language features may only be accessible via an intention. If you don't know how to go on, press **Alt-Enter** and see if the intentions window has something useful to offer.

■ **Entering Expressions** Expressions can be entered linearly. So if you have a number literal **42** and you want to change that expression to **42 + 2** you can simply move to the right side of the **42** and type **+** and then **2**. However, if you want to add something on the left side of an expression (e.g. changing **42** to **10 + 42**) then you have to move to the *left* side of the **42** and enter **+** as well; you can then enter **10**.

■ **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**. The reverse is also supported. The context menu on a program element supports Find Useages. This can also be activated via **Alt-F7**.

Within an mbeddr program, you can also press **Ctrl-F12** to get an outline view that lists all top level or important elements in that particular program so you can navigate to it easily.

■ **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.

■ **Select in Project** To select the currently edit root note in the project explorer (the big tree of the left), press **Alt-F1** and then **Enter**.

■ **Make and Rebuild** By default, **Ctrl-F9** (and **Cmd-F9** on the Mac) makes the current module, i.e. it regenerates and recompiles if the current module has changed. We recommend using the Keymap preferences to assign **Ctrl-Alt-F9** (and **Cmd-Alt-F9** on the Mac) to a complete Rebuild of the current module; sometimes changes aren't detected correctly and a full rebuild is necessary.

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.

5 Using mbeddr

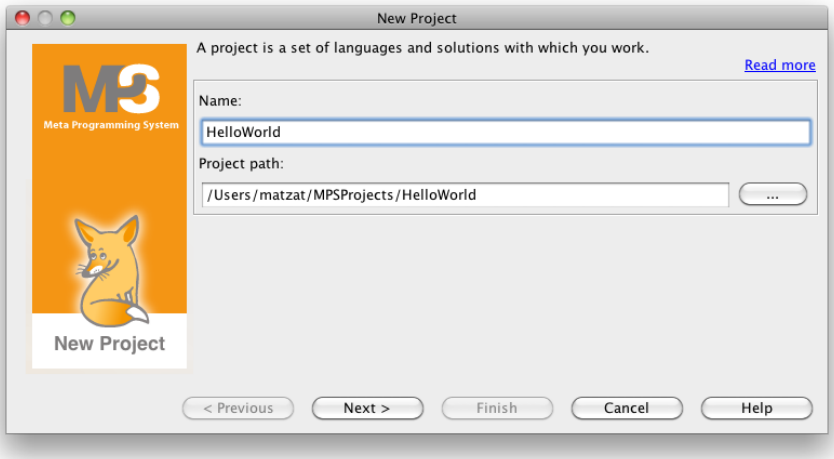
5.1 Hello World Example

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.

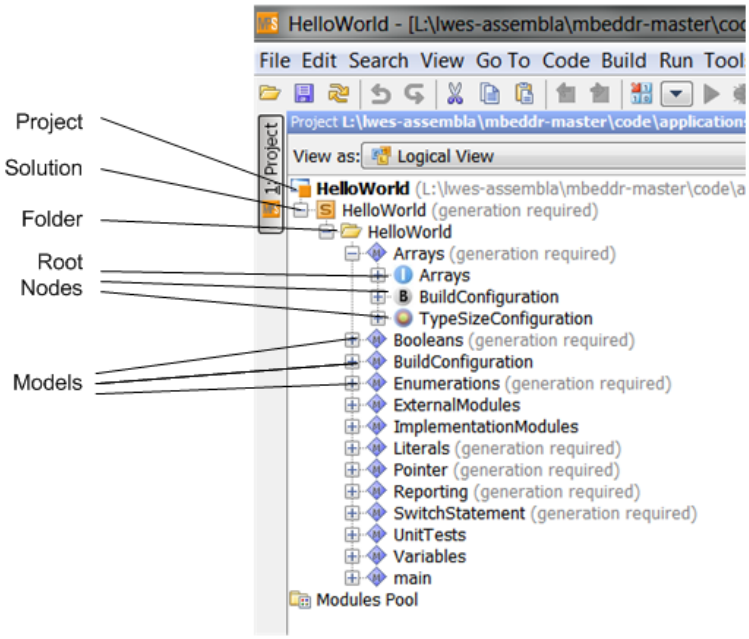
This tutorial showcases many of the features of mbeddr in an integrated example. The sources ZIP `com.mbeddr.tutorial.zip` is available from the download page at `mbeddr.com`. It is also part of the complete distro package. In the git repository the sources can be found in the `code/applications/tutorial` folder. Simply open the `.mpr` file to play with the example.

Notice that the tutorial does not discuss every aspect of every mbeddr extension — please refer to the respective chapters in the user guide.

■ **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.



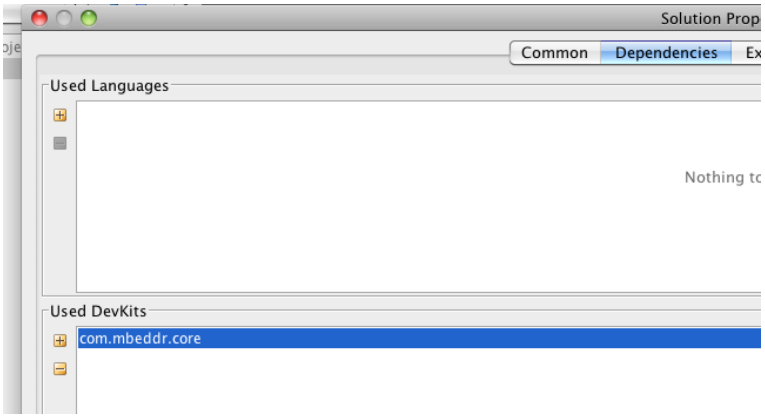
■ **Project Structure and Settings** An MPS project is a collection of solutions[1]. A *solution* is an application project that *uses* existing languages. Solutions contain any number of models; models contain root nodes. Physically, models are XML files that store MPS code. They are the relevant version control unit, and the fundamental unit of configuration.



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

In the solution, create a new model with the name **main**, prefixed with the solution's name: select *New* → *Model* from the solution's context menu. No sterotype.

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



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

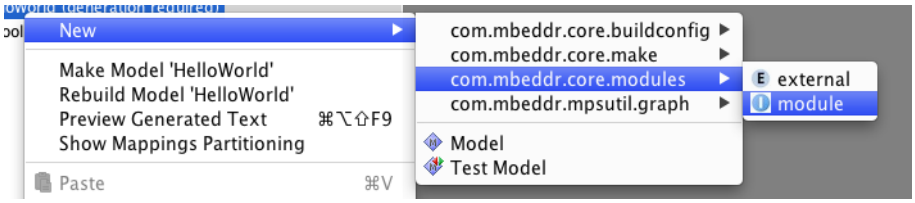
Note: In this tutorial we create the basic building blocks manually. However, there are also a couple of Wizards in the Code menu that create these things automatically.

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

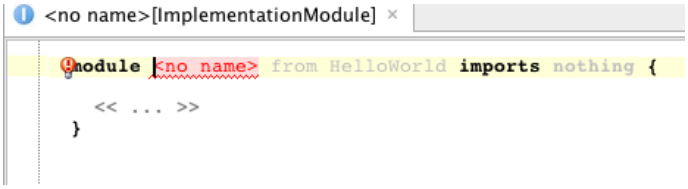
So to get started, we create a new **implementation module** using the model's context

menu as shown in the following screenshot (note that MPS may shorten package names; so instead of **com.mbeddr.core.modules** it may read **c.m.core.modules**):



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.

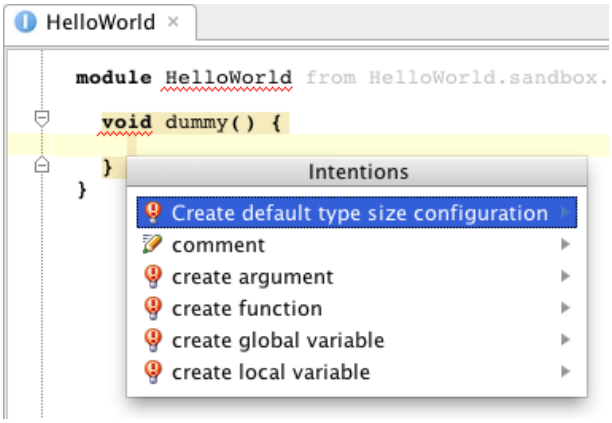
As a result, you will get an empty implementation module. It currently has no name (the name is red and underlined) and only a placeholder «...» where top level C constructs such as functions, **struct**s, or **enum**s can be added later.



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

```
module HelloWorld 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 7.1.5.



■ **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**) and then enter a name and the opening parentheses[2].
- 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 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

[2]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*.

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

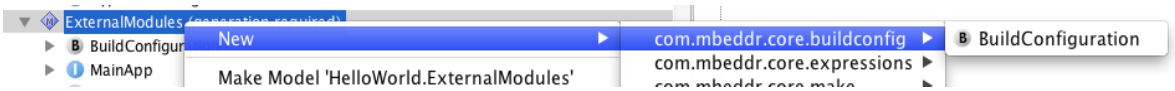
Within the message list, hit **Return** or type **message** to create a new message. Change the type from **ERROR** to **INFO** with the help of code completion. Specify the name **hello**. Add a message property by hitting **return** between the parentheses. The type should be a **string** and the name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

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

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

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

  int32 main(int8 argc, string[ ] args) {
    report(0) log.hello("World");
    return 0;
  }

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

■ **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[3]. 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: <none>
```

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
}
```

Note: Note that the whole process shown so far can be performed via a Wizard. Once you have a model that uses the **mbeddr.core** devkit, you can select select the **Make Hello World** wizard from the **Code** menu. It creates the same program. Also, the **Make Minimal System** creates a module with a main function, a type size configuration and a build configuration. **Make Minimal Test** creates a similar thing, but with a **test case** in it.

■ **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):

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

```
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).

■ **Visualizing the Program Structure** mbeddr comes with built-in UML-like diagramming capabilities. One kind of visualization is the module dependency structure of executables. To try it out, select the executable and execute the *Visualize* context menu option. You'll get a totally unimpressive diagram with just one node (because there's just one module in your executable). We'll get back to visualizations once they are more impressive.

In the visualization view, you can use the zoom in/zoom out buttons to change the size of the diagram, you can drag it around, you can refresh it, and, once you have more than one diagram, you can use the forward/back arrows to move between the 10 most recent diagrams. You can also save a diagram as an SVG file.

Some program elements may have more than one way how they can be visualized. If so, these alternatives can be selected from the combo box in the header of the visualization view.

Finally, most elements in the diagrams are clickable: the underlying program element is then selected in the editor.

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

5.2 Function Pointers

5.2.1 The Basic Program

As the first example, we will add a configurable event handler using function pointers. We create a new module **FunctionPointers** using the context menu **New -> c.m.core.module -> ImplementationModule** on the current model.

Inside it, we will add a **struct** called **DataItem** that contains two members. You create the **struct** by just typing **struct** inside the module. You add the members by simply starting to type the **int8** types.

```
struct DataItem {
  int8 id;
};
```

We then create two functions that are able to process the **DataItem**s. Here is one function that does nothing (intentionally). You enter this function by starting out with the **DataItem** type, then typing the name and then using the **(** to actually create the function (the thing has been a global variable up to this point!):

```
DataItem process_doNothing(DataItem e) {
  return e;
}
```

Other functions with the same signature may process the data in some specific way; We can generalize those into a function type using a **typedef**. Note that entering the function type **()=>()** is in fact a little bit cumbersome. The alias for entering it is **funtype**:

```
typedef (DataItem)=>(DataItem) as DataProcessorType;
```

We can now create a global variable that holds an instance of this type and that acts as a global event dispatcher. We also create a new, empty **test case** that we will use for making sure the program actually works. In the test we assign a reference to **process_doNothing** to that event handler.

```
DataProcessorType processor;
exported test case testProcessing {
  processor = :process_doNothing;
}
```

We can now write the first simple test:

```
exported test case testProcessing {
  processor = :process_doNothing;
  DataItem i1;
  i1.id = 42;
  DataItem i2 = processor(i1);
  assert(0) i2.id == 42;
}
```

Let us complete this into a runnable system. In the **Main** module we change our **main** function to run our new test. Note how we import the **FunctionPointers** module; we call the test case, which is visible because it is **exported**:

```
module Main imports FunctionPointers {
  exported int32 main(int32 argc, string*[] argv) {
    return test testProcessing;
  }
}
```

Looking at the build configuration we see an error that complains that the binary is inconsistent, because the **FunctionPointers** module is not included. We can fix this with a quick fix. This results in the following binary:

```
executable MbeddrTutorial isTest: true {
  used external libraries
  used mbeddr libraries
  included modules
    Main (mbeddr.tutorial.main.m1)
    FunctionPointers (mbeddr.tutorial.main.m1)
}
```

5.2.2 Building and Running

We can now build the system using **Ctrl-F9** or by selecting **Rebuild** from the context menu of the solution in the logical view on the left. If you have installed **gcc** correctly the binary should actually be compiled automatically. Here is the info message you should get in the MPS messages view:

```
make finished successfully for mbeddr.tutorial.main/mbeddr.tutorial.main.m1
```

Let us run this program from the command line. To get to the respective location in the file system, select the solution in the logical view, open the properties and copy the **Solution File** location. In my case it is:

```
/Users/markusvoelter/Documents/mbeddr/mbeddr.core/code/applications/tutorial/solutions/
    mbeddr.tutorial.main/mbeddr.tutorial.main.msd
```

We can open a console and **cd** to this directory, removing the last segment **mbeddr.tutorial.main.msd** from the path. In that directory we can **cd** to **source_gen**, and from there we can navigate down to the directory for the model: **cd mbeddr/tutorial/m1**

Using **ls** there, we can see the following:

```
$ ls -ll
total 104
-rw-r--r--  1 markusvoelter  staff   1189 Oct 30 21:11 FunctionPointers.c
-rw-r--r--  1 markusvoelter  staff    159 Oct 30 21:11 FunctionPointers.h
-rwxr-xr-x  1 markusvoelter  staff   9028 Oct 30 21:11 Main
-rw-r--r--  1 markusvoelter  staff    338 Oct 30 21:11 Main.c
-rw-r--r--  1 markusvoelter  staff    162 Oct 30 21:11 Main.h
-rw-r--r--  1 markusvoelter  staff    433 Oct 30 21:11 Makefile
drwxr-xr-x  4 markusvoelter  staff    136 Oct 30 21:11 bin
-rw-r--r--  1 markusvoelter  staff    943 Oct 30 21:11 module_dependencies.gv
-rw-r--r--  1 markusvoelter  staff  14069 Oct 30 21:11 trace.info
```

Importantly, we see a **Makefile**, so we can call **make**. This will build the binary:

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

This results in an executable **MbeddrTutorial**. We can run it by calling **./MbeddrTutorial** or by calling **make test**. The output should look as follows:

```
$ make test
./MbeddrTutorial
$$runningTest: running test () @FunctionPointers:test_testProcessing:0#767515563077315487
```

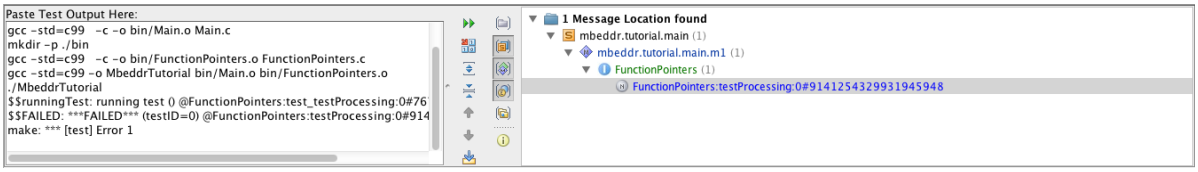
So the test succeeded, everything seems fine. Let us try to introduce an error by somehow breaking the assertion.

```
assert(0) i2.id == 0;
```

After regenerating the code (**Ctrl-F9** or **Rebuild**) we can call **make test** again and we get:

```
$ make test
mkdir -p ./bin
gcc -std=c99   -c -o bin/Main.o Main.c
mkdir -p ./bin
gcc -std=c99   -c -o bin/FunctionPointers.o FunctionPointers.c
gcc -std=c99 -o MbeddrTutorial bin/Main.o bin/FunctionPointers.o
./MbeddrTutorial
$$runningTest: running test () @FunctionPointers:test_testProcessing:0#767515563077315487
$$FAILED: ***FAILED*** (testID=0) @FunctionPointers:testProcessing:0#9141254329931945948
make: *** [test] Error 1
```

As you can see the test fails. It says that in the **FunctionPointers** module, **testProcessing** test case, assert number **0**. You can either navigate to the offending **assert** manually. Alternatively you can copy the console output into the clipboard and then paste it into the text box into the window that opens from the **Analyze -> mbeddr Analyze Error Output** menu:



Double clicking on the respective line opens the editor on the offending **assert** statement.

5.2.3 Lambdas

In contrast to regular C, mbeddr also provides lamdas, i.e. anonymous functions. They can be passed to functions that take function types as an argument. However, they can also be assigned to variables that have a function type, such as the **processor** above. Here is an example:

```
exported test case testLambdaProcessing {
  Trackpoint i1 = {
    id = 1
    timestamp = 0 s
    x = 0 m
    y = 0 m
    alt = 50 m
  };

  processor = [tp|
    tp.alt = 100 m;
    tp;
    ];
```

```
  assert(0) processor(i1).alt == 100 m;
}
```

A lamda is expressed as **[arg1, arg2, ...|statements]**. The type of the arguments is inferred from the context, they don't have to be specified. If several statements are required (as in the example above), they are layouted vertically. If only an expression is required, it is shown in line:

```
Trackpoint i1 = {
  ...
  alt = 100 m
};

processor = [tp|tp;];

assert(0) processor(i1).alt == 100 m;
```

⇒ This part of the tutorial only provided a few examples of Function Pointers. For a full discussion of Function Pointers see Section 7.1.8.

5.3 Physical Units

Let us go back to the definition of **DataItem** and make it more useful. Our application is supposed to work with tracking data (captured from a bike computer or a flight logger). We change the **struct** as follows (note that renaming the thing is trivial — you just change the name!):

```
struct Trackpoint {
  int8 id;        // sequence ID of the track point
  int8 timestamp; // timestamp as taken from GPS time
  int8 x;         // longtitude, simplified as a number
  int8 y;         // latitude, simplified as a number
  int8 alt;       // altitude as of the GPS
  int8 speed;     // current speed, if available
};
```

We can now enhance our test case the following way:

```
exported test case testProcessing {
  Trackpoint i1 = {
    id = 1
    x = 0
    y = 0
    alt = 100
  };
  processor = :process_doNothing;
  Trackpoint i2 = processor(i1);
  assert(0) i2.id == 1 && i2.alt == 100;
```

We can now use physical units to add more semantics to this data structure. We add the **com.mbeddr.physicalunits** devkit to the model properties. Then we can add units to the members. To add a unit, simply press **/** at the right side of one of the **int8** types:

```
struct Trackpoint {
  int8 id;
  int8/s/ timestamp;
  int8/m/ x;
  int8/m/ y;
  int8/m/ alt;
  int8 speed;
};
```

s and **m** are SI base units, so they are available by default. For the **speed** member we need to add **m/s**. Since this is not an SI base unit we first have to define it. To do so we create a new **UnitContainer** root in the current model (you can find the **UnitContainer** in the **c.m.ext.physicalunits** submenu of the **New** context menu on the current model). In it we can create a **derived unit**. Entering the **mps** and **speed** is trivial. Entering the meters per second is a bit cumbersome right now: press **m** for the meters, press **Enter**, press **s** for the seconds and then type **-1** directly on the right side of the **s** to enter the exponent:

```
Unit Configuration
                        -1
derived unit mps = m s    for speed
```

We can now go back to the **Trackpoint** and make the **speed** property use a unit: **int8/mps/ speed;**

Adding these units results in errors in the existing code because you cannot simply assign a plain number to a variable or member whose type includes a physical unit (**int8/m/ length = 3;** is illegal). Instead you have to add units to the literals as well. You can simply type the unit after the literal to get to the following:

```
Trackpoint i1 = {
  id = 1
  timestamp = 0 s
  x = 0 m
  y = 0 m
  alt = 100 m
};
...
assert(0) i2.id == 1 && i2.alt == 100 m;
...
assert(1) i3.id == 1 && i3.alt == 0 m;
```

If you try to rebuild now you will run into build errors:

```
no configuration item "physical units" found in this model. Please add a configuration item
     in your Build Configuration.
```

To fix this we have to go to the build configuration and add the respective configuration item:

```
Configuration Items
  reporting: printf (add labels false)
  physical units (config = Units Declarations (mbeddr.tutorial.main.m1))
```

If we rebuild now, everything should generate normally and we should be able to run the test again. Nothing should have changed so far. However, if we were to write the following code, we would get an error:

```
int8 someInt = i1.x + i1.speed; // error, adding apples and pears
```

The problem with this code is that you cannot add a length (**i1.x**) and a speed (**i1.speed**). And the result is certainly not a plain **int8**, so you cannot assign the result to **someInt**. Adding **i1.x** and **i1.y** will work, though. Also, you can calculate with units to write the following code. This gives you an additional level of type safety.

```
Trackpoint i4 = {
  id = 1
  timestamp = 10 s
  x = 100 m
  y = 0 m
  alt = 100 m
};
int8/mps/ speed = (i4.x - i2.x) / (i4.timestamp - i2.timestamp);
```

⇒ This part of the tutorial only provided a few examples of Physical Units. For a full discussion of Physical Units see Section 8.1.

5.4 Components

Let us now introduce components to further structure the system. We start by factoring the **Trackpoint** data structure into a parate module and export it to make it accessible from importing modules.

```
module DataStructures imports nothing {
  exported struct Trackpoint {
    int8 id;
    int8/s/ timestamp;
    int8/m/ x;
    int8/m/ y;
    int8/m/ alt;
    int8/mps/ speed;
  };
}
```

5.4.1 An Interface with Contracts

We now define an interface that handles **Trackpoint**s. To be able to do that we have to add the **com.mbeddr.components** devkit to the current model. We can then enter a client-server interface in a new module **Components**. We use pointers for the trackpoints here to optimize performance. Note that you can just press ∗ on the right side of **Trackpoint** to make it a **Trackpoint∗**:

```
module Components imports DataStructures {

  exported cs interface TrackpointProcessor {
    Trackpoint* process(Trackpoint* p);
  }

}
```

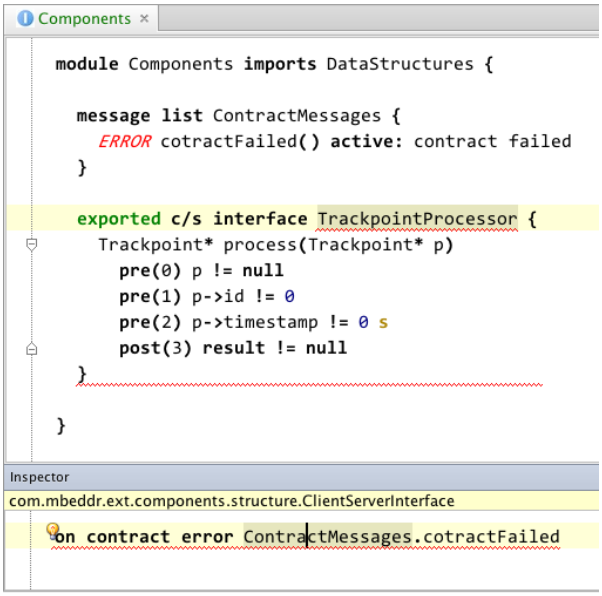
To enhance the semantic "richness" of the interface we can add preconditions. To do so, use an intention **Add Precondition** on the operation itself. Please add the following pre- and postconditions (note how you can of course use units in the precondition):

```
Trackpoint* process(Trackpoint* p)
  pre(0) p != null
  pre(1) p->id != 0
  pre(2) p->timestamp != 0 s
  post(3) result->id != 0
```

After you have added these contracts, you will get an error message on the interface. The problem is this: if a contract (pre- or postcondition) fails, the system will report a message (this message can be deactivated in case you don't want any reporting). However, for the program to work you have to specify a message on the interface. We create a new message list and a messge:

```
messagelist ContractMessages {
  ERROR contractFailed() active: contract failed
}
```

You can now open the inspector for the interface and reference this message from there:



There are still errors. The first one complains that the message list must be exported if the interface is exported. We fix it by exporting the message list (via an intention). The next error complains that the message needs to have to integer arguments to represent the operation and the pre- or postcondition. We change it thusly:

```
exported messagelist ContractMessages {
  ERROR contractFailed(int8 op, int8 pc) active: contract failed
}
```

5.4.2 A First Component

Let us create a new component by typing **component**. We call it **Nuller**. It has one provided port called **processor** that provides the **TrackpointProcessor** interface:

```
exported component Nuller extends nothing {
  provides TrackpointProcessor processor
}
```

We get an error that complains that the component needs to implement the operations defined by the **TrackpointProcessor** interface; we can get those automatically generated by using a quick fix on the provided port. This gets us the following:

```
exported component Nuller extends nothing {
  provides TrackpointProcessor processor
  Trackpoint* processor_process(Trackpoint* p) <- op processor.process {
    return null;
  }
```

```
}
```

The **processor_process** runnable is triggered by an incoming invocation of the **process** operation defined in the **TrackpointProcessor** interface. The **Nuller** simply sets the altitute to zero:

```
Trackpoint* processor_process(Trackpoint* p) <- op processor.process {
  p->alt = 0 m;
  return p;
}
```

Let us now write a simple test case to check this component. To do that, we first have to create an instance of **Nuller**. We create an instance configuration that has an instance of this component. Also, we add an adapter. An adapter makes a provided port of a component instance (**Nuller.processor**) available to a regular C program under the specified name **n**:

```
instances nullerInstances extends nothing {
  instance Nuller nuller
  adapt n -> nuller.processor
}
```

Now we can write a test case that accesses the **n** adapter — and through it, the **processor** port of the **Nuller** component instance **nuller**. We create a new **Trackpoint**, using 0 as the **id** — intended to trigger a contract violation (remember **pre(1) p->id != 0**). To enter the **&tp** just enter a **&**, followed by **tp**:

```
exported test case testNuller {
  Trackpoint tp = {
    id = 0
  };
  n.process(&tp);
}
```

Before we can run this, we have to make sure that the **instances** are initialized (cf. the warning you get on them). We do this right in the test case:

```
exported test case testNuller {
  initialize instances;
  Trackpoint tp = {
    id = 0
  };
  n.process(&tp);
}
```

To make the system work, you have to import the **Components** module into the **Main** module so you can call the **testNuller** test case from the **test** expression in **Main**. In the build configuration, you have to add the missing modules to the executable (using

the quick fix). Finally, also in the build configuration, you have to add the **components** configuration item:

```
Configuration Items:
  reporting: printf (add labels false)
  physical units (config = Units Declarations (mbeddr.tutorial.main.m1))
  components: no middleware
              wire statically: false
```

You can now rebuild and run. As a result, you'll get contract failures:

```
./MbeddrTutorial
$$runningTest: running test () @FunctionPointers:test_testProcessing:0#767515563077315487
$$runningTest: running test () @Components:test_testNuller:0#767515563077315487
$$contractFailed: contract failed (op=0, pc=1) @Components:null:-1#1731059994647588232
$$contractFailed: contract failed (op=0, pc=2) @Components:null:-1#1731059994647588253
```

We can fix these problems by changing the test data to conform to the contract:

```
Trackpoint tp = {
  id = 10
  timestamp = 10 s
  alt = 100 m
};
n.process(&tp);
assert(0) tp.alt == 0 m;
```

Let us provoke another contract violation by returning from the implementation in the **Nuller** component a **Trackpoint** whose **id** is 0:

```
Trackpoint* processor_process(Trackpoint* p) <- op processor.process {
  p->alt = 0 m;
  p->id = 0;
  return p;
}
```

Running it again provokes another contract failure. Notice how the contract is specified on the *interface*, but they are checked for each *component* implementing the interface. There is no way how an implementation can violate the interface contract without the respective error being reported!

5.4.3 Collaborating and Stateful Components

Let us look at interactions between components. We create a new interface, the **TrackpointStore**. It can store and return trackpoints[4]. Here is the basic interface:

[4]Sure, it is completely overdone to separate this out into a separate interface/component, but for the sake of the tutorial it makes sense.

```
exported cs interface TrackpointStore1 {
  void store(Trackpoint* tp)
  Trackpoint* get()
  Trackpoint* take()
  boolean isEmpty()
}
```

Let us again think about the semantics: you shouldn't be able to get or take stuff from the store if it is empty, you should not put stuff into it when it is full, etc. These things can be expressed as pre- and postconditions. The following should be pretty self-explaining. The only new thing is the **query** operation. Queries can be used from inside pre- and postconditions, but cannot modify state[5]

```
exported cs interface TrackpointStore1 {
  void store(Trackpoint* tp)
    pre(0) isEmpty()
    pre(1) tp != null
    post(2) !isEmpty()
  Trackpoint* get()
    pre(0) !isEmpty()
  Trackpoint* take()
    pre(0) !isEmpty()
    post(1) result != null
    post(2) isEmpty()
  query boolean isEmpty()
}
```

These pre- and postconditions mostly express a valid sequence of the operation calls: you have to call **store** before you can call **get**, etc. This can be expressed directly with protocols:

```
exported cs interface TrackpointStore2 {

  // store goes from the initial state to a new state full
  void store(Trackpoint* tp)
    protocol init(0) -> new full(1)

  // get expects the state to be full, and remains there
  Trackpoint* get()
    protocol full -> full

  // take expects to be full and then becomes empty (i.e. init)
  Trackpoint* take()
    post(0) result != null
    protocol full -> init(0)

  // and isEmpty has no effect on the protocol state
  query boolean isEmpty()
}
```

The two interfaces are essentially equivalent, and both are checked at runtime and lead to errors if the contract is violated.

[5]Currently this is not yet checked. But it will be.

We can now implement a component that provides this interface. Most of the following code should be easy to understand based on what we have discussed so far. There are two new things. There is a field **`Trackpoint* storedTP;`** that represents component state. Second there is an **`on-init`** runnable: this is essentially a constructor that is executed as an instance is created.

```
exported component InMemoryStorage extends nothing {
  provides TrackpointStore1 store
  Trackpoint* storedTP;

  void init() <- on init {
    storedTP = null;
  }

  void trackpointStore_store(Trackpoint* tp) <- op store.store {
    storedTP = tp;
  }
  Trackpoint* trackpointStore_get() <- op store.get {
    return storedTP;
  }
  Trackpoint* trackpointStore_take() <- op store.take {
    Trackpoint* temp = storedTP;
    storedTP = null;
    return temp;
  }
  boolean trackpointStore_isEmpty() <- op store.isEmpty {
    return storedTP == null;
  }
}
```

To keep our implementation module **`Components`** well structured we can use sectios. A **`section`** is a named part of the implementation module that has no semantic effect beyond that. Sections can be collapsed.

```
module Components imports DataStructures {

  exported messagelist ContractMessages {...}

  section processor {...}

  section store {
    exported cs interface TrackpointStore1 {
      ...
    }
    exported cs interface TrackpointStore2 {
      ...
    }
    exported component InMemoryStorage extends nothing {
      ...
    }
  }

  instances nullerInstances {...}
  test case testNuller {...}
  instances interpolatorInstances {...}
  exported test case testInterpolator { ... }
}
```

We can now implement a second processor. For subsequent calls of **process**, it computes the average of the two last speeds of the passed trackpoints. Let us start with the test case. Note how **p2** has its speed changed to the average of the **p1** and **p2** originally.

```
exported test case testInterpolator {
  initialize interpolatorInstances;
  Trackpoint p1 = {
    id = 1
    timestamp = 1 s
    speed = 10 mps
  };
  Trackpoint p2 = {
    id = 1
    timestamp = 1 s
    speed = 20 mps
  };

  ip.process(&p1);
  assert(0) p1.speed == 10 mps;

  ip.process(&p2);
  assert(1) p2.speed == 15 mps;
}
```

Let us look at the implementation of the **Interpolator**. Here it is.

```
exported component Interpolator extends nothing {
  provides TrackpointProcessor processor
  requires TrackpointStore1 store
  init int8 divident;
  Trackpoint* processor_process(Trackpoint* p) <- op processor.process {
    if (store.isEmpty()) {
      store.store(p);
      return p;
    } else {
      Trackpoint* old = store.take();
      p->speed = (p->speed + old->speed) / divident;
      store.store(p);
      return p;
    }
  }
}
```

A few things are worth mentioning. First, the component **requires** another one. More specifically it only expresses a requirement towards an interface, **TrackpointStore1** in our case. Any component that implements this interface can be used to fulfil this requirement (we'll discuss how below). Second, we use an **init** field. This is a regular field from the perspective of the component (i.e. it can be accessed from within the implementation), but it is special in that a value for it has to be supplied when the component is instantiated. Third, this example shows how to call operations on required interfaces (**store.store(p);**).

The only remaining step before running the test is to define the instances. Here is the code:

```
instances interpolatorInstances extends nothing {
  instance InMemoryStorage store
  instance Interpolator ip(divident = 2)
  connect ip.store to store.store
  adapt ip -> ip.processor
}
```

Two interesting things. First, notice how we pass in a value for the init field **divident** as we define an instance of **Interpolator**. Second, we use **connect** to connect the required port **store** of the **ip** instance to the **store** provided port of the **store** instance. If you don't do this you will get an error on the **ip** instance since it *requires* this thing to be connected (there are also **optional** required ports which may remain unconnected).

You can run the test case now. On my machine here it works successfully :-)

5.4.4 Mocks

Let us assume we wanted to test if the **Interpolator** works correctly with the **TrackpointStore** interface. Of course, since we have already described the interface contract semantically we would find out quickly if the **Interpolator** would behave badly. However, we can make such a test more explicit. Let us revisit the test from above:

```
exported test case testInterpolator {
  initialize interpolatorInstances;
  Trackpoint p1 = {...};
  Trackpoint p2 = {...};

  ip.process(&p1);
  assert(0) p1.speed == 10 mps;

  ip.process(&p2);
  assert(1) p2.speed == 15 mps;
}
```

In this test, we expect the following: when we call **process** first, the store is still empty, so the interpolator stores a new trackpoint. When we call **process** again, we expect the interpolator to call **take** and then **store**. In both cases we expect **isEmpty** to be called first. We can test for this behavior explicitly via a mock. A mock is a component that specifies the behavior it expects to see on a provided port during a specific test case.

The crucial point about mocks is that a mock implements each operation *invocation* separately (the **step**s below), whereas a regular component or even a stub just describe each operation with *one* implementation. This makes a mock implementation much simpler

— it doesn't have to replicate the algorithmic implementation of the real component. Let us look at the implementation:

```
mock component StorageMock report messages: true {
  provides TrackpointStore1 store
  Trackpoint* lastTP;
  total no. of calls is 5
  sequence {
    step 0: store.isEmpty return true;
    step 1: store.store {
        assert 0: parameter tp: tp != null
      }
      do { lastTP = tp; }
    step 2: store.isEmpty return false;
    step 3: store.take return lastTP;
    step 4: store.store
  }
}
```

This mock component expresses that we expect 5 calls in total. Then we describe the sequence of calls we expect. The first one must be a call to **isEmpty** and we return **true**. Then we expect a **store**, and for the sake of the example, we check that **tp** is not **null**. We also store the **tp** parameter in a field **lastTP** so we can return it later (you can add the parameter assertions and the **do** body with intentions). We then expect another **isEmpty** query, which we now answer with **false**. At this point we expect a call to **take**, and another call to **store**. Notice how we return **null** from **take**: this violates the postcondition! However, pre- and postconditions are *not* checked in mock components because their checking may interfere with the expectations! Also, we have slightly changed the test case so we don't stumble over the **null**. We don't **assert** anything about the result of the **process** calls:

```
ipMock.process(&p1);
ipMock.process(&p2);
```

Two more steps are required for this thing to work. The first one is the instances and the wiring. Notice how we now connect the interpolator with the mock:

```
instances interpolatorInstancesWithMock extends nothing {
  instance StorageMock storeMock
  instance Interpolator ip(divident = 2)
  connect ip.store to storeMock.store
  adapt ipMock -> ip.processor
}
```

The second thing is the test case itself. Obviously, we want the test case to fail if the mock saw something other than what it expects on its port. We can achieve this by using the **validate mock** statement in the test. Here is the complete test case (notice the **validate mock** in the last line):

```
exported test case testInterpolatorWithMock {
```

```
  initialize interpolatorInstancesWithMock;
  Trackpoint p1 = {
    id = 1
    timestamp = 1 s
    speed = 10 mps
  };
  Trackpoint p2 = {
    id = 1
    timestamp = 1 s
    speed = 20 mps
  };
  ipMock.process(&p1);
  ipMock.process(&p2);
  validatemock (0) interpolatorInstancesWithMock:storeMock;
}
```

5.4.5 Sender/Receiver Interfaces

So far we have always used client/server interfaces to communicate between components. These essentially define a set of operations, plus contracts, that can be invoked in a client/server style. However, mbeddr comes with a second kind of interface, the sender/receiver interface. In this case, the providing and requiring components exchange data items.

To demonstrate how they work, let us explore another aspect of the application around **Trackpoint**s (the example is in the **ComponentsSRI** implementation module). The data has to be collected in the airplane. Let us assume we have the following components: a GPS to provide the position, a speed indicator for the speed, and a flight recorder, whose job it is to create lists of **Trackpoint**s that capture the progress of the flight. All these components are time-triggered, i.e. it is assumed that they execute in regular intervals, by some kind of scheduler. They all provide an interface **Timed** that provides an operation **tick** that is called by the scheduler. So far, these components don't exchange any data yet: sender/receiver interfaces will be used for that later[6]. Here is the code so far:

```
module ComponentsSRI imports DataStructures {

  exported cs interface Timed {
    void tick()
  }

  exported component GPS extends nothing {
    provides Timed timed
    void update() <- op timed.tick {}
  }
```

[6]Note that this time-triggered architecture is very widespread in embedded software. In future releases of mbeddr we will provide direct support for time-triggered runnables, so you don't have to use an explicit interface such as **Timed**

```
  exported component SpeedIndicator extends nothing {
    provides Timed timed
    void update() <- op timed.tick {}
  }

  exported component FlightRecorder extends nothing {
    provides Timed timed
    void timed_tick() <- op timed.tick {}
  }

}
```

Let's now look at the data exchange, focussing on the position first. Here is a sender/receiver interface position provider. The interface declares a set of data elements, in this case with physical units:

```
exported sr interface PositionProvider {
  int8/m/ x;
  int8/m/ y;
  uint16/m/ alt;
}
```

The GPS is supposed to provide this data, so we give it a provided port with this interface:

```
exported component GPS extends nothing {
  provides PositionProvider pos
  provides Timed timed
  void init() <- on init {
    pos.x = 0 m;
    pos.y = 0 m;
    pos.alt = 0 m;
  }
  void update() <- op timed.tick {
    pos.x++;
    pos.y++;
  }
}
```

Note how from within component runnables we can use expressions to assign to the data values defined in the interface as if they were normal fields. Let us now look at the flight recorder. It is supposed to read the data written by the GPS (and the same with the speed indicator):

```
exported component FlightRecorder extends nothing {
  provides Timed timed
  requires PositionProvider pp
  requires SpeedProvider sp
  Trackpoint[1000] recordedFlight;
  uint16 count = 0;
  void timed_tick() <- op timed.tick {
    with (recordedFlight[count]) {
      id = ((int8) count)
      x = pp.x
```

```
      y = pp.y
      alt = pp.alt
      speed = sp.speed
    };
    count++;
  }
}
```

Inside the **with**-statement, we can access the data acquired via the **pp** and **sp** required ports.

What distinguishes this from global variables, of course, is that the component instances still have to be wired: required ports have to be connected to provided ports, in this case, defining access to the data items:

```
instances instances {
  instance GPS gps
  instance SpeedIndicator si
  instance FlightRecorder recorder
  connect recorder.sp to si.speed
  connect recorder.pp to gps.pos
}
```

⇒ This part of the tutorial only provided a few examples of Interfaces and Components. For a full discussion of Interfaces and Components see Section 8.2.

5.4.6 Visualing Components

mbeddr's diagramming capabilities are put to use in two ways in the context of components: component/interface dependencies and instance diagrams.

■ **Component/Interface Dependencies** Select a component or an interface and execute the *Visualize* action from the context menu (or press **Ctrl-Alt-V**). Fig. 5.1 shows the result.

■ **Instance/Wiring Diagrams** You can also select an instance configuration and visualize it. You'll get a diagram that shows component instances and their connections (Fig. 5.2).

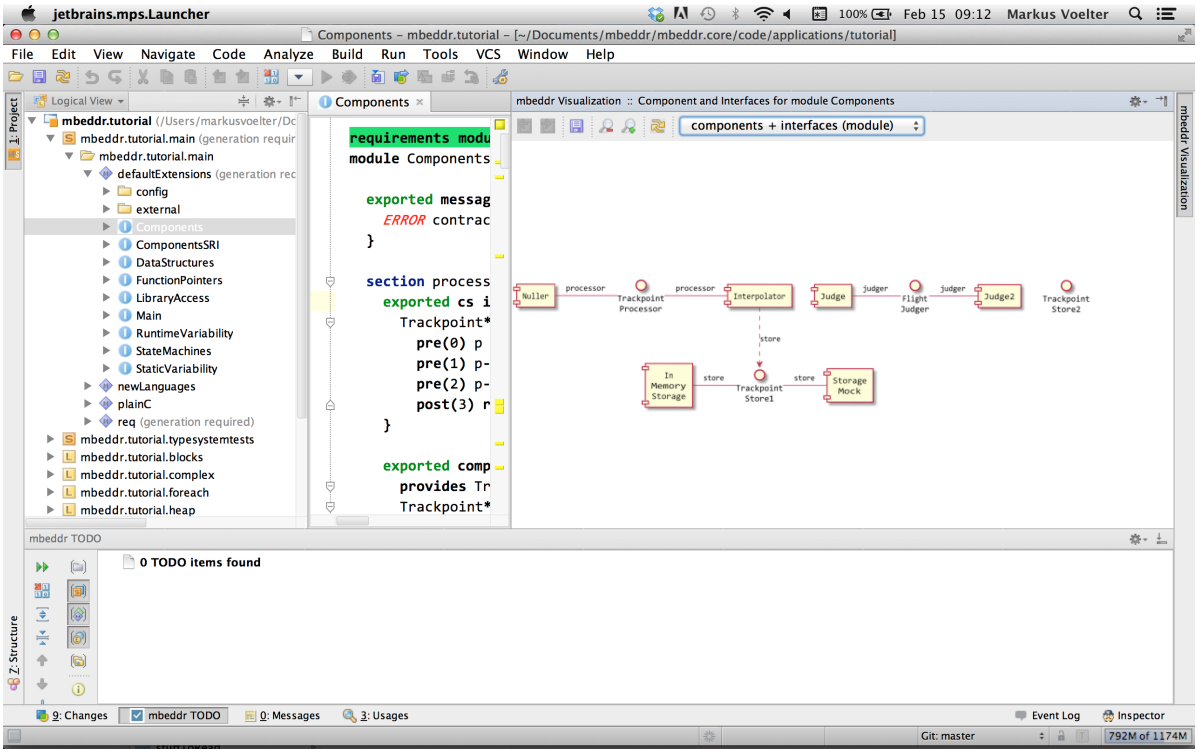


Figure 5.1: The interface/components dependency diagram shows all components visible from the current module, the interfaces, and the provided (solid lines) and required ports (dashed lines).

5.4.7 Contract Verification

mbeddr comes with support for verifying contracts of components statically. This verification is based on C-level dataflow analysis with CBMC. Let's set up our tutorial to use verification.

Let us verify the **InMemoryStorage** component. To do so, first add the **com.mbeddr.analyses.component** devkit to the model that contains the components code. Then use an intention to add the **verifiable** flag to the component. To make the verification work, you will have to provide some more information in the inspector:

```
configuration items
  entryPoint: verification
  loops unwinding: 2
  unwinding assertions: false
```

Let us look at the three parameters you have to set here: The first one determines from

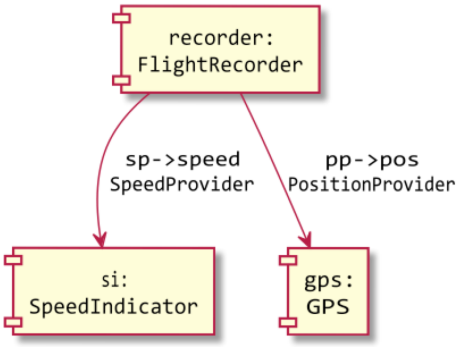


Figure 5.2: This diagram shows component instances and their connectors. The label in the instance boxes contain the instance name and the component name (after the colon). The edges represent connectors. The label shows the required port (before the arrow, the provided port name (after the arrow), and the name of the interface used by the two ports (on the new line).

where the program is "executed". The entry point should be selected to be "close" to the to-be-verified component (if you verify the whole system, then, at least for big systems, this will take long). In our case we use a special test case **verification**, which looks as follows:

```
instances verificationInstances {
  instance Interpolator interpol(divident = 2)
  connect interpol.store to store.store
  instance InMemoryStorage store
}

exported test case verification {
  initialize verificationInstances;
  Trackpoint p1 = {id = 1, timestamp = 1 s, speed = 10 mps };
  Trackpoint p2 = {id = 2, timestamp = 2 s, speed = 20 mps };
  $verificationInstances:interpol.processor_process(&p1);
  $verificationInstances:interpol.processor_process(&p2);
}
```

The second line in the configuration determines how often a loop is executed. You should start with low numbers to keep verification times low. Finally, the third parameter determines if the verification should fail in case it cannot be proven that the **unwinding loops** number is sufficient.

You can now run the verification by selecting the component and executing the **Verify Component** action. After a few seconds, you'll get a result table that reports everything as ok (see Fig. 5.3): every precondition of every operation in every provided port has been proven to be correct.

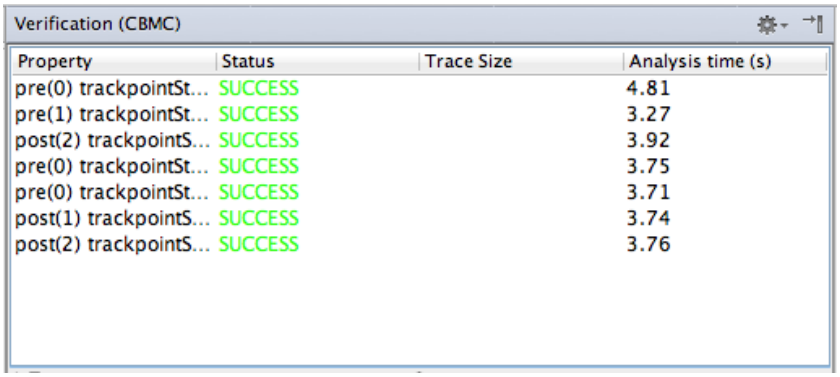


Figure 5.3: The table that shows the verification results; everything is ok in this case.

Let us introduce an error. The following version of the **trackpointStore_store** runnable does not actually store the trackpoint. This violates the postcondition, which claims that **storedTP != null**. Note that have to have the return: for the analysis to work, all paths through the body of a function (or a runnable) must end with a return (you'll get an in-IDE error if you don't do this).

```
void trackpointStore_store(Trackpoint* tp) <- op store.store {
  return;
}
```

Let us rerun the verification. Now we get an error, as shown in Fig. 5.4. Note how the lower part of the table now shows the execution trace that led to the contract violation. You should check the **Call/Return** checkbox to filter the trace to only show the call/return-granularity, and not every statement. You can also double-click onto the trace elements to select the particular program element in the code.

5.5 Decision Tables

Let us implement another interface, one that lets us judge flights (we do this in a new section in the **Components** module). The idea is that clients add trackpoints, and the **FlightJudger** computes some kind of score from it (consider some kind of biking/flying competition as a context):

```
exported cs interface FlightJudger {
  void reset()
  void addTrackpoint(Trackpoint* tp)
  int16 getResult()
}
```

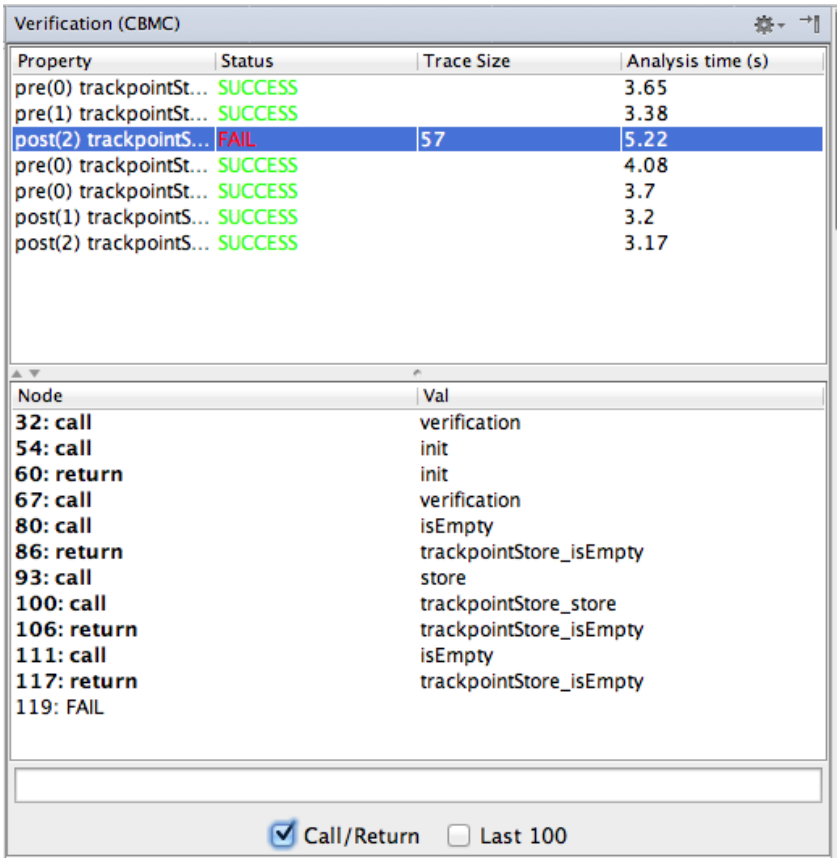


Figure 5.4: The table that shows the verification results; now we have an error, and the trace in the bottom half shows an example execution that led to the error.

Here is the basic implementation of a component that provides this interface.

```
exported component Judge extends nothing {
  provides FlightJudger judger
  int16 points = 0;
  void judger_reset() <- op judger.reset {
    points = 0;
  }
  void judger_addTrackpoint(Trackpoint* tp) <- op judger.addTrackpoint {
    points += 0; // to be changed
  }
  int16 judger_getResult() <- op judger.getResult {
    return points;
  }
}
```

Of course the implementation of **`addTrackpoint`** that just adds **`0`** to the **`points`** doesn't make much sense yet. The amount of points added should depend on how fast and how high the plane (or whatever) was going. The following screenshot shows an embed-

ded decision table that computes points (Notice we mix the components language, the decision tables and the units in one integrated program):

```
void judger_addTrackpoint(Trackpoint* tp) ← op judger.addTrackpoint {
  points += int16, 0
```

	tp->alt < 2000 m	tp->alt >= 2000 m
tp->speed < 150 mps	0	10
tp->speed >= 150 mps	5	20

```
} runnable judger_addTrackpoint
```

Let us write a test. Of course we first need an instance of **Judge**:

```
instances instancesJudging extends nothing {
  instance Judge j
  adapt j -> j.judger
}
```

Below is the test case. It contains two things you maybe haven't seen before. There is a second form of the **for** statement that iterates over a range of values. The range can be exclusive the ends or inclusive (to be changed via an intention). In the example we iterate from 0 to 4, since 5 is excluded. The **introduceunit** construct can be used to "sneak in" a unit into a regular value. This is useful for interacting with non-unit-aware (library) code. Note how the expression for **speed** is a way of expressing the same thing without the **introduceunit** in this case. Any expression can be surrounded by **introduceunit** via an intention.

```
exported test case testJudging {
  initialize instancesJudging;
  j.reset();
  Trackpoint[5] points;
  for (i in [0..5[) {
    points[i].id = i;
    points[i].alt = introduceunit[1850 + 100 * i -> m];
    points[i].speed = 130 mps + 10 mps * i;
    j.addTrackpoint(&points[i]);
  }
  assert(0) j.getResult() == 0 + 0 + 20 + 20 + 20;
}
```

5.5.1 Verifying the Decision Table

So far so good. The test case is fine. However, as with many tests, this one only tests part of the overall functionality. And in fact, you may have noticed that an error lurks inside our decision table: for 2000 m of altitude, the table is non-deterministic: both conditions in the column header work! We could find this problem with more tests, or we could use formal verification. Let's use the latter approach.

To do so, add the **com.mbeddr.analyses.dectab** devkit. Go back to the decision table and select the **Toggle Verifiable** intention. Marking the table this way enables the verification functionality and also reports errors in case the table uses expressions that cannot be verified. For example, if you were to use a non-linear expression **(tp->alt * tp->alt) > 2000 m2** as a condition, verification would not work. Our table does not have this problem, so we can now select **Verify Decision Table** from the table's context menu.

Below is the result we get. First, the checker reports that the table is complete, i.e. that all possible value combinations are taken care of. However, it also reports failires because the cells (1,1) and (2,1) are inconsistent. The problem is – as we expect – related to the altitude being 2000 m, because in this case it is not decidable which alternative should be used.

```
SUCCESS: Table complete.
FAIL: cells (1, 1) and (2, 1) are inconsistent.
  tp.x : 0
  tp.y : 0
  tp.speed : 0
  tp.timestamp : 0
  tp.id : 0
  tp.alt : 2000
FAIL: cells (1, 2) and (2, 2) are inconsistent.
  tp.x : 0
  tp.y : 0
  tp.speed : 150
  tp.timestamp : 0
  tp.id : 0
  tp.alt : 2000
```

We can fix the problem for example by changing the **tp->alt <= 2000 m** to **tp->alt < 2000 m**. Running the verification again results in this:

```
SUCCESS: Table complete.
OK: All cells are consistent
```

⇒ This part of the tutorial only provided a few examples of Verifying Decision Tables. For a full discussion of Verifying Decision Tables see Section 10.

5.6 Accessing Libraries

So far we have not accessed any functions external to the mbeddr program — everything was self-contained. Let us now look at how to access external code. We start with the simplest possible example. We would like to be able to write the following code:

```
module LibraryAccess imports nothing {
  exported test case testPrintf {
  printf("Hello, World\n");
    int8 i = 10;
    printf("i = %i\n", i);  }
  }
```

To make this feasible, we have to integrate C's standard **printf** function. We could import all of **stdio** automatically (we'll do that below). Alternatively, if you only need a few API functions from some library, it is simpler to just write the necessary proxies manually. Let's use the second approach first.

5.6.1 Manual Library Import

External functions are represented in a so-called **external module**. After you create one and give it a name, it looks like this:

```
// external module contents are exported by default
external module stdio_stub imports nothing resources nothing {
}
```

An external module is always associated with one or more "real" header files. The trick is that when an implementation module imports an external module in mbeddr, upon generation of the C code, the referenced real header is included into the C file. So the first thing we need to do is to express that this **stdio_stub** external module represents the **stdio.h** file:

```
// external module contents are exported by default
external module stdio_stub imports nothing resources header <stdio.h> {
}
```

In general, we also have to specify the **.o** or **.a** files that have to be linked to the binary during the **make** process (in the **resources** section of the external module). In case of **stdio.h**, we don't have to specify this — gcc somehow does this automatically.

So what remains to do is to write the actual **printf** prototype. This is a regular function signature. The ellipsis can be added via an intention. The same is true for the **const** modifier. This leads us to this:

```
// external module contents are exported by default
external module stdio_stub imports nothing resources header: <stdio.h> {
  void printf(const char* format, ...);
}
```

To be able write the test case, we have to import the **stdio_stub** into our **LibraryAccess** implementation module. And in the build configuration we have to add the **LibraryAccess** and the **stdio_stub** to the binary. We should also call the **testPrintf** test case from **Main**.

5.6.2 Automatic Header Import

Later we will need **malloc** and **free** so we can work with dynamic memory. We could create a **stdlib_stub** external module with these two functions manually, like we did it for **stdio** above. However, for the sake of example, we use the automatic import here.

First make sure the **com.mbeddr.core.cstub** language is configured for your model. Then create a new **HeaderImportSpecHFile** in your model. Once created, you can specify an import path. This directly should contain all header files that need to be imported. Note that you should *not* just point it to **use/include**, because importing all of these headers can take a long time! Instead copy the headers you need into a separate directly and specify that one in the import spec. This tutorial comes with a directly **headers** that contains only **stdlib.h**. You can now press the **(Re-)Import Headers** button.

Two things will happen. First, a dialog will open up that reports problems with the import. You can ignore the errors for now. Second, a new external module named **stdlib** is added to the model. Double-click it to open, and rename it to **stdlib_stub**. Once open, you will notice that there are a lof or errors in the file. This is for two reasons. The first one is that this header imports other headers that are not accessible — they were not in the **headers** directory. Consequently all kinds of symbols referred to from the **stdlib** headers are not defined. The second reason is that parsing and importing header files is generally a delicate operation, and for reasons that are beyond this tutorial, it is very likely that some things cannot be parsed. We discuss details . However, for our purposes the import is successful — **malloc** and **free** are correctly imported.

⇒ This part of the tutorial only provided a few examples of Textual C Code. For a full discussion of Textual C Code see Section 7.6.

5.7 State Machines

Next to components and units, state machines are one of the main C extensions available in mbeddr. They can be used directly in C programs, or alternatively, embedded into components. To keep the overall complexity of the example manageable, we will show state machine use directly in C.

5.7.1 Implementing a State Machine

To use state machines, the respective model has to use the **com.mbeddr.statemachines** devkit. We then create a new module **StateMachines** and add it to the build configuration. We can also create a test case and call it from **Main** (there is an intention on the **test** expression that automatically adds all test cases that are visible).

Our example state machine once again deals with judging a flight. Here are the rules:

- Once a flight lifts off, you get 100 points
- For each trackpoint where you go more than 100 mps, you get 10 points
- For each trackpoint where you go more than 200 mps, you get 20 points
- You should land as short as possible; for each trackpoint where you are on the ground, rolling, you get 1 point deducted.
- Once you land successfully, you get another 100 points.

This may not be the best example for a state machine, but it does show all of the state machine's features while staying with our example so far :-) So let's get started with a state machine.

```
statemachine analyzeFlight initial = <no initial> {
}
```

We know that the airplane will be in various states: on the ground, flying, landing (and still rolling), landed, and crashed. So let's add the states. Once you add the first of these states, the **initial** state will be set; of course it can be changed later:

```
statemachine analyzeFlight initial = beforeFlight {
  state beforeFlight {
  }
```

```
  state airborne {
  }
  state landing {
  }
  state landed {
  }
  state crashed {
  }
}
```

Our state machine is also a little bit untypical in that it only takes one event **next**, which represents the next trackpoint submitted for evaluation. Note how an event can have arguments of arbitrary C types, **Trackpoint** in the example. There is one more event that resets the analyzer:

```
statemachine analyzeFlight initial = beforeFlight {
  in next(Trackpoint* tp)
  in reset()
  state beforeFlight ...
}
```

We also need a variable **points** that keeps track of the points as they accumulate over the flight analysis:

```
statemachine analyzeFlight initial = beforeFlight {
  in next(Trackpoint* tp)
  in reset()
  var int16 points = 0
  state beforeFlight ...
}
```

We can now implement the rules outlined above using transitions and actions. Let us start with some simple ones. Whenever we enter **beforeFlight** we reset the points to 0. We can achieve this with an entry action in **beforeFlight**:

```
state beforeFlight {
  entry { points = 0; }
}
```

We also understand that all states other than **beforeFlight**, the **reset** event must transition back to the **beforeFlight**. Note that as a consequence of the entry action in the **beforeFlight** state, the points get resetted in all three cases:

```
state airborne {
  on reset [ ] -> beforeFlight
}
state landing {
  on reset [ ] -> beforeFlight
}
state landed {
  on reset [ ] -> beforeFlight
}
```

We can now implement the rules. As soon as we submit a trackpoint where the altitude is greater than zero we can transition to the airborne state. This means we have successfully taken off, and we should get 100 points in bonus. **TAKEOFF** is a global constant representing 100 (**#define TAKEOFF = 100;**):

```
state beforeFlight {
  entry { points = 0; }
  on next [tp->alt > 0 m] -> airborne
  exit { points += TAKEOFF; }
}
```

Let us look at what can happen while we are in the air. First of all, if when we are airborne we receive a trackpoint with zero altitude and zero speed (without going through an orderly landing process), we have crashed. If we are at altitute zero with a speed greater than zero, we are in the process of landing. The other two cases deal with flying at over 200 and over 100 mps. In this case we stay in the **airborne** state (by transitioning to itself) but we increase the points:

```
state airborne {
  on next [tp->alt == 0 m && tp->speed == 0 mps] -> crashed
  on next [tp->alt == 0 m && tp->speed > 0 mps] -> landing
  on next [tp->speed > 200 mps] -> airborne { points += VERY_HIGH_SPEED; }
  on next [tp->speed > 100 mps] -> airborne { points += HIGH_SPEED; }
  on reset [ ] -> beforeFlight
}
```

Note that the transitions are checked in the order of their appearance in the state machine; if several of them are ready to fire, the first one is picked. So be careful to put the "tighter cases" first. The landing process is essentially similar:

```
state landing {
  on next [tp->speed == 0 mps] -> landed
  on next [ ] -> landing { points--; }
  on reset [ ] -> beforeFlight
}
state landed {
  entry { points += LANDING; }
  on reset [ ] -> beforeFlight
}
```

5.7.2 Interacting with Other Code — Outbound

So how do we deal with the **crashed** state? Assume this flight analyzer is running on some kind of server, analyzing flights that are submitted via the web (a bit like **http://onlinecontest.org**). If we detect a crash, we want to notify a bunch of people of the event, so they can call the the BFU or whatever. In any case, assume we want to

call external code. You can do this in two ways. The first one is probably obvious. We simply create a function which we call from an entry or exit action:

```
statemachine FlightAnalyzer initial = beforeFlight {
  ...
  state crashed {
    entry { raiseAlarm(); }
  }
}
...
void raiseAlarm() {
  // send emails or whatever
}
```

Another alternative, which is more suitable for formal analysis (as we will see later) involves out events. From the entry action we **send** an out event, which we define earlier.

```
statemachine FlightAnalyzer initial = beforeFlight {
  out crashNotification()
  ...
  state crashed {
    entry { send crashNotification(); }
  }
}
```

Sending an event this way has no effect (yet), but we express from within the state machine that something that corresponds semantically to a crash notification will happen at this point (as we will see, this allows us to write model checkers that verify that a crash notification will go out). What remains to be done is to bind this event to application code. We can do this by adding a binding to the out event declaration:

```
out crashNotification() => raiseAlarm
```

The effect is the best of both worlds: in the generated code we do call the **raiseAlarm** function, but on the state machine level we have abstracted the implementation from the intent.

5.7.3 Interacting with Other Code — Inbound

Let us write some test code that interacts with a state machine. To do a meaningful test, we will have to create a whole lot of trackpoints. So to do this we create helper functions. These in turn need **malloc** and **free**, so we first create an additional external module that represents **stdlib.h**:

```
// external module contents are exported by default
external module stdlib_stub imports nothing resources header: <stdlib.h>{
```

```
  void* malloc(size_t size);
  void free(void* ptr);
}
```

We can now create a helper function that creates a new trackpoint based on an altitude and speed passed in as arguments:

```
Trackpoint* makeTP(uint16 alt, int16 speed) {
  static int8 trackpointCounter = 0;
  trackpointCounter++;
  Trackpoint* tp = ((Trackpoint*) malloc(sizeof Trackpoint));
  tp->id = trackpointCounter;
  tp->timestamp = introduceunit[trackpointCounter -> s];
  tp->alt = introduceunit[alt -> m];
  tp->speed = introduceunit[speed -> mps];
  return tp;
}
```

We can now start writing (and running!) the test. We first create an instance of the state machine (state machines act as types and can be instantiated). We then initialize the state machine by using the **sminit** statement:

```
exported test case testSM1 {
  FlightAnalyzer f;
  sminit(f);
}
```

Initially we should be in the **beforeFlight** state. We can chech this with an assertion:

```
assert(0) smIsInState(f, beforeFlight);
```

We also want to make sure that the **points** are zero initially. To be able to write a constraint that checks this we first have to make the **points** variable **readable** from the outside. An intention on the variable achieves this. We can then write:

```
assert(1) f.points == 0;
```

Let us now create the first trackpoint and pass it in. This one has speed, but no altitude, so we are in the take-off run. We assume that we remain in the **beforeFlight** state and that we still have 0 points:

```
smtrigger(f, next(makeTP(0, 20)));
assert(2) smIsInState(f, beforeFlight) && f.points == 0;
```

Now we lift off by setting the alt to 100 meters:

```
smtrigger(f, next(makeTP(100, 100)));
assert(3) smIsInState(f, airborne) && f.points == 100;
```

So as you can see it is easy to interact from regular C code with a state machine. For testing, we have special support that checks if the state machine transitions to the desired state if a specific event is triggered. Here is some example code (note that you can use the **test statemachine** construct only within test cases):

```
test statemachine f {
  next(makeTP(200, 100)) -> airborne
  next(makeTP(300, 150)) -> airborne
  next(makeTP(0, 90)) -> landing
  next(makeTP(0, 0)) -> landed
}
```

This concludes our discussion of state machines. If you are a C buff, you may have noticed that our application contains a memory leak. We allocate all kinds of heap data (in the **makeTP** function), but we never free it again. Read the chapter to do something about that.

⇒ This part of the tutorial only provided a few examples of State Machines. For a full discussion of State Machines see Section 8.3.

5.7.4 Verifying State Machines

Just like decision tables, state machines can also be verified. While in decision tables the verification is based on SMT solving, the state machine verification uses model checking. However, the fundamental approach is similar:

- Mark a state machine as **verifiable** using an intention
- Verifiable state machines are restricted in some ways, so after marking the state machine as **verifiable**, you may have to refactor it in a way so it is inside the verifiable subset. For example, it is not possible to assign to the same variable more than once during one transition (taking into account all exit and entry actions).
- You can then use the context menu and select **Verify State Machine** to run the verification
- After running the verification, the result is shown in a separate view.

Model checking requires that users specify a set of properties which the state machine must implement. The model checker checks for each property whether it holds *in all cases*. If it doesn't, the result of the model checker shows a counter example, i.e. one

possible execution of the state machine that leads to a situation where the property does not hold. In mbeddr, every state machine is checked for a specific set of properties *automatically*, so you as the user don't have to specify them. These automatic checks include the following:

- Is every state reachable, i.e. is there *some* sequence of events in the state machine that leads into each state. If not, the state is dead and can be removed.
- Is every transition potentially executable, i.e. is there *some* sequence of events that fires each transition. If not, the transition could be removed.
- Are all transitions deterministic, i.e. is it always clear which transition must fire. If not, more than one transition could fire at a given time, and the decision which one to fire is unclear[7].
- The state machines language supports bounded integers (i.e. integers that specify a value range). For each variable that uses a bounded integer type, the verifier checks that the variable actually stays within the specified bounds.

■ **Verifying the FlightAnalyzer Machine** We attach the **verifiable** flag to the **FlightAnalyzer** state machine and run the verifier. It has two problems. Firt, It claims that the **landing** state has nondeterministic transitions. The counter example is this:

```
State beforeFlight
in_event: next   next(0, -32768)
State beforeFlight
in_event: next   next(1, -32768)
State airborne
in_event: next   next(0, 101)
State landing
in_event: next   next(0, 0)
State landed
in_event: next   next(0, -32768)
```

Note: Contrary to intuition, the actually problematic step may not be the last one! In the example here, the **next(0, 0)** is the offending situation!

In this example, in the **landing** state, the **next(0, 0)** event is fired and results in nondeterminism. Let us look at the code:

```
state landing {
  on next [tp->speed == 0 mps] -> landed
  on next [ ] -> landing { points--; }
  on reset [ ] -> beforeFlight
```

[7] In the mbeddr implementation (based on a **switch** statement) the first of these transitions will fire. However, one should not rely on this fact and the verifier marks it as a problem.

```
}
```

The first transition fires if **speed == 0**, which is the case for **next(0, 0)**. The second one fires in any case. Of course what we wanted to express there is that this one should only fire *otherwise*. It the generated C code it happens to work correctly because of the ordering of the transitions. But in general, the situation is ambiguous: **next(0, 0)** potentially fires both. We can fix the problem by adding another guard condition (rerun the model checker to test this!):

```
state landing {
  on next [tp->speed == 0 mps] -> landed
  on next [tp->speed != 0 mps ] -> landing { points--; }
  on reset [ ] -> beforeFlight
}
```

The second problem is in the **airborne** state. Once again, the actual problem is in the second-to-last step: **next(0, 101)**. Here is the respective state:

```
state airborne {
  on next [tp->alt == 0 && tp->speed == 0] -> crashed
  on next [tp->alt == 0 && tp->speed > 0] -> landing
  on next [tp->speed > 200] -> airborne { points += VERY_HIGH_SPEED; }
  on next [tp->speed > 100] -> airborne { points += HIGH_SPEED; }
  on reset [ ] -> beforeFlight
}
```

The first problem we observe is that transition 4 is also true whenever transition 3 is true as well. So we should change transition four to

```
on next [tp->speed > 100 && tp->speed <= 200] ->
   airborne { points += HIGH_SPEED; }
```

However, rerunning the verification still FAILs this state. The real problem is an ambiguity between transition 2 and transition four, because we do not check for the altitude in transitions 3 and 4. Once again, we imply an ordering of the guard conditions ... the C implementation works fine. Here is the correct formulation of the state's transitions:

```
state airborne {
  on next [tp->alt == 0 && tp->speed == 0] -> crashed
  on next [tp->alt == 0 && tp->speed > 0] -> landing
  on next [tp->alt > 0 && tp->speed > 200] ->
        airborne { points += VERY_HIGH_SPEED; }
  on next [tp->alt > 0 && speed > 100 && speed <= 200] ->
        airborne { points += HIGH_SPEED; }
  on reset [ ] -> beforeFlight
}
```

5.7.5 Hierarchical State Machines

State machines can also be hierarchical. This means that a state may contain essentially a sub-state machine. The following piece of code shows an example. Here are the semantics:

- When a transition from outside a composite state targets a composite state, the initial state in that composite state is activated.
- A composite state can have transitions. These act as if they were defined for each of the states of the composite state.
- If a transition from an inner state A crosses a composite state-boundary (B), then the actions happen in the following order: exit actions of A, exit actions of B, transition action, and entry action of the transition's target.

```
statemachine HierarchicalFlightAnalyzer initial = beforeFlight {
  in next(Trackpoint* tp)
  in reset()
  out crashNotification() => raiseAlarm
  readable var int16 points = 0
  state beforeFlight {
    on next [tp->alt > 0 m] -> airborne
    exit { points += TAKEOFF; }
  }
  composite state airborne initial = flying {
    on reset [ ] -> beforeFlight { points = 0; }
    on next [tp->alt == 0 m && tp->speed == 0 mps] -> crashed
    state flying (airborne.flying) {
      on next [tp->alt == 0 m && tp->speed > 0 mps] -> landing
      on next [tp->speed > 200 mps] -> flying { points += VERY_HIGH_SPEED; }
      on next [tp->speed > 100 mps] -> flying { points += HIGH_SPEED; }
    }
    state landing (airborne.landing) {
      on next [tp->speed == 0 mps] -> landed
      on next [ ] -> landing { points--; }
    }
    state landed (airborne.landed) {
      entry { points += LANDING; }
    }
  }
  state crashed {
    entry { send crashNotification(); }
  }
}
```

5.7.6 State Machine Diagrams

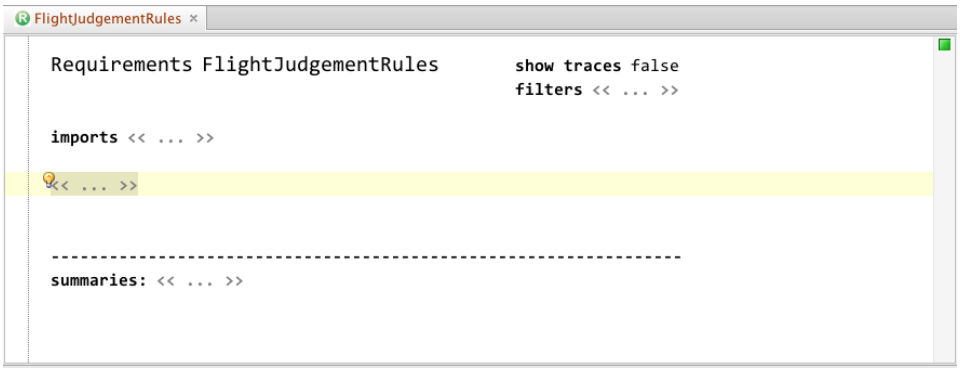
State Machines can also be rendered as a diagram; Fig. 5.5 shows an example.

5.8 Requirements

In the previous chapter we implemented a state machine that judged flights based on a set of rules. These rules were:

- Once a flight lifts off, you get 100 points
- For each trackpoint where you go more than 100 mps, you get 10 points
- For each trackpoint where you go more than 200 mps, you get 20 points
- You should land as short as possible; for each trackpoint where you are on the ground, rolling, you get 1 point deducted.
- Once you land successfully, you get another 100 points.

Let is capture these rules as requirements explictly in the tool. To do so, create a new model **`mbeddr.tutorial.main.req`** and add the **`com.mbeddr.cc.reqtrace`** devkit to it. Inside the model, create a new **`RequirementsModule`**. Call it **`FlightJudgementRules`**:

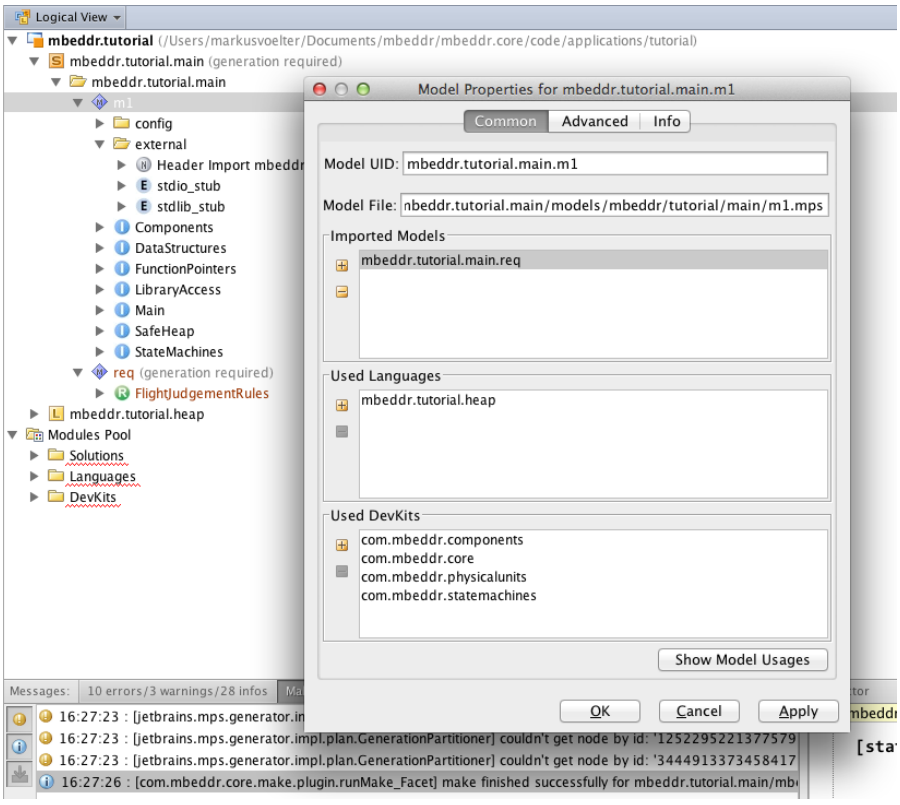


You can now enter the requirements from above. Make up a useful name for each of them and copy the text above into the summary or into the detail description text box.

To build hierarchies you can add child requirements with an intention. Here is how the result looks for me:

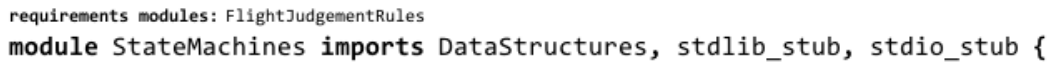
```
functional PointsForTakeoff (0): Once a flight lifts off, you get 100 points
functional InFlight (0): Points you get in flight
  functional FasterThan100 (0): For each trackpoint where you go more than 100 mps, y
  functional FasterThan200 (0): For each trackpoint where you go more than 100 mps, y
functional Landing (0): Stuff Relating to Landing
  functional ShortLandingRoll (0): You should land as short as possible
  functional FullStop (0): Once you land successfully, you get another 100 points.
```

Of course, now that we have captured the requirements we want to relate them to the implementation code, which is, in this case, a state machine. So let's go back to the state machine we built in the previous chapter. To be able to trace from the state machine to these requirements we first have to make the **req** model visible to the **m1** model that contains the state machine. Open **m1**'s properties, go to the **Imported Models** section and add the **req** model[8]:



[8]Note that this import is a *physical* import between the models (XML files essentially), whereas the **import** clause on modules is a *logical* namespace import.

While you are in the model properties dialog of **m1**, you can also add the **com.mbeddr.cc.reqtrace** devkit to that model; we need it for tracing. We want to establish traces between implementation code and the requirements we just captured. To make this possible we first have to specify that the **StateMachines** implementation module contains code that traces to the **FlightJudgementRules** requirements. We can do this by attaching a **Reference to a Requirements Module** to the implementation module by using an intention:



We can now add traces in the state machine. Remember the contents of the **beforeFlight** state:

```
state beforeFlight {
  entry { points = 0; }
  on next [tp->alt > 0 m] -> airborne
  exit { points += TAKEOFF; }
}
```

When the state is exited, we add 100 points (represented by the **TAKEOFF** constant). This is an implementation of the first requirement/rule. Select the exit action and execute the **Add Trace** action. Doing this on the other relevant program elements leads to a program with traces. The color of the trace actually depends on the kind of trace; the **implements** kind defaults to green:

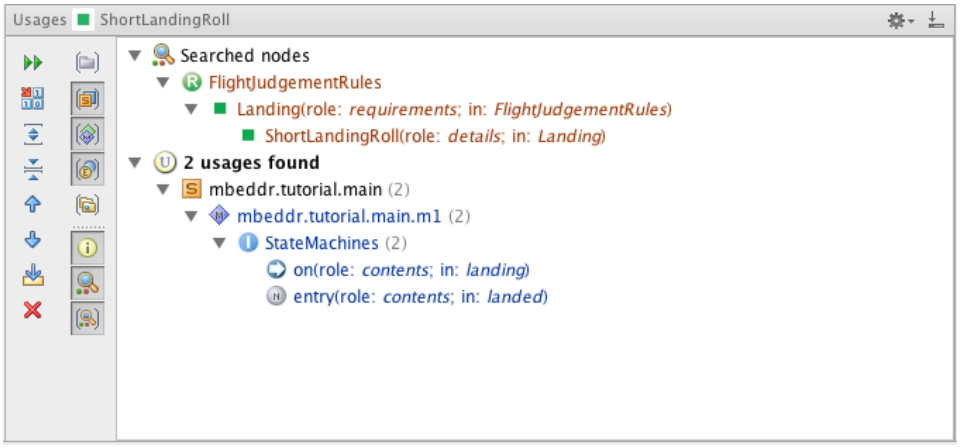
```
[#define TAKEOFF = 100;]-> implements PointsForTakeoff
[#define HIGH_SPEED = 10;]-> implements FasterThan100
[#define VERY_HIGH_SPEED = 20;]-> implements FasterThan200
[#define LANDING = 100;]-> implements Landing

statemachine FlightAnalyzer initial = beforeFlight {
  in next(Trackpoint* tp) <no binding>
  in reset() <no binding>
  out crashNotification() ⇒ raiseAlarm
  readable var int16 points = 0
  state beforeFlight {
    entry { points = 0; } entry
    on next [tp->alt > 0 m] ➔ airborne
    [exit { points += TAKEOFF; } exit]-> implements PointsForTakeoff
  } state beforeFlight
  state airborne {
    on next [tp->alt == 0 m && tp->speed == 0 mps] ➔ crashed
    on next [tp->alt == 0 m && tp->speed > 0 mps] ➔ landing
    [on next [tp->speed > 200 mps] ➔ airborne { points += 20; } on next]-> implements FasterThan200
    [on next [tp->speed > 100 mps] ➔ airborne { points += 10; } on next]-> implements FasterThan100
    on reset [ ] ➔ beforeFlight
  } state airborne
  state landing {
    on next [tp->speed == 0 mps] ➔ landed
    [on next [ ] ➔ landing { points--; } on next]-> implements ShortLandingRoll
    on reset [ ] ➔ beforeFlight
  } state landing
  state landed {
    [entry { points += LANDING; } entry]-> implements ShortLandingRoll
    on reset [ ] ➔ beforeFlight
  } state landed
  state crashed {
    entry { send crashNotification(); } entry
  } state crashed
}
```

Note how we attach traces to constants, transitions and actions: the trace facility is orthogonal with regards to the language constructs that are annotated. It works for all.

Returning back to the **FlightJudgementRules** requirements, we can select the heading and run the **Update Trace Status** intention. Requirements that are traced from program code are marked blue:

```
⊞ ● functional PointsForTakeoff (0): Once a flight lifts off, you get 100 points implemented
⊞ ● functional InFlight (0): Points you get in flight
  ⊞ ● functional FasterThan100 (0): For each trackpoint where you go more than 100 mps, you get 10 points implemented
  ⊞ ● functional FasterThan200 (0): For each trackpoint where you go more than 100 mps, you get 20 points implemented
⊞ ● functional Landing (0): Stuff Relating to Landing implemented
  ⊞ ● functional ShortLandingRoll (0): You should land as short as possible implemented
  ⊞ ● functional FullStop (0): Once you land successfully, you get another 100 points.
```

We notice that the **FullStop** requirement is still red, so it seems we haven't trace it correctly. Maybe we mixed something up with the landing stuff. Let's find out from where we trace to the **ShortLandingRoll** requirement. To do this, select the **Find Usages** entry from the context menu on the **ShortLandingRoll** requirement. In the dialog that opens uncheck all Finders except **Traces** and press **OK**. Here is the result you get:



The dialog reveals that we have attached **`ShortLandingRoll`** to two elements. A transitin and an entry action. That's wrong: the second one should have been the **`FullStop`** requirement. Change it, rerun **`Update Trace Status`** on the requirements module, and everything should be blue.

Finally, if you are annoyed by your beautiful state machine being polluted by all these requirements traces, you can hide them: select **`false`** for **`show traces`** in the top right corner of the **`FlightJudgementRules`** requirements module. The traces are still there, of course (and remain attached as you copy, paste or move program elements), but they are not shown.

⇒ This part of the tutorial only provided a few examples of Requirements and Tracing. For a full discussion of Requirements and Tracing see Section 9.1.

5.9 Product Line Variability

mbeddr supports two kinds of variability: runtime and static. *Runtime variability* makes the decision about which variant to run as the program executes. The binary contains the code for all options. Since the code must be aware of the variability, the underlying language must know about variability. *Static variability* makes the decision before program execution. In particular, the MPS generators remove all the program code that is not part of a particular variant. The binary is tailored, and the mechanism is generic — the target language does not have to be aware of the variability. We discuss both approaches in this chapter.

Both approaches have in common that in mbeddr, you first describe the available vari-

ability on an abstract level that is unrelated to the implementation artifacts that realize the variability. We use feature models for this aspect. A feature model describes the available variability (aka the configuration space) in a software system. Let us start by creating a feature model that describes variability for processing flights. To do so, add the **com.mbeddr.variability** devkit to your model and add a **VariabilitySupport** node into your program. After giving it a name, it looks as follows:

```
Variability Support FlightVariability
```

In this node you can now create a feature model. It has a name, and we specify the root feature **processing**.

```
feature model FlightProcessor {
  processing
```

We now add two sub-features **nullify** and **normalizeSpeed**. You can add children via an intention. By default, children are or-features (marked by the **?** in the parent), which means that each of the children can be in the system or not.

```
feature model FlightProcessor
  processing ? {
    nullify
    normalizeSpeed
  }
```

Let us add children to **normalizeSpeed**: **maxCustom** and **max100**. They are xor, so only one of them can be in a configuration. **maxCustom** has an attribute **maxSpeed**:

```
feature model FlightProcessor
  processing ? {
    nullify
    normalizeSpeed xor {
      maxCustom [int16/mps/ maxSpeed]
      max100
    }
  }
```

So here is what this all means: this feature model describes the variability of a flight processor (as usual...). If **nullify** is selected, the processor sets the altitude to 0 — we had seen this before. **normalizeSpeed** changes the speed. If the speed is over 100 (**max100**), it is set to 100. If **maxCustom** is selected and the speed is over **maxSpeed**, it is set to **maxSpeed**[9]. Notice how this is purely conceptual variability, and we haven't connected it to implementation code.

[9] I do realize that the example is getting more and more hairbrained, but what the heck ... it does illustrate mbeddr :-)

We can now go ahead and define configurations. These are instances of feature models, i.e. some of the features are selected, other aren't. A configuration has to be valid wrt. to its feature model: for example, a configuration cannot have **maxCustom** and **max100** selected at the same time, since those two features are mutually exclusive (**xor**). Here is the simplest possible configuration. It has no feature except the mandatory root feature:

```
configuration model cfgDoNothing configures FlightProcessor
  processing {
  }
```

Let us create another one by copying this one and then selecting features (press **Ctrl-Space** between **processing**'s curlies):

```
configuration model cfgNullifyOnly configures FlightProcessor
  processing {
    nullify
  }
```

A third configuration includes the **maxCustom** feature. You can add the the value of **maxSpeed** with an intention. Note that if you tried to add **max100** as well you'd get an error — the two are mutually exclusive.

```
configuration model cfgNullifyMaxAt200 configures FlightProcessor
  processing {
    nullify
    normalizeSpeed {
      maxCustom [maxSpeed = 200 mps]
    }
  }
```

5.9.1 Runtime Variability

As mentioned above, runtime variability means that we will evaluate a configuration *at runtime* and based on the evaluation, make decisions about program execution. To try this out, let us create a new module with the beginnings of a test case in it. Make sure you call the test case from **Main**.

```
module RuntimeVariability imports FunctionPointers {

  exported test case testRuntimeVar {
    Trackpoint tp = {
      id = 1
      timestamp = 0 s
      x = 0 m
      y = 0 m
      alt = 50 m
      speed = 220 mps
    };
```

```
  }
}
```

Now let's add variability support. It involves several steps. First we have to create a representation of the feature model in the program (we generate a **struct** that holds configurations). We do that by adding the following construct:

```
module RuntimeVariability imports FunctionPointers {

  feature model @ runtime for FlightProcessor;

  exported test case testRuntimeVar {...}
```

We can now create a function that processes trackpoints depending on the configuration. There are two noteworthy things. First, we pass values of type **fmconfig<FlightProcessor>** to the function, representing configurations for the **FlightProcessor** feature model (**cfgDoNothing**, **cfgNullifyOnly**, and **cfgNullifyMaxAt200** are valid values for this type). Second we use the **variant** statement to make parts of the procedural code depent on the set of selected features:

```
Trackpoint processTrackpoint(fmconfig<FlightProcessor> cfg, Trackpoint tp) {
  Trackpoint result;
  variant<cfg> {
    case (nullify && maxCustom) {
      result = process_nullifyAlt(tp);
      if (tp.speed > maxCustom.maxSpeed) {
        result.speed = maxCustom.maxSpeed;
      }
    }
    case (nullify && max100) {
      result = process_nullifyAlt(tp);
      if (tp.speed > 100 mps) {
        result.speed = 100 mps;
      }
    }
    case (nullify) { result = process_nullifyAlt(tp); }
    default { result = process_doNothing(tp); }
  }
  return result;
}
```

Note that the **case**s are evaluated top-to-bottom, so more specialized cases must be higher in the list. Also, only one **case** will ever be executed — no **break** needed! Notice how the **variant** construct is a new statement, so this does depend on C. Currently we support only this statement, but expressions or feature-dependent states (in state machines) would also be feasible, of course.

We can now go back to our test case and write assertions, calling the **processTrackpoint** function with several configuration models. Below is the first one. We first create a variable of type **fmconfig<FlightProcessor>** that holds a configuration (same type as in

the argument to **processTrackpoint**). We then use the **store config** construct to store a configuration (**cfgDoNothing**) into the **cfg** variable[10]. Then we call **processTrackpoint** with the configuration and the trackpoint and assert the result:

```
exported test case testRuntimeVar {
  Trackpoint tp = {...};

  fmconfig<FlightProcessor> cfg;

  store config<FlightProcessor, cfgDoNothing> into cfg;
  Trackpoint res1 = processTrackpoint(cfg, tp);
  assert(0) res1.alt == 50 m;
  assert(1) res1.speed == 220 mps;
}
```

We can store other configurations into **cfg** and reuse the variable:

```
store config<FlightProcessor, cfgNullifyOnly> into cfg;
Trackpoint res2 = processTrackpoint(cfg, tp);
assert(2) res2.alt == 0 m;
assert(3) res2.speed == 220 mps;

store config<FlightProcessor, cfgNullifyMaxAt200> into cfg;
Trackpoint res3 = processTrackpoint(cfg, tp);
assert(2) res3.alt == 0 m;
assert(3) res3.speed == 200 mps;
```

⇒ This part of the tutorial only provided a few examples of Product Line Variability. For a full discussion of Product Line Variability see Section 9.2.

5.9.2 Static Variability

Let us now look at static variability. As mentioned, it relies on the same variability specification (using feature models) as the dynamic variability. However, in case of static variability the variant is created during the transformation process in MPS. Let's build a simple example.

We first create a new implementation module with a test case in it (and we call the test from **Main**). We also add a simple function that nullifies the altitude of the trackpoint (as usual...):

```
module StaticVariability imports DataStructures {

  Trackpoint* process_trackpoint(Trackpoint* t) {
    t->alt = 0 m;
```

[10]This construct is a bit awkward, it would be nicer if this were an expression. However, because of limitations in how C deals with structs this is not possible without a performance hit. We didn't want to take that performance hit and have opted to go with the uglier syntax.

```
    return t;
  }

  exported test case testStaticVariability {
    Trackpoint tp = {
      id = 1
      alt = 2000 m
      speed = 150 mps
    };
    assert(1) process_trackpoint(&tp)->alt == 0 m;
  }
}
```

Let's now assume that we only want to do the nullification in the function if the **nullify** feature is selected in a particular configuration. To achive this we have to do two things. First we have to attach a dependency to a feature model to the implementation module. An intention achieves this. Second we have to annotate the **t->alt = 0 m;** statement with a presence condition. A presence condition is essentially a Boolean expression over feature in a feature model. If that expression is **false** for a given configuration, the annotated element will be removed. So let's add a presence condition to the **t->alt = 0 m;** with an intention, and make it depend on the **nullify** feature. We do the same on the assertion in the test case, and we add another assertion with the **!nullify** presence condition. Here is the resulting code:

```
Variability from FM: FlightProcessor
Rendering Mode: product line
module StaticVariability imports DataStructures {

  Trackpoint* process_trackpoint(Trackpoint* t) {
    {nullify}
    t->alt = 0 m;
    return t;
  } process_trackpoint (function)

  exported test case testStaticVariability {
    Trackpoint tp = {
      id = 1
      alt = 2000 m
      speed = 150 mps
    };

    {!nullify}
    assert(0) process_trackpoint(&tp)->alt == 2000 m;
    {nullify}
    assert(1) process_trackpoint(&tp)->alt == 0 m;
  } testStaticVariability(test case)

}
```

The color of the annotated code depends on the presence condition expression. All parts with the same expression have the same color. You can now play with the settings. By selecting **rendering mode: variant** and selecting a particular configuration model you can see and edit the program "cut down" to the respective configuration.

If you rebuild you model, nothing happens. While thre is product line variability expressed in the model, we don't yet consider it during generation. To do this we have to add a new configuration item in the build configuration.

```
variability mappings {
  fm FlightProcessor -> cfgNullifyOnly
}
```

This specifies that during build, the **cfgNullifyOnly** configuration should be used. By changing this mapping, you can build the system for several configurations (Section 9.2 discusses how you can build several variants at the same time).

Note that the effect in terms of succeeding test cases is the same in both examples :-) It is hence hard to see if it works. We can inspect the generated source code to see what's

going on. You can either open the file from the file system, or you can use the **Preview Generated Files** item in the context menu of the model.

⇒ This part of the tutorial only provided a few examples of Product Line Variability. For a full discussion of Product Line Variability see Section 9.2.

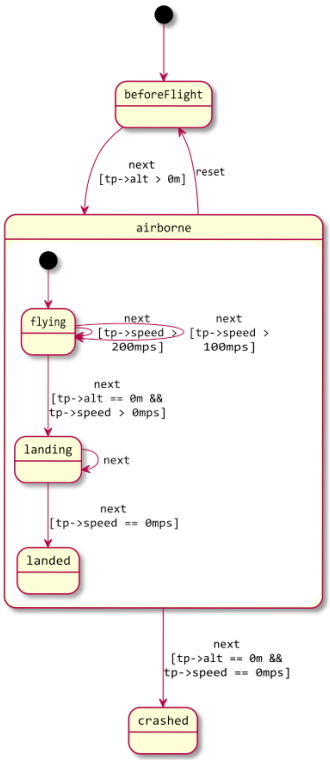


Figure 5.5: A visualization of a state machine in mbeddr. You can click on the states and transitions to select the respective element in the editor.

6 Extending mbeddr

This chapter discusses the *extension* of the mbeddr system by developing new languages or extending C. The chapter is essentially a collection of various example languages that each explain different aspects; there is also some overlap, i.e. some things may be explained more than once.

This chapter is not intended as a complete MPS tutorial, although, of course, by explaining how to build mbeddr extensions, we implicitly explain MPS. However, we do recommend to take a look at some of the tutorials on the MPS website. In particular, we recommend reading the LWC11 tutorial at

http://code.google.com/p/mps-lwc11/wiki/GettingStarted

We also recommend reading a paper that explains the various approaches for language composition:

http://voelter.de/data/pub/Voelter-GTTSE-MPS.pdf

The latter paper is also available as a set of narrated screencasts. Please watch them in this order:

http://www.youtube.com/watch?v=lNMRMZk8KBE
http://www.youtube.com/watch?v=ObWBUvHHNNg
http://www.youtube.com/watch?v=Y_KvApgKZZQ
http://www.youtube.com/watch?v=0vELEMU_Efc
http://www.youtube.com/watch?v=2_6DzD73qDs
http://www.youtube.com/watch?v=4vUkbzRHF2Q
http://www.youtube.com/watch?v=Z23-Vpy6RFY
http://www.youtube.com/watch?v=HbnttNqXpdM

This chapter consists of several sections. Each of them explain various aspects of mbeddr language extensions. While the different sections look as if they could be read separately

because they cover different examples, they do in fact build on each other: later sections do not repeat explanations for things explained in earlier sections. So we suggest you read through the whole chapter, beginning to end.

We also recommend you open the the actual example project and explore the code as you read the tutorials. Especially in the later sections we don't discuss every detail in the tutorial text.

■ **What if it doesn't work?** If you run into problems there are several proven ways of debugging language definitions. We discuss these things in Section 6.10. If it is not a debugging problem but just generally "strange behavior", you may want to try the following things:

- In case MPS doesn't see languages that should be visible or the editors for concepts isn't display, try closing and reopening the current project. Sometimes a restart of MPS can also help.
- If these tricks do not help, then you may want to try to use the **`File -> Invalidate Caches`** menu and restart MPS this way. MPS keeps a set of cache data structures, and sometimes these become corrupted. Invalidating the cache deletes the cache and rebuilds it completely.
- If you have problems with dependencies you may want to use **`Add Missing Imports`** on a model. In particular, this often helps in cases where MPS reports a *language X not found* even though it is actually mentioned as a Used Language for the respective model.

■ **Migrating Languages** If you define your own languages or extensions, it is likely that these change over time based on what you learn from using them. In this case you have to deal with existing programs: how do you keep them up-to-date with the changing language. In Section 6.11 we discuss some strategies to handle this challenge.

6.1 MPS Language Development Primer

As we have suggested in the intro to this part, we cannot provide a complete MPS tutorial and instead refer to the general MPS documentation. However, here are some basics that may help you get started.

6.1.1 The Structure of Programs and Languages

■ **Concrete and Abstract Syntax** Programs can be represented in their abstract syntax and the concrete syntax forms. The *concrete syntax* is the notation with which the user interacts as he edits a program. It may be textual, symbolic, tabular, graphical or any combination thereof. The *abstract syntax* is a data structure that represents the semantically relevant data expressed by a program (also known as a meta model or the *structure* of a language). It does not contain notational details such as keywords, symbols, white space or positions, sizes and coloring in graphical notations. The abstract syntax is used for analysis and downstream processing of programs. A language definition includes the concrete and the abstract syntax, as well as rules for mapping one to the other. *Parser-based* systems map the concrete syntax to the abstract syntax. Users interact with a stream of characters, and a parser derives the abstract syntax by using a grammar and mapping rules. *Projectional* editors like MPS go the other way round: user interactions, although performed through the concrete syntax, *directly* change the abstract syntax. The concrete syntax is a mere projection (that looks and feels like text when a textual projection is used). No parsing takes place.

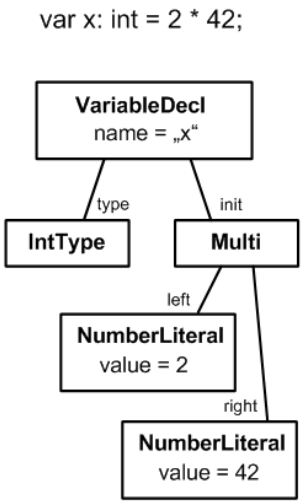
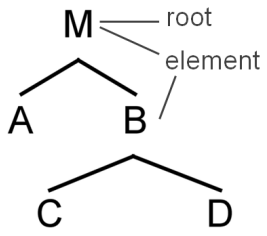


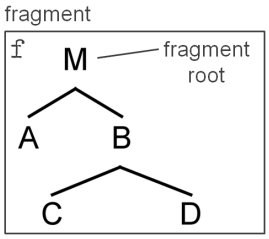
Figure 6.1: Concrete and abstract syntax for a textual variable declaration. Notice how the abstract syntax does not contain the keyword `var` or the symbols `:` and `;`.

The abstract syntax of programs are primarily trees of program *elements* or *nodes* in MPS' terminology. Each element is an instance of a *language concept*, or *concept* for short. A language is essentially a set of concepts (we'll come back to this below). Every element (except the root) is contained by exactly one parent element.

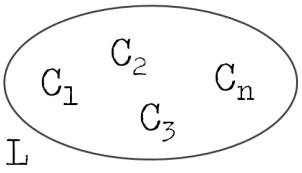


Syntactic nesting of the concrete syntax corresponds to a parent-child relationship in the abstract syntax. There may also be any number of non-containing cross-references between elements, established during editing in a projectional system.

■ **Fragments** A program may be composed from several program *fragments*. A fragment is a standalone tree, a partial program. In MPS fragments are known as *Root Nodes*.

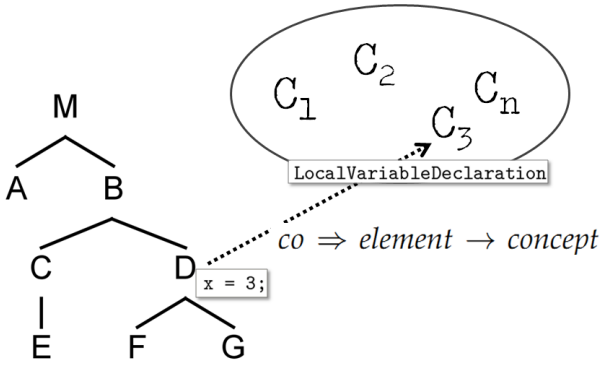


■ **Languages** A language l consists a set of language concepts C_l and their relationships[1]. We use the term *concept* to refer to all aspects of an element of a language, including concrete syntax, abstract syntax, the associated type system rules and constraints as well as some definition of its semantics.



In a fragment, each element e is an instance of a concept c defined in some language l. In the example below, the statement `int x = 3;` is an instance of the `LocalVariableDeclaration` concept. `int` is an instance of `IntType`, and the `3` is an instance of `NumberLiteral`.

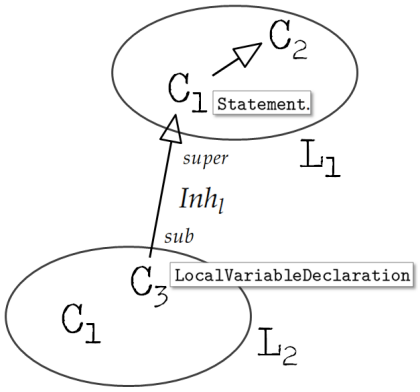
[1] In the world of grammars, a concept is essentially a *Nonterminal*. We will discuss the details about grammars in the implementation section of this book



We define an inheritance relationship that applies the Liskov Substitution Principle (LSP) to language concepts. The LSP states that,

In a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may be substitutes for objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.)

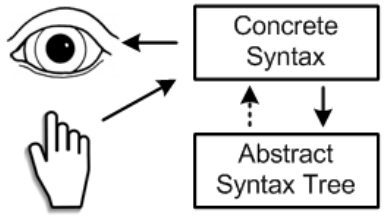
The LSP is well known in the context of object-oriented programming. In the context of language design it implies that a concept c_{sub} that extends another concept c_{super} can be used in places where an instance of c_{super} is expected.



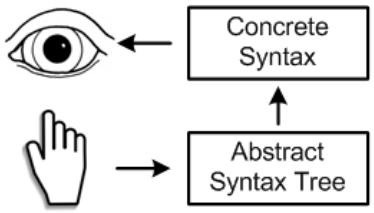
The **`LocalVariableDeclaration`** introduced above extends the concept **`Statement`**. This way, a local variable declaration can be used wherever a **`Statement`** is expected, for example, in the body of a function, which is a **`StatementList`**.

6.1.2 Projectional Editing

In the parser-based approach, a grammar specifies the sequence of tokens and words that make up a structurally valid program. A parser is generated from this grammar. A parser is a program that recognizes valid programs in their textual form and creates an abstract syntax tree or graph. Analysis tools or generators work with this abstract syntax tree. Users enter programs using the concrete syntax (i.e. character sequences) and programs are also stored in this way.



Projectional editors (also known as structured editors) work without grammars and parsers. A language is specified by defining the abstract syntax tree, then defining projection rules that render the concrete syntax of the language concepts defined by the abstract syntax. Editing actions *directly* modify the abstract syntax tree. Projection rules then render a textual (or other) representation of the program. Users read and write programs through this projected notation. Programs are stored as abstract syntax trees, usually as XML. As in parser-based systems, backend tools operate on the abstract syntax tree.



Projectional editing is well known from graphical editors; virtually all of them use this approach. However, projectional editors can also be used for textual syntax. While in the past projectional text editors have acquired a bad reputation mostly because of bad usability, as of 2011, the tools have become good enough, and computers have become fast enough to make this approach feasible, productive and convenient to use. MPS is a good example of such a usable tool.

This explicit instantiation of AST objects happens by picking the respective concept from the code completion menu using a character sequence defined by the respective concept. If at any given program location two concepts can be instantiated using *the same character sequence*, then the projection editor prompts the user to decide[2]. Once a concept is instantiated, it is stored as a node with a unique ID (UID) in the AST. References between program elements are based on actual pointers (references to UIDs), and the projected syntax that represents the reference can be arbitrary. The AST is actually an abstract syntax graph *from the start* because cross-references are first-class rather than being resolved after parsing[3] The program is stored using a generic tree persistence mechanism, often XML.

Defining a projectional editor, instead of defining a grammar, involves the definition of projection rules that map language concepts to a notation. It also involves the definition of event handlers that modify the AST based on users' editing gestures. The way to define the projection rules and the event handlers is specific to the particular tool used.

The projectional approach can deal with arbitrary syntactic forms including traditional text, symbols (as in mathematics), tables or graphics. Since no grammar is used, grammar classes are not relevant here. In principle, projectional editing is simpler in principle than parsing, since there is no need to "extract" the program structure from a flat textual source. However, as we will see below, the challenge in projectional editing lies making the editing experience convenient. In particular, editing notations that look like text should be editable with the editing gestures known from text editors. Here are some of the approaches MPS uses to achieve this. For you as the language developer this is relevant because you have to explicitly define some of these:

- Every language concept that is legal at a given program location is available in the code completion menu. In naive implementations, users have to select the language concept (based on its name) and instantiate it. This is inconvenient. In MPS, languages can instead define aliases for language concepts, allowing users to "just type" the alias, after which the concept is immediately instantiated[4].

- So-called side transforms make sure that expressions can be entered conveniently. Consider a local variable declaration `int a = 2;`. If this should be changed to `int a = 2+3;` the `2` in the init expression needs to be replaced by an instance of the binary `+` operator, with the `2` in the left slot and the `3` in the right. Instead of removing the `2` and manually inserting a `+`, users can simply type `+` on the right side

[2] As discussed above, this is the situation where many grammar-based systems run into problems from ambiguity.

[3] There is still one single containment hierarchy, so it is really a tree with cross-references.

[4] By making the alias the same as the leading keyword (e.g., `if` for an `IfStatement`), users can "just type" the code.

of the **2**; the system performs the tree restructuring that moves the **+** to the root of the subtree, puts the **2** in the left slot, and then puts the cursor into the right slot, to accept the second argument. This means that expressions (or anything else) can be entered linearly, as expected. For this to work, operator precedence has to be specified, and the tree has to be constructed taking these precedences into account. Precedence is typically specified by a number associated with each operator, and whenever using a side transformation to build an expression, the tree is automatically reshuffled to make sure that those operators with a higher precedence number are further down in the tree.

- Delete actions are used to similar effect when elements are deleted. Deleting the **3** in **2+3** first keeps the plus, with an empty right slot. Deleting the **+** then removes the **+** and puts the **2** at the root of the subtree.

- Wrappers support instantiation of concepts that are actually children of the concepts allowed at a given location. Consider again a local variable declaration **int a;**. The respective concept could be **LocalVariableDeclaration**, a subconcept of **Statement**, to make it legal in method bodies (for example). However, users simply want to start typing **int**, i.e. selecting the content of the **type** field of the **LocalVariableDeclaration**. A wrapper can be used to support entering **Type**s where **LocalVariableDeclaration**s are expected. Once a **Type** is selected, the wrapper implementation creates a **LocalVariableDeclaration**, puts the **Type** into its **type** field, and moves the cursor into the **name** slot.

- Smart references achieve a similar effect for references (as opposed to children). Consider pressing **Ctrl-Space** after the **+** in **2+3**. Assume further, that a couple of local variables are in scope and that these can be used instead of the **3**. These should be available in the code completion menu. However, technically, a **VariableReference** has to be instantiated, whose **variable** slot then is made to point to any of the variables in scope. This is tedious. Smart references trigger special editor behavior: if in a given context a **VariableReference** is allowed, the editor *first* evaluates its scope to find the possible targets and then puts those targets into the code completion menu. If a user selects one, *then* the **VariableReference** is created, and the selected element is put into its **variable** slot. This makes the reference object effectively invisible in the editor.

- Smart delimiters are used to simplify inputting list-like data that is separated with a specific separator symbol, such as parameter lists. Once a parameter is entered, users can press comma, i.e. the list delimiter, to instantiate the next element.

6.1.3 Language Aspects

When defining a language in MPS, you have to define several language aspects. Some of them are familiar from the disussion above, others haven't been mentioned yet. In the rest of this part of the user guide we'll provide example for all of these aspects; here is a general overview:

■ **Structure** The structure, or abstract syntax or meta model, has been discussed above. In terms of structure, a language concept consists of a name, child concepts, references to other concepts and primitive properties (integer, Boolean, string or an enumeration). A Concept may also extend *one* other concept and implement any number of concept interfaces. In addition, a concept can also have *static* properties, children and references; MPS refers to those as *concept* properties/children/references. The definition of a new concept always starts with the structure.

■ **Editor** The editor aspect defines the concrete syntax, or the projection rules. Currently, each concept has one editor (unless a concept inherits the editor from its super-concept). Interface concepts do not have editors, but they may define editor *components* that can be embedded into other editors. Editors consist of cells, where each cell may contain a constant (i.e. a keyword or a symbol), a property value (such as the name), a child cell, a reference cell or a collection of other cells. In fact, the editor for a concept is defined in two parts: the editor for the concept in the primary editor as well as the concept's representation in the inspector (essentially a properties view). The editor aspect also defines (some) *actions* and keymaps. These determine the reaction of MPS if a certain key is pressed in given cell or when a node is deleted.

■ **Actions** The actions aspect contains mostly wrappers and side transformations (as discussed above).

■ **Behavior** The behavior aspect essentially contains methods defined on concepts. Such methods can implement arbitrary behavior and can be called on each instance of the concept. Note that concept interfaces cannot just declare methods (as in Java), but they can also provide an implementation. In that sense, interfaces are more like Scala traits.

■ **Constraints** The constraints aspects seems to be a collection of relatively independent things. In particular, you define scopes for references (i.e. the set of valid target nodes beyond their type), constraints for properties (value ranges, regular expressions), as well as context dependencies for concepts. The latter allows you to restrict where a concept can be used, for example, an `assert` statement may be restricted to test cases.

■ **Type System** In the type system you can specify typing rules for concepts. These include inference rules (the type of a variable reference is the type of the variable referenced by it), subtyping rules (`int` is a subtype of `float`) and so-called non-typesystem rules. The latter are essentially boolean expressions that evaluate any part of the model. For example, they can be used to check for name uniqueness or naming conventions. The type system aspect may also contain quick fixes that can used to resolve errors raised by type system rules.

■ **Intentions** Intentions are the similar to quick fixes in that they can be activated via `Alt-Enter` and then selected from the menu that pops up. They are different in that an intention is not associated with an error, but just generally with a concept. For example, an intention could allow a user to mark a type as `const` by selecting `Make Const` from the intentions menu (`Alt-Enter`) of a type.

■ **Refactorings** Refactorings are similar to intentions in that they change the structure of a program. They are also typically associated with a specific concept. There are several differences though: a refactoring shows up in the `Refactorings` menu, it can have a keyboard shortcut associated with it, it can be written to be able to handle several nodes at a time, and it can query the user for input before it is executed.

■ **Generator** The generator aspect is used to define the mapping of one concept to another one. MPS relies on multi-stage transformations, where only the last one creates text and all others are tree-to-tree mappings. The generator aspects contains the tree-to-tree mappings, and the textgen aspect (discussed below) handles the final to-text transformation. Generator consist of so-called mapping configurations that contain transformation rules. There are different kinds of rules that transform nodes in different ways. For example, a reduction rule replaces any occurence of a concept with the tree fragment created by the template associated with the reduction rule. Note that you get IDE support for the target language in the transformation template! A generator may also contain procedural mapping scripts.

■ **Textgen** The textgen aspect for a language concept defines the mapping of the concept to a text buffer. This should only be used for the concept of a base language such as Java, C or XML. Any higher-level concepts that create programs expressed in such a base language should use generators (previous paragraph).

■ **Find Usages** The Find Usages aspect supports custom finders: when the `Find Usages` item is selected from the context menu of a node, these custom finders show up there. Instead of finding *all* usages of a node, these custom finders can filter based on the usage context. For example, for local variable, a finder may only show usages where the variable is used on the left of an assignment statement.

■ **Dataflow** The dataflow aspect constructs a dataflow graph for programs. For each language concept, a so-called dataflow builder can be defined whose task it is to construct the dataflow graph fragment for the respective concept. Based on this dataflow graph, a set of dataflow analyses can be performed.

■ **Scripts** The scipts aspect contains migration or enhancement scripts for language concepts. They are useful in handling language migration problems.

6.2 A Hello World Extension

Note: The code for this example can be found in the tutorial in the **mbeddr.tutorial.foreach** language.

In this section we show the complete implementation of a simple C extension. As an example, we introduce a **foreach** statement. It supports conveniently iterating over C arrays. Users have to specify the array over which to iterate, as well as the size of the array, because in C an array cannot be queried for its size. Inside the body of the **foreach**, a variable named **it** acts as a reference to the current iteration's array element. The code below shows a test case that uses a **foreach** statement.

```
test case testForEach {
  int8 sum = 0;
  int8[] array = {1, 2, 3, 4, 5};
  foreach (array sized 5) {
    sum += it;
  }
  assert(0) sum == 15;
}
```

The **foreach** extension should be modular, i.e. it should live in a separate language module so users can decide whether they want to use it in their programs or not. Therefore we create a new language named **ForeachLanguage**. It extends the C core language, since we will refer to concepts defined in C.

In the following we first describe the definition of the **foreach** itself; we look at the definition of its assciated **it** expression in a separate paragraph at the end of this subsection.

■ **Structure/Abstract Syntax** In the new language, we define the **ForeachStatement** concept. It should be usable wherever C expects **Statement**s (i.e. in functions), so **ForeachStatement** extends the **Statement** concept defined in the C core. **ForeachState-**

ments have three children: an **Expression** that represents the array, an **Expression** for the array length, and a **StatementList** for the body. **Expression** and **StatementList** are both defined in the C core. Fig. 6.2 shows a class diagram of the structure and the code below shows the code that defines this structure in MPS.

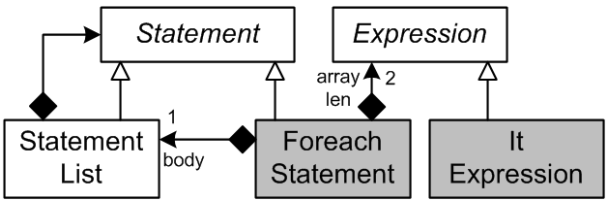


Figure 6.2: UML class diagram showing the structure of the **ForeachLanguage**. Concepts provided by the C core are in white boxes, new concepts are grey.

```
concept ForeachStatement extends Statement
  children:
    Expression     array    1
    Expression     len      1
    StatementList  body     1
  concept properties:
    alias foreach
```

Note how rhe definition of the **ForeachStatement** relies heavily on concepts defined in the C core: it extends **Statement** and it owns two **Expression**s and a **StatementList**. Note that we also define an alias **foreach**. This way, a user can simply type **foreach** in statement context to instantiate a **ForeachStatement**.

■ **Editor/Concrete Syntax** An editor defines the concrete syntax of a concept. The **foreach** editor (Fig. 6.3) consists of a horizontal list of editor cells containing: the **foreach** keyword, the opening parenthesis, the embedded editor of the **array**, the **sized** keyword, the embedded editor of the **len** expression, the closing parenthesis and the editor of the **body**. Note how we simply embed editors of those concepts defined in the C core. We do not need to worry about possible syntactic ambiguities, since those cannot arise in projectional editors.

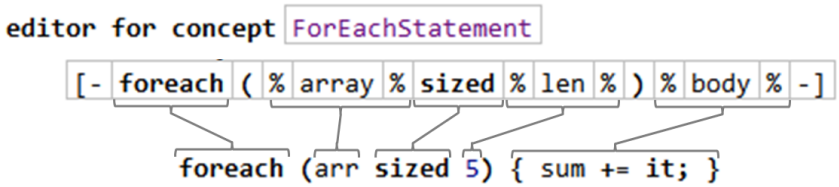


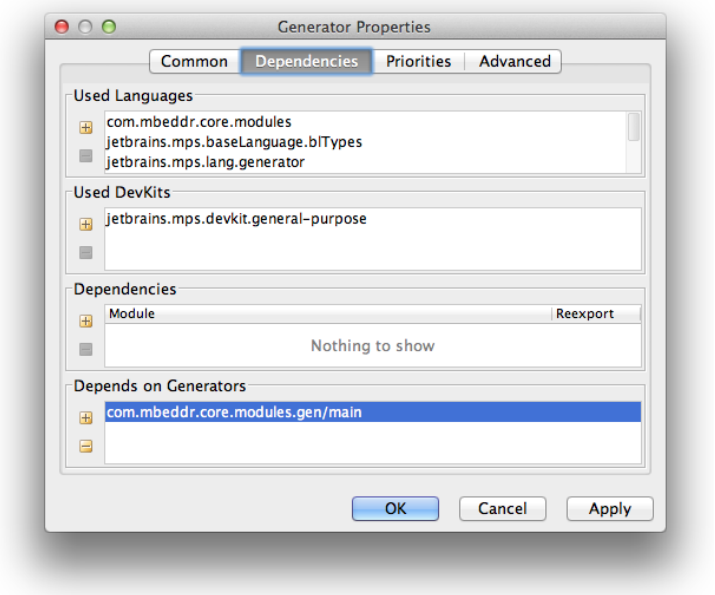
Figure 6.3: *Top:* the editor definition of the **foreach** statement. *Bottom:* An example instance, and how the editor definition relates to it.

■ **Type System** Two type checks must be enforced. The type of the `array` must be `ArrayType`, and the type of `len` must be `int64` (or any of its shorter subtypes). The code below shows the necessary code using MPS' rule-based type system engine. The first line establishes a subtype relation between the type of the `len` expression and `int64`, essentially requiring `len` to be `int64` or a (shorter) subtype. The second line checks that `array` is actually an array type.

```
rule typeof_ForeachStatement applicable for ForeachStatement as fes
  do {
    typeof(fes.len) :<=: <int64>;
    if (!(fes.array.type.isInstanceOf(ArrayType))) {
      error "array required" -> fes.array;
  }
}
```

■ **Generator** The generator reduces a `ForeachStatement` to a regular `for` statement that iterates over the elements with a counter variable `__c` (Fig. 6.4). Inside the `for` body, we create a variable `__it` that refers to the array element at position `__c`. We also copy in all the other statements from the body of the `foreach`. Note that this generator essentially uses macro-style expansion. While this is the simplest type of generator in MPS, there are also more advanced transformations. In particular, there are non-local transformations that support creating new program elements in arbitrary locations in the code, not just in place of the original node.

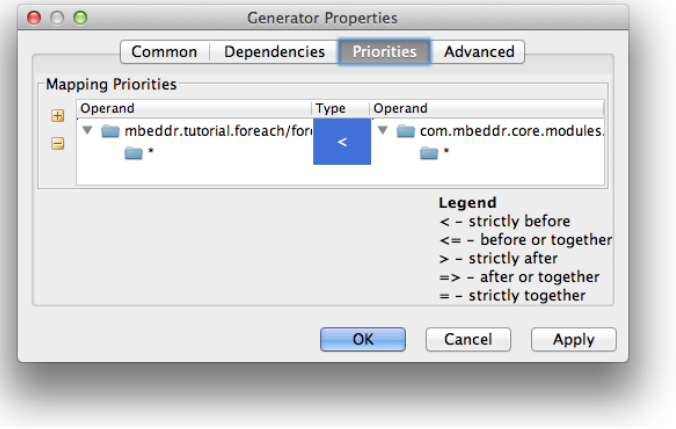
To make this generator work well with the C-to-text generator, we have to make sure that this one runs before the C-to-text generator. To do this we specify a dependency on the `modules.gen` generator:



```
concept     ForEachStatement
inheritors false
condition   <always>
 -->
content node:
void dummy() {
  int8_t[ ] x;
  <TF for ($COPY_SRC$[int64_t] __c = 0; __c < $COPY_SRC$[10]; __c++) { TF>
    $COPY_SRC$[int8_t] __it = $COPY_SRC$[x][__c];
    $COPY_SRCL$[int8_t x; ]
  }
}
```

Figure 6.4: The **foreach** generator. The header defines that the rule applies to **ForeachStatement**s. A **ForeachStatement** is replaced by the part framed by **<TF .. TF>** when the template is executed. The **dummy** function around it provides context and is not part of the generator output. The **COPY_SRC** and **COPY_SRCL** macros contain expressions (not shown in the screenshot) that determine with what the example nodes (**10**, **int8 x;**) are replaced during transformation execution. For example, the expression behind the **COPY_SRC[x]**, is **node.array**, replacing the dummy **x** with the content in the **foreach**'s **array** child.

We then add a **runs strictly before** priority from our generator to the **modules.gen** generator. This ensures that ours runs before the C-to-text generator.



■ **The `it` Expression** In terms of abstract syntax, the **ItExpression** must inherit from the C's **Expression** concept to make it usable in an expression context. We define

an alias **it**. The editor is trivial, it consists of one cell with the keyword **it**. The type of **it** must be the base type of the **array** (e.g. **int** in case of **int[]**). The typesystem code is below. The first line retrieves the base type of the array we iterate over. Since we know that the **it** expression must only be used inside a **ForeachStatement**, we can ascend the containment tree until we find one, get its **array** child, take the type of it, case it to an **ArrayType** and then get its **baseType**. The second line establishes a relation between the type of the **it** expression and the **bt** we just retrieved.

```
rule typeof_ItExpression applicable ItExpression as it
  do {
    node<Type> bt = typeof(it.ancestor<ForeachStatement>.array) : ArrayType.baseType;
    typeof(it) :==: bt.copy;
  }
```

We also need a **can be child** constraint that prevents the use of the **it** expression outside of the **foreach** In case of **it** we check that we are below a **StatementList** and a **ForeachStatement**. The code is below.

```
concept constraints ItExpression {
  can be child
    (context, scope, parentNode, link, childConcept)->boolean {
      parentNode.ancestor<ForeachStatement>.isNotNull &&
      parentNode.ancestor<StatementList>.isNotNull;
  }
}
```

Finally, we have to define the generator. The **foreach** generator (discussed above) defines a local variable **__it** in the body of the generated **for** loop. We can thus simply reduce an **ItExpression** into a **LovalVariableReference** that refers to **__it**. Fig. 6.5 shows the reduction rule.

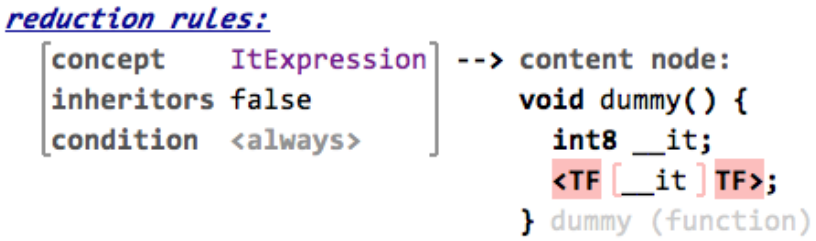


Figure 6.5: The reduction rule for the **it** expression. Notice how the template fragment only contains the **__it** variable reference, so only that reference is generated. The rest is scaffolding. It is necessary because, if we don't have a variable **__it**, we couldn't write the reference to that variable in the template. This is why we have function and a local variable declaration *in the template* even though it is not generated.

6.3 Overview over C Extensions

In this section we discuss in which particular ways C needs to be extensible in order to create meaningful language extensions. We then discuss a set of examples and how to build them. This section is intended as an overview; more detailed examples follow in the subsequent sections. We start with a brief overview over the modular nature of the C implementation itself.

6.3.1 C is modular itself

C can be partitioned into expressions, statements, functions, etc. We have factored these parts into separate language modules to make each of them reusable without pulling in all of C. The **expressions** language is the most fundamental language. It depends on no other language and defines the primitive types, the corresponding literals and the basic operators. Support for pointers and user defined data types (**enum, struct, union**) is factored into the **pointers** and **udt** languages, respectively. **statements** contains the procedural part of C, and the **modules** language covers modularization. Fig. 6.6 shows an overview over some of the languages and constructs.

6.3.2 Ways to Extend C

To be able to have really extensible language, several diffent ways of extending C are necessary. These include:

- **Top Level Constructs** Top level constructs (on the level of functions or **struct** declarations) are necessary. This enables the integration of test cases, state machines, or interfaces and components.

- **Statements** New statements, such as **assert** or **fail** statements in test cases, must be supported. If statements introduce new blocks, then variable visibility and shadowing must be handled correctly, just as in regular C. Statements may have to be restricted to a specific context; for example the **assert** or **fail** statements must *only* be used in test cases and not in any other statement list.

- **Expressions** New kinds of expressions must be supported. An example is a decision table expression that represents a two-level decision tree as a two dimensional

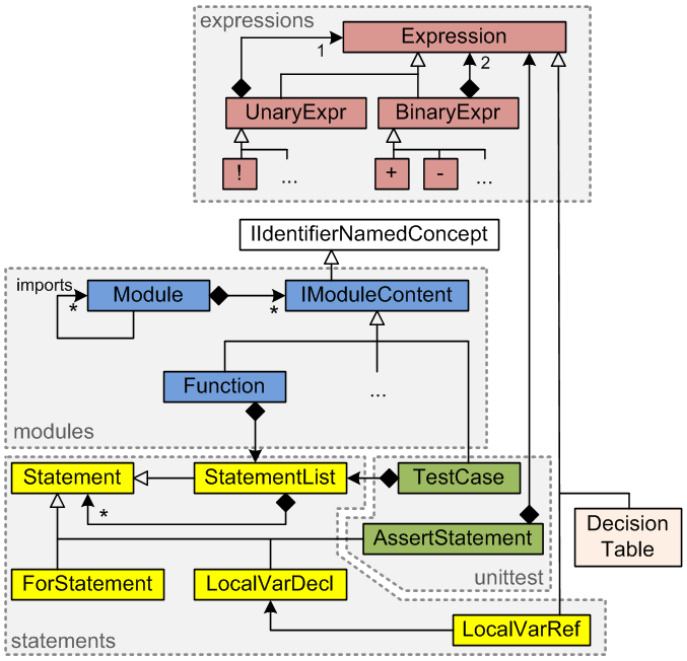


Figure 6.6: Anatomy of the mbeddr language stack: the diagram shows some of the language concepts, their relationships and the languages that contain them.

table. Many more examples exist, expressions are maybe the most important kind of extension.

- **Types and Literals** New types, e.g. for matrices, complex numbers or quantities with physical units must be supported. This also requires defining new operators and overriding the typing rules for existing ones. New literals may also be required: for example, physical units could be attached to number literals (as in `10kg`).

- **Transformation** Alternative transformations for existing language concepts must be possible. For example, in a module marked as `safe`, `x + y` may have to be translated to `addWithBoundsCheck(x, y)`, a call to an `inline` function that performs bounds-checking besides the addition.

- **Meta Data Decoration** It should be possible to add meta data such as trace links to requirements or product line variability constraints to arbitrary program nodes, without changing the concept of the node.

- **Restriction** It should be possible to define contexts that restrict the use of certain language concepts. Like any other extension, such contexts must be definable *after*

the original language has been implemented, without invasive change. For example, the use of pointer arithmetic should be prohibited in modules marked as *safe* or the use of real numbers should be prohibited in state machines that are intended to be model checked (model checkers do not support real numbers).

In the following subsections we provide examples for each of these ways.

6.3.3 Top-Level Constructs: Test Cases

Unit Tests are new top-level constructs (Fig. 6.7) introduced in a separate *unittest* language that extends the C core. They are like **void** functions without arguments. The *unittest* language also introduces **assert** and **fail** statements, which can only be used inside test cases. In this section we illustrate the implementation of the **test case** construct as well as of the **assert** and **fail** statements available inside test cases.

```
module UnitTestDemo imports Sensors {
  exported test case sensorReadTest {
    assert(0) readSensor() > 0;
    assert(1) readSensor() < 1000;
  }
}
```

```
#include "Sensor.h"
int8_t UnitTestDemo_test_sensorReadTest() {
 int8_t __failures = 0;
 printf("running test @UnitTestDemo:test_sensorReadTest:0\n");
 if ( !(Sensor_readSensor() > 0) ) {
  __failures++;
  printf("FAILED: @UnitTestDemo:test_sensorReadTest:1\n");
  printf("  testID = %d\n",0);
 }
 if ( !(Sensor_readSensor() < 1000) ) { ... }
 return __failures;
}
```

Figure 6.7: The *unittest* language introduces test cases as well as **assert** and **fail** statements which can only be used inside of a test case. Test cases are transformed to functions, and the **assert** statements become **if** statements with a negated condition. The generated code also counts the number of failures so it can be reported to the user via a binary's exit value.

■ **Structure** **Module**s own a collection of **IModuleContent**s, an interface that defines the properties of everything that can reside directly in a module. All top-level constructs such as **Function**s implement **IModuleContent**. **IModuleContent** extends **IIdentifier-NamedConcept** interface, which provides a **name** property. **IModuleContent** also defines a Boolean property **exported** that determines whether the respective module content is visible to modules that import this module. This property is queried by the scoping rules that determine which elements can be referenced. Since the **IModuleContent** interface can also be implemented by concepts in other languages, new top level constructs such as the **TestCase** in the **unittest** language can implement this interface, as long

as the respective language has a dependency on the **modules** language, which defines **IModuleContent**. Fig. 6.6 shows some of the relevant concepts and languages.

■ **Constraints** A test case contains a **StatementList**, so any C statement can be used in a test case. **StatementList** becomes available to the unit test language through its dependency on the **statements** language. **unittest** also defines new statements: **assert** and **fail**. They extend the abstract **Statement** concept defined in the **statements** language. This makes them valid in *any* statement list, for example in a function body. This is undesirable, since the transformation of **assert**s into C depends on them being used in a **TestCase**. To enforce this, a **can be child** constraint is defined. This constraint restricts an **AssertStatement** to be used only inside a **TestCase** by checking that at least one of its ancestors is a **TestCase**:

```
  concepts constraints AssertStatement {
    can be child
      (context, scope, parentNode, link, childConcept)->boolean {
        parentNode.ancestor<TestCase>.isNotNull;
      }
}
```

■ **Transformation** The new language concepts in **unittest** are reduced to C concepts: the **TestCase** is transformed to a **void** function without arguments and the **assert** statement is transformed into a **report** statement defined in the **util** language. The **report** statement, in turn, it is transformed into a platform-specific way of reporting an error (console, serial line or error memory). The code below shows an example of this two-step process.

```
test case exTest {
  int x = add(2, 2);
  assert(0) x == 4;

}
```

```
void test_exTest {
  int x = add(2, 2);
  report
    test.FAIL(0)
    on !(x == 4);}
```

```
void test_exTest {
  int x = add(2, 2);
  if (!(x == 4)) {
    printf("fail:0");
} }
```

6.3.4 Statements: Safeheap Statement

Note: The code for this example can be found in the tutorial in the **mbeddr.tutorial.heap** language.

We have seen the basics of integrating new statements in the previous section where **assert** and **fail** extended the **Statement** concept inherited from the C core languages. In this section we focus on statements that require handling local variable scopes and visibilities. We implement a part of the **safeheap** statement mentioned earlier (see

```
exported test case testSafeHeap {
  safeheap [ Trackpoint* tp1                          ]{
           [ Trackpoint* tp2 = makeTP(100, 100)       ]
           [ int8* y                                  ]
    tp1->alt = 100 m;
    assert(0) tp1->alt == 100 m;

    *y = 0;
    assert(1) *y == 0;

    assert(2) tp2->alt == 100 m;
    assert(3) tp2->speed == 100 mps;

  }
} testSafeHeap(test case)
```

Figure 6.8: A **safeheap** statement declares heap variables which can only be used inside the body of the statement. When the body is left, the memory is automatically freed.

```
editor for concept SafeHeapBlock
  node cell layout:
    [- safeheap (> % vars % <) % body % -]
       /empty cell: <constant>
```

Figure 6.9: The editor for the safeheap statement; details are discussed in the running text.

Fig. 6.8), which automatically frees dynamically allocated memory. The variables introduced by the **safeheap** statement must only be visible inside its body and they have to shadow variables of the same name declared in outer scopes (such as the **a** declared in the second line of the **measure** function in Fig. 6.8).

■ **Structure** The **safeheap** statement extends **Statement**. It contains a **StatementList** as its body, as well as a list of **SafeHeapVar**s. These extend **LocalVarDecl**, so they fit with the existing mechanism for handling variable shadowing (explained below).

■ **Editor** The safeheap editor is shown in Fig. 6.9. It comprises a collection cell (indent) that contains the keyword, a vertical collection of the variables (with the **draw-brackets** style option set) plus the body (which brings its own curlies and line break policy).

There is an important twist to the editor definition and the way it can be used, though. The **vars** collection contains a list of **SafeHeapVar** nodes. So, when the user edits this collection, he has to instantiate a new **SafeHeapVar**, for example using the alias. However, as you will find out when using the editor, you can just select the type of that variable, and once you have selected one (by typing or by selection from the code completion menu) a new **SafeHeapVar** is created and that type is set as its type. This is achieved using a wrapper action:

```
substitute actions wrap_SafeHeapVarWithType

substituted node: SafeHeapVar
  condition :
    (..., parentNode, ...)->boolean {
      parentNode.ancestor<concept = SafeHeapBlock, +>.isNotNull;
    }

  actions :
    add custom items  (output concept: SafeHeapVar)
      wrap Type
        wrapper block
          (..., parentNode, nodeToWrap, ...)->node<SafeHeapVar> {
            node<SafeHeapVar> v = new node<SafeHeapVar>();
            v.type = nodeToWrap;
            v;
          }
        return small part
          false
```

This action says: whenever a **SafeHeapVar** is expected somewhere (**substituted noe**), the user can actually also enter a **Type** (**wrap**). In that case, we still create a **SafeHeapVar** (**output concept**), and we get it via the code in the wrapper block. There we create an instance of **SafeHeapVar**, set the type to be the **nodeToWrap** and then return that newly created variable.

Actions like this play an important role in making the MPS/mbeddr editor usable!

■ **Behaviour** **LocalVarRef**s are expressions that reference **LocalVarDecl**. A scope constraint, a mechanism provided by MPS, determines the set of visible variables for a given **LocalVarRef**. We implement this constraint by plugging into mbeddr's generic local variable scoping mechanism using the following approach. The constraint ascends the containment tree until it finds a node which implements **ILocalVarScopeProvider** and calls its **getLocalVarScope** method. A **LocalVarScope** has a reference to an outer scope, which is set by finding *its* **ILocalVarScopeProvider** ancestor, effectively building a hierarchy of **LocalVarScope**s. To get at the list of the visible variables, the **LocalVarRef** scope constraint calls the **getVisibleLocalVars** method on the innermost **LocalVarScope** object. This method returns a flat list of **LocalVarDecl**s, taking into account that variables owned by a **LocalVarScope** that is *lower* in the hierarchy shadow variables of the same name from a *higher* level in the hierarchy. So, to plug the **SafeHeapStatement** into

this mechanism, it has to implement **ILocalVarScopeProvider** and implement the two methods shown in the code below. A **safeheap** statement implements the two methods declared by the **ILocalVarScopeProvider** interface. **getContainedLocalVariables** returns the **LocalVarDecl**s that are declared between the parentheses (see Fig. 6.8). **getLocalVarScope** constructs a scope that contains these variables and then builds the hierarchy of outer scopes by relying on its ancestors that also implement **ILocalVarScopeProvider**. The index of the statement that contains the reference is passed in to make sure that only variables declared *before* the reference site can be referenced.

```
public LocalVarScope getLocalVarScope(node<> ctx, int stmtIdx) {
  LocalVarScope scope = new LocalVarScope(getContainedLocalVariables());
  node<ILocalVarScopeProvider> outerScopeProvider = this.ancestor<ILocalVarScopeProvider>;
  if (outerScopeProvider != null)
    scope.setOuterScope(outerScopeProvider.
                  getLocalVarScope(this, this.index));
  return scope;
}

public sequence<node<LocalVariableDecl>> getContainedLocalVars() {
  this.vars;
}
```

■ **Type System** To make the **safeheap** statement work correctly, we have to ensure that the variables declared and allocated in the *safeheap* statement do not escape from its scope. To prevent this, an error is reported if a reference to a **safeheap** variable is passed to a function. The code elow shows the code: theq type system rule reports an error if a reference to a local variable declared and allocated by the **safeheap** statement is used in a function call.

```
checking rule check_safeVarRef for concept = LocalVarRef as lvr {
    boolean isInSafeHeap = lvr.ancestor<SafeHeapStatement>.isNotNull;
    boolean isInFunctionCall = lvr.ancestor<FunctionCall>.isNotNull;
    boolean referencesSafeHeapVar = lvr.var.parent.isInstanceOf(SafeHeapStatement);
    if (isInSafeHeap && isInFunctionCall && referencesSafeHeapVar)
         error "cannot pass a safe heap var to a function" -> lvr;
}
```

6.3.5 Expressions: Decision Tables

Decision Tables are an example of extending expressions. An example is shown in Fig. 6.10. A decision table represents nested **if** statements. It is evaluated to the value of the first cell whose column and row headers are **true** (the evaluation order is left to right, top to bottom). A default value (**FAIL**) is specified to handle the case where none of the column/row header combinations is **true**. Since the compiler and IDE have to compute a type for expressions, the decision table specifies the type of its result values explicitly (**int8**).

```
enum mode { MANUAL; AUTO; FAIL; }
mode nextMode(mode mode, int8_t speed) {
  return mode, FAIL
```

	mode == MANUAL	mode == AUTO
speed < 30	MANUAL	AUTO
speed > 30	MANUAL	MANUAL

```
}
```

```
typedef enum __MODE {MANUAL, AUTO, FAIL } _MODE;
_MODE nextMode( _MODE mode, int8_t speed ) {
 if (current == MANUAL) {
  if (speed <= 30) {return MANUAL;}
  if (speed >= 30 && speed < 50) {return MANUAL;}
 }
 if (current == AUTO) { ... }
 return FAIL;
}
```

Figure 6.10: A decision table evaluates to the value in the cell for which the row and column headers are **true**, a default value otherwise (**FAIL** in the example). By default, a decision table is translated to nested **if**s in a separate function. The figure shows the translation for the common case where a decision table is used in a **return**. This case is optimized to not use the indirection of an extra function.

Expressions are different from statements in that they can be evaluated to a *value* as the program executes. During editing and compilation, the *type* of an expression is relevant for the static correctness of the program. So extending a language regarding expressions requires extending the type system rules as well.

■ **Structure** The decision table extends the **Expression** concept defined in the **expressions** language. Decision tables contain a list of expressions for the column headers, one for the row headers and another one for the result values. It also contains a child of type **Type** to declare the type of the result expressions, as well as a default value expression. The concept defines an alias **dectab** to allow users to instantiate a decision table in the editor. Obviously, for non-textual notations such as the table, the alias will be different than the concrete syntax (in textual notations, the alias is typically made to be the same as the "leading keyword", e.g. **assert**).

■ **Editor** Defining a tabular editor is straight forward: the editor definition contains a **table** cell, which delegates to a Java class that implements **ITableModel**. This is similar to the approach to the approach used by Java Swing. It provides methods such as **getValueAt(int row, int col)** or **deleteRow(int row)**, which have to be implemented for any specific table-based editor. To embed another node in a table cell (such as the expression in the decision table), the implementation of **getValueAt** simply returns this node.

■ **Type System** As mentioned above, MPS uses unification in the type system. Language concepts specify type equations that contain type literals (such as **boolean**) as well as type variables (such as **typeof(dectab)**). The unification engine then tries to as-

sign values to the type variables so that all applicable type equations become **true**. New language concepts contribute additional type equations. Fig. **??** shows those for decision tables. New equations are solved along with those for existing concepts. For example, the typing rules for a **ReturnStatement** ensure that the type of the returned expression is the same or a subtype of the type of the surrounding function. If a **ReturnStatement** uses a decision table as the returned expression, the type calculated for the decision table must be compatible with the return type of the surrounding function.

```
  // the type of the whole decision table expression
  // is the type specified in the type field
typeof(dectab) :==: typeof(dectabc.type);
  // for each of the expressions in
  // the column headers, the type must be Boolean
foreach expr in dectab.colHeaders {
  typeof(expr) :==: <boolean>;
}
  // ... same for the row headers
foreach expr in dectabc.rowHeaders {
  typeof(expr) :==: <boolean>;
}
  // the type of each of the result values must
  // be the same or a subtype of the table itself
foreach expr in dectab.resultValues {
  infer typeof(expr) :<=: typeof(dcectab);
}
  // ... same for the default
typeof(dc.def) :<=: typeof(dectab);
```

6.3.6 Types and Literals: Physical Units

Physical Units are new types that also specify a physical unit in addition to their actual data type (see Fig. 6.11). 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 correct type checks for types with units. The type system also performs unit computations to deal correctly with **speed = length/time**, for example.

■ **Structure** Derived and convertible **UnitDeclaration**s are **IModuleContents**. Derived unit declarations specify a name (**mps**, **kmh**) and the corresponding SI base units (**m**, **s**) plus an exponent; a convertible unit declaration specifies a name and a conversion formula. The backbone of the extension is the **UnitType** which is a composite type that has another type (**int**, **float**) in its **valueType** slot, plus a unit (either an SI base unit or a reference to a **UnitDeclaration**). It is represented in programs as **baseType/unit/**. We also provide **LiteralWithUnit**s, which are expressions that contain a **valueLiteral** and, like the **UnitType**, a unit (so we can write **100 kmh**).

```
derived unit mps = m s^-1 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;
}
```

Figure 6.11: The *units* extension ships with the SI base units. Users can define derived units (such as the **mps** in the example) as well as convertible units that require a numeric conversion for mapping 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 reported if incompatible units 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.

■ **Scoping** **LiteralWithUnit**s and **UnitType**s refer to a **UnitDeclaration**, which is a module content. According to the visibility rules, valid targets for the reference are the **UnitDeclaration**s in the same module, and the *exported* ones in all imported modules. This rule applies to *any* reference to *any* module contents, and is implemented generically in mbeddr. The code below shows the code for the scope of the reference to the **UnitDeclaration**. We use an interface **IVisibleNodeProvider**, (implemented by **Module**s) to find all instances of a given type. The implementation of **visibleContentsOfType** simply searches through the contents of the current and imported modules and collects instances of the specified concept. The result is used as the scope for the reference.

```
link {unit} search scope:
    (model, scope, refNode, enclosingNode, operationContext)
                              ->sequence<node<UnitDeclaration>> {
      enclosingNode.ancestor<IVisibleNodeProvider>.
              visibleContentsOfType(concept/UnitDeclaration/); }
```

■ **Type System** We have seen how MPS uses equations and unification to specify type system rules. However, there is special support for binary operators that makes overloading for new types easy: overloaded operations containers essentially specify 3-tuples of *(leftArgType, rightArgType, resultType)* plus applicability conditions to match type patterns and decide on the resulting type. Typing rules for new (combinations of) types can be added by specifying additional 3-tuples.

The code below shows the overloaded rules for C's **MultiExpression** (the language concept the implements the multiplication operator ∗) when applied to two **UnitType**s: the result type will be a **UnitType** as well, where the exponents of the SI units are added. This code overloads the **MultiExpression** to work for **UnitType**s. In the **is applicable** section we check whether there is a typing rule for the two value types (e.g. **int ∗ float**). This is achieved by trying to compute the resulting value type. If none is found, the types cannot be multiplied. In the computation of the **operation type** we create a new **UnitType** that uses the **resultingValueType** as the value type and then computes the resulting unit by adding up the exponents of component SI units of the two operand types.

```
operation concepts: MultiExpression
  left operand type: new node<UnitType>()
  right operand type: new node<UnitType>()
is applicable:
  (op, leftOpType, rightOpType)->boolean {
    node<> resultingValueType = operation type(op,
                  leftOpType.valueType , rightOpType.valueType );
    resultingValueType != null; }
operation type:
  (op, leftOpType, rightOpType)->node<> {
     node<> resultingValueType = operation type(op,
               leftOpType.valueType,  rightOpType.valueType );
     UnitType.create(resultingValueType,
                      leftOpType.unit.toSIBase().add( rightOpType.unit.toSIBase(), 1 ) );
  }
```

While any two units can legally be used with ∗ and / (as long as we compute the resulting unit exponents correctly), this is not true for + and -. There, the two operand types must be the same (in terms of their representation in SI base units). We express this by using the following expression in the **is applicable** section:
leftOpType.unit.isSameAs(rightOpType.unit).

The typing rule for the **LocalVariableDeclaration** requires that the type of the **init** expression must be the same or a subtype of the **type** of the variable. To make this work correctly, we have to define a type hierarchy for **UnitType**s. We achieve this by defining the supertypes for each **UnitType**: the supertypes are those **UnitType**s whose unit is the same, and whose **valueType** is a supertype of the current **UnitType**'s value type. The code below shows the rule. This typing rule computes the direct supertypes of a **UnitType**. It iterates over all immediate supertypes of the current **UnitType**'s value type, wrapped into a **UnitType** with the same unit as the original one.

```
subtyping rule supertypeOf_UnitType
               for concept = UnitType as ut {
  nlist<> res = new nlist<>;
  foreach st in immediateSupertypes(ut.valueType) {
    res.add(UnitType.create(st, ut.unit.copy));
  }
  return res;
}
```

6.3.7 Meta Data: Requirements Traces

Annotations are concepts whose instances can be added as children to a node N without this being specified in the definition of N's concept. While structurally the annotations are children of the annotated node, the editor is defined the other way round: the annotation editor delegates to the editor of the annotated element. This allows the annotation editor to add additional syntax *around* the annotated element. Optionally, it is possible to explicitly restrict the concepts to which a particular annotation can be attached. We use annotations in several places: the **safe** annotation discussed in the previous section, the requirements traces and the product line variability presence conditions.

■ **Structure** We illustrate the annotation mechanism based on the requirements traces. The code below shows the structure. Notice how it extends the MPS-predefined concept **NodeAttribute** (it sould be named **Node- Annotation**). It also specifies a **role**, which is the name of the property that is used to store **TraceAnnotation**s under the annotated node. Annotations have to extend the MPS-predefined concept **NodeAttribute**. They can have an arbitrary child structure (**tracekind**, **refs**), but they have to specify the **role** (the name of the property that holds the annotated child under its parent) as well as the **attributed** concept: the annotations can only be attached to instances of this concept (or subconcepts).

```
concept TraceAnnotation extends NodeAttribute implements <none>
  children:
    TraceKind        tracekind   1
    TraceTargetRef   refs        0..n
  concept properties:
    role = trace
  concept links:
    attributed = BaseConcept
```

■ **Editor** As mentioned above, in the editor, annotations look as if they *surrounded* their parent node (although they are in fact children). Fig. 6.12 shows the definition of the editor of the requirements trace annotation (and an example is shown in Fig. 2.1): it puts the trace to the right of the annotated node. Since MPS is a projectional editor, there is base-language grammar that needs to be made aware of the additional syntax in the program. This is key to enabling arbitrary annotations on arbitrary program nodes.

Annotations are typically attached to a program node via an intention. Intentions are an MPS editor mechanism: a user selects the target element, presses **Alt-Enter** and selects **Add Trace** from the popup menu. The code below shows the code for the intention that attaches a requirements trace. An intention definition consists of three parts. The **description** returns the string that is shown in the intentions popup menu.

```
editor for concept TraceAnnotation
  node cell layout:
    [> [> attributed node <] ?[> % tracekind % F(> % refs % <) <] <]
```

Figure 6.12: The editor definition for the **ReqTrace** annotation (an example trace annotation is shown in Fig. 2.1). It consists of a vertical list **[/ .. /]** with two lines. The first line contains the reference to the requirement. The second line uses the **attributed node** construct to embed to the editor of the program node to which this annotation is attached. So the annotation is always rendered right on top of whatever syntax the original node uses.

The **isApplicable** section determines under which conditions the intention is avavailable in the menu — in our case, we can only add a trace if there is no trace yet on the target node. Finally, the **execute** section performs the action associated with the intention. In our case we simply put an instance of **TraceAnnotation** into the **@trace** property of the target node.

```
intention addTrace for BaseConcept {
  description(node)->string {
    "Add Trace";
  }
  isApplicable(node)->boolean {
    node.@trace == null;
  }
  execute(editorContext, node)->void {
    node.@trace = new node<TraceAnnotation>();
  }
}
```

6.3.8 Alternative Transformations: Range Checking

■ **Safe Modules** Safe modules restrict C to help prevent writing risky code. For example, runtime range checking is performed for arithmetic expressions and assignments. The **safemodules** language defines an *annotation* to mark **Modules** as safe (we will discuss annotations in the next subsection). If a module is safe, the binary operators such as **+** or ∗ are replaced with calls to functions that, in addition to performing the addition or multiplication, perform a range check.

■ **Transformation** The transformation that replaces the binary operators with function calls is triggered by the presence of this annotation on the **Module** which contains the operator. Fig. 6.13 shows the code. The **@safeAnnotation != null** checks for the presence of the annotation. MPS uses priorities to specify relative orderings of transfor-

```
concept    PlusExpression
condition  (node, genContext, operationContext)->boolean {
             node.ancestor<ImplementationModule>.@safeAnnotation != null;
           }
  -->
module dummy imports arithmeticOps {
  void dummy() {
    <TF addWithRangeCheck($COPY_SRC$[1], $COPY_SRC$[2]) ]TF>;
} }
```

Figure 6.13: This *reduction rule* transforms instances of **PlusExpression** into a call to a library function **addWithRangeChecks**, passing in the left and right argument of the **+** using the two **COPY_SRC** macros. The **condition** ensures that the transformation is only executed if the containing **Module** has a **safeAnnotation** attached to it. A transformation priority defined in the properties of the transformation makes sure it runs before the C-to-text transformation.

mations, and MPS then calculates a global transformation order for any given model. We use a priority to express that this transformation runs *before* the final transformation that maps the C tree to C text for compilation.

6.3.9 Restriction: Preventing Use of Reals Numbers

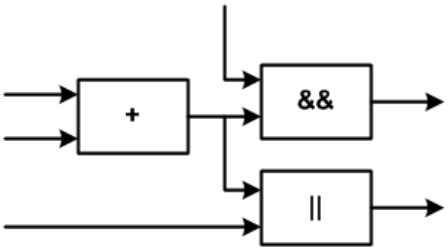
We have already seen in earlier how constraints can prevent the use of specific concepts in certain contexts. We use the same approach for preventing the use of real number types inside model-checkable state machines: a **can be ancestor** constraint in the state machine prevents instances of **float** in the state machine if the **verifiable** flag is set.

6.4 Data Flow Blocks

Note: The code for this tutorial is in the **com.mbeddr.blocks** language. Please open the project and click around in the language as you read this tutorial, since not every detail of the implementation is explained here.

6.4.1 An Example

Imagine a language for dataflow programming as shown in the image below. Defining such systems consists of two steps. The first one is the definition of the blocks – ports, data types and the logic inside them. The second aspect is the wiring of block instances to create a complete system.



The first aspect can be done textually very well and will be discussed in this tutorial (the second aspect is something for the later time when MPS will support graphical notations). Here is an example program that defines the `+` block. This block has **`in`** ports, **`out`** ports as well as the logic dependent on its state[5]:

```
module Blocks imports nothing {

  block Adder [port in int8 a    ] {
              |port in int8 b    |
              |port out int8 sum |
              []
    state On { sum = a + b; }
    state Off { sum = 0; }
  }

}
```

Below is a more interesting block that integrates an input **`signal`**. It has a configuration property **`avgOver`** that determines how many values the integrator should integrate over. It also has internal data structures to manage the rolling buffer necessary for the integration. Finally, the example shows how a block can call out to a C function:

[5]The state simply represents operation modes and can be switched externally. For example, a block may have the **`on`** or **`off`** states.

```
#define MAX_INTEGRATION_SIZE = 20;

block Integrator [port in int16 signal      ]              {
                 [port out int16 integrated ]
                 [property int8 avgOver     ]
                 [var int16[MAX_INTEGRATION_SIZE] values]
                 [var int8 index            ]
  state On {
    values[index] = signal;
    index++;
    if (index > avgOver) {
      index = 0;
    } if
    integrated = avg(values, avgOver);
  }
  state Off { integrated = signal; }
}

int16 avg(int16[MAX_INTEGRATION_SIZE] data, int8 size) { ... }
```

6.4.2 The Outer Structure of Blocks

In this subsection we discuss the outer structure of blocks (i.e. the blocks themselves, ports, properties, variables and states). The necessary extensions to write the application logic are discussed in the next subsection.

■ **Structure** Let us look at the structure of the implementation of blocks. A **Block** implements the **IModuleContent** interface so it can be used inside of modules. It also owns a collection of **IBlockInterfaceElement**s (the super type of **Port**s and **Property**s), a collection of **Variable**s as well as a collection of **BlockState**s. It has an alias **block** so the it can be instantiated easily.

```
concept Block extends BaseConcept
              implements IModuleContent

  children:
    IBlockInterfaceElement interfaces   0..n specializes: <none>
    Variable               variables    0..n specializes: <none>
    BlockState             states       0..n specializes: <none>

  concept properties:
    alias = block
```

A **Port** implements **IBlockInterfaceElement**, which in turn extends **IIdentifierNamedConcept** so ports have a name. A port also implements **ITyped**, an interface defined by the **core.expressions** language that adds a **type** property to the element (rememeber from the example above that a port has a type such as **int8**). A port also has a direction

property which is represented as an **enum**:

```
enumeration datatype PortDirection

  member type : integer
  no default : false
  default : in
  member identifier : derive from presentation

  value 0   presentation in     (default)
  value 1   presentation out
```

The second kind of block interface, the **Property**, looks essentially similar to a **Port** – it just doesn't have a direction property. The same is true for the **Variable**, even though it is not part of the block interface (only of its implementation).

The implementation also consists of several **BlockState**s. A **BlockState** also has a name (by implementing **IIdentifierNamedConcept**) as well as a **BlockStateImpl**. While this has been not shown in the examples above, a state, instead of implementing its own behavior, can express that its behavior is the same as that of another state (**state Standby = On**). **BlockStateImpl** is abstract and has two subconcepts: **CalcMethodBSI** and **SameAsOtherStateBSI**. **CalcMethodBSI** contains a **StatementList** to contain actual application logic. **SameAsOtherStateBSI** has reference to another **BlockState**:

```
concept CalcMethodBSI extends BlockStateImpl
  children:
    StatementList body 1

  concept properties:
    alias = {
```

```
concept SameAsOtherStateBSI extends BlockStateImpl

  references:
    BlockState state 1

  concept properties:
    alias = =
```

■ **Editor** The editors for ports, properties and variables are all essentially the same: an indent collection of a keyword, the type and the name, plus the direction in case of the **Port**. Here is the editor for the port:

```
editor for concept Port
  node cell layout:
    [- port { dir } % type % { name } -]
```

For the state implementation the editor is either just the body (`%body%`) or the reference to the other state (`[- = (%state%->name) -]`). That other state is represented by its name. The only really interesting editor is the one for the `Block` itself. Here is the code:

```
editor for concept Block
  node cell layout:
    [-
    block { name } [/                                    {
                        (> % interfaces %                <)
                           /empty cell: <constant>
                        (> % variables %                 <)
                           /empty cell: <constant>
                        /]
      (- % states % /empty cell: <constant> -)
    }
    -]
```

On its outermost level, it consists of an indent collection (`[- .. -]`). It contains the keyword, the name, and then a vertical collection (`[/ .. /]`) which in turn contains a vertical child collection for the interfaces (ports and properties) and variables. These child collections have their `draw-brackets` style set to true (this renders the brackets around the collection, see examples above). On the right of the vertical collection we have an opening brace, the collection of states and a closing brace. Notice that the states are projected as an indent collection even though in the example we see that they are projected below each other. This is achieved by setting three style attributes on the indent collection cell:

```
indent-layout-on-new-line : true
indent-layout-new-line-children : true
indent-layout-indent : true
```

■ **Type System** The variables, ports and properties must have unique names for each block. While this constraint could be implemented manually, there is direct support available in mbeddr. The `Block` implements the `IContainerOfUniqueNames` interface. This interface requires the implementation of a behavior method `getUniquelyNamedElements` that returns the colleciton of elements whose contents must be named uniquely. The check for uniqueness is then performed automatically in this collection. Here is the implementation for the `Block`:

```
public sequence<node<INamedConcept>> getUniquelyNamedElements()
      overrides IContainerOfUniqueNames.getUniquelyNamedElements {
  nlist<INamedConcept> res = new nlist<INamedConcept>;
  res.addAll(this.interfaces);
```

```
  res.addAll(this.variables);
  res;
}
```

We do not have to write typing rules for variables, ports and properties since their is already a typesystem rule for the interface **ITyped**:

```
rule typeof_ITyped {
  applicable for concept = ITyped as it
  overrides false

  do {
    typeof(it) :==: typeof(it.type);
  }
}
```

■ **Generator** From a block, we generate the following artifacts:

- a **struct** representing the collection of input ports
- a **struct** representing the collection of output ports
- a **struct** representing the data held by a block instance (i.e. properties and variables)
- an **enum** representing the states
- and a function that contains the block behavior.

All of these artifacts must replace the original block, so we can use a reduction rule. To organize the resulting code nicely we use a **section** that contains all these things. So here is the generator code, we'll explain it below.

```
concept    Block
inheritors false
condition  <always>
--> content node:
    module Dummy imports nothing {
      <TF section $[blockSection] {
        enum $[states] {
          $LOOP$[$[literal]; ]
        }

        struct $[input] {
          $LOOP$[$COPY_SRC$[int8 ]$[x]; ]
        };

        struct $[output] {
          $LOOP$[$COPY_SRC$[int8 ]$[x]; ]
        };

        struct $[data] {
          states state;
          $LOOP$[$COPY_SRC$[int8 ]$[var]; ]
          $LOOP$[$COPY_SRC$[int8 ]$[prop]; ]
        };

        void $[processBlock](input* input, output* output, data* data) {
          switch (data->state) {
            $LOOP$[case ->$[literal]: {
                     $COPY_SRCL$[int8 x; ]
                     break;
                   } case
          } switch
        } processBlock (function)


      }
    }
```

We start out with a reduction rule (since we want to replace the block with the result of the transformation). On the right side we use an **`inline tempalte with context`**. The context is the **`Dummy`** module; inside it we have the section which we mark as the template fragment (**`<TF .. TF>`**), so only that section replaces the block when the transformation executes.

The name property of the section uses a property macro to change the template name of the section (**`blockSection`**) to the name of the block from which this section is generated (using **`node.name`** as the expression in the macro). Inside the section we generate the contents discussed above.

The **`enum`** uses a property macro to change its name to a name that is derived from the block name. We call a behavior on the block that creates that name (**`node.stateEnumName()`**). Inside the enum we create a literal for each state by simply **`LOOP`**ing over the states of the input block.

The struct for the input data also uses a property macro to create a suitably unique name. Inside the struct we loop over all the **in** ports using an expression that calls into another behavior method:

```
public sequence<node<Port>> inports() {
  this.ports().where({~it => it.dir.is(<in>); });
}
```

The member in the struct needs to be templatized as well: a property macro changes the name, and the we also use a **COPY_SRC** macro to replace the dummy type **int8** with a copy of the type specified for the particular port in the input model. We use the same approach for the **out** ports.

The struct that represents the data of a block instance contains a member typed to the states **enum** generated before. Note how we do *not* need any kind of macro to make this work, we simply refer to the **states** enum – the reference is automatically resovled correctly because we are under the same parent (the section) as the enum when it is generated. We then generate members for the properties and variables using the same appraoch as for the ports discussed above.

We can now look at the generation of the function that implements the port behavior. Obviously we use a property macro to adapt the function's name to the input block. Then we define a signature that takes the three generated structs as arguments. Once again we don't have to do anything about them in terms of macros, since the references are resolved correctly automatically. Inside the function we **switch** over the various states and we add a case for each state that has an actual calculation associated with it. Here is the code for the **LOOP** macro:

```
node.states.where({~it => it.impl.isInstanceOf(CalcMethodBSI); });
```

Now, in the **case** statement we have to refer to the enum literal that represents the state addressed in that particular **case**. We do so by attaching a reference macro to the reference to the enum literal – this macro "bends" the reference target to the enum literal that corresponds to the current state. To do so, the macro simply returns the name of that literal (**node.name**). That name is resolved and bound to the actual literal lazily.

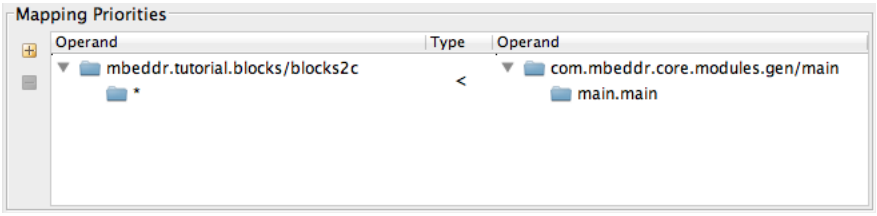
Inside the **case** statement we have a dummy statement (**int8 x;**) and a **break**. We replace the dummy statement with a list of all the statements in the body of the **CalcMethodBSI** using a **COPY_SRCL** (note the **L** for list!):

```
node.impl : CalcMethodBSI.body.statements;
```

There is one last thing we have to set up in order to make the generator work (and of course we have to implement the generators for the inside of blocks; see next subsection): We have to make sure our generator runs *before* the generator that creates text from the C tree to which we transform our blocks.

In the properties of our generator we specify a dependency on the **modules.gen** generator (that is the one that ultimately creates text from C trees):

On the **Priorities** tab we specify that our generator runs **strictly before** the **modules.gen** generator:



6.4.3 The Inside of Blocks

The inside of blocks is essentially arbitrary C code. However, it must be possible to refer to properties, ports and variables. All of these references are expressions, and all work essentially the same way. We'll illustrate it for ports.

■ **Structure** We define a **PortRef** concept that extends **Expression**. It has one reference to a **Port**:

```
concept PortRef extends Expression

  references:
    Port port 1
```

Concepts like this one (with *exaclty one* reference) are called *smart references*. They behave specially in the editor. If you press **Ctrl-Space** at a place where such a concept is allowed (in expressin context in this case), the code completion menu shows *all valid targets* for the reference instead of allowing users to explicitly instantiate the concept.

Once a target is selected (a **Port** in this case), the concept is instantiated and the reference is set.

■ **Editor** The editor for the **PortRef** is trivial; we simply show the name of the referenced port: **(%port%->name)**. You may have noticed that the port references use different colors depending on the direction of the port. This is implemented as a style attribute:

```
text-foreground-color : (node, editorContext)->Color {
  node.dir.is(<out>) ? new Color(4, 138, 22) : new Color(4, 29, 138);
}
```

■ **Scopes and Constraints** We have to ensure that only those ports are valid reference targets that live in the same block as the port reference. To do this we simply ascend the tree until we find a **Block** and then ask it for its block:

```
concept constraints PortRef {

  link {port}
    referent set handler:<none>
    scope:
      (... , enclosingNode, ...)->join(ISearchScope | sequence<node<Port>>) {
         enclosingNode.ancestor<concept = Block>.ports();
      }
    validator: <default>
    presentation : <no presentation>

}
```

Note that this only work if the **PortRef** is acually used under a **Block**! Note that a **PortRef** is an expression, so it can be used whereever an **Expression** is expected. We have to add a constraint to limit where it can be used. Since this is also true for references to variables and properties we introduce an interface **IBlockExpression** that is implemented by the various reference expressions. That interface has a **can be child** constraint that allows instances to exist only if they end up *somewhere under* a **Block**:

```
concept constraints IBlockExpression {
  can be child
    (node, parentNode, link, childConcept, scope, operationContext)->boolean {
      parentNode.ancestor<concept = Block>.isNotNull;
    }
}
```

■ **Type System** There are two type system concerns we have to deal with. One is the type of the port reference expression itself. It is simply the type of the referenced port (similarly for the variable and property references):

```
rule typeof_PortRef {
  applicable for concept = PortRef as pr
  do {
    typeof(pr) :==: typeof(pr.port);
  }
```

```
}
```

We also have to make sure that only **out** ports can be assigned to (**in** ports can only be read). Expressions that can be used on the left side of an assignment are called *lvalue*. Whether an expression is an lvalue may depend not just on the actual concept (a **LocalVariableReference** may be an lvalue but a **NumberLiteral** is never an lvalue), but it may also depend on other characteristics. In our case a **PortRef** is an lvalue if the port is an **out** port. To implement this we simply override the **isLValue** behavior method:

```
concept behavior PortRef {
  public boolean isLValue() overrides Expression.isLValue {
    this.port.dir.is(< out >);
  }
}
```

■ **Generator** Based on our understanding of the generator for blocks in general it should be clear what we have to do in the generator: a **PortRef** must be reduced to a reference to the respective member of the input or output struct passed to the implementation function. Here is the respective generator:

```
concept     PortRef
inheritors  false
condition   <always>
--> content node:
    module Dummy imports nothing {
      struct S {
        int8 x;
      };
      void dummy(S* s) {
        <TF [ ->$[s]->->$[x] ] TF>;
      } dummy (function)
    }
```

On the right side of the reduction rule we once again use an inline template with context. We only want to generate a reference to a member of a **struct** instance passed as an argument, but in order to be able to write down that reference in the template we fqirst have to have a **struct** and an argument of that struct in the template: we add a dummy module, inside the module we put a struct **S** with a member **x** and a function that takes a pointer to an **S** as its argument. Inside the function we can now refer to a member of the **s** argument by writing **s->x**. By marking only **s->x** as the template fragment (make sure you do not include the semicolon in the template fragment), the template will only generate that member access when it reduces the **PortRef**.

We need two reference macros to make it work. The first one is attached to the reference to the argument **s**; it distinguishes between the input and output structs (represented

as two different arguments named **input** and **output**). The reference macro expression just returns the respective names:

```
node.port.dir.is(<in>) ? "input" : "output";
```

The second reference macro is attached to the member reference. It needs to be bent to the member that has the same name as the port: **node.port.name**.

So when/where do we actually call the reduction rules for the **PortRef**? Remember the big template for the overall block discussed earlier? Inside the generated processing function we use a **COPY_SRCL** macro to copy over all the statements in the body of the block. However, **COPY_SRC(L)** macros don't just *copy* the contents. They also apply reduction rules where applicable. So if we use a **PortRef** anywhere inside the body statements, that **COPY_SRCL** macro invokes that reduction rule.

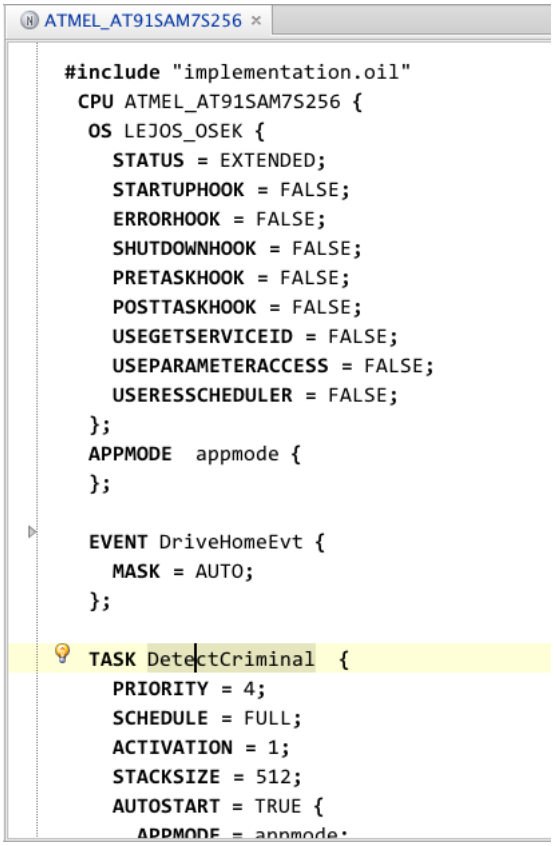
6.5 OS Configuration

Note: The code for this example can be found in the tutorial in the **mbeddr.tutorial.osconfig.*** languages.

In this section we discuss the development of external DSLs (i.e. new languages that are not directly related to C). In the chapter we also discuss the integration of such a language with mbeddr C after the fact. As an example we use operating system configuration, something that is necessary in many scenarios, among them the Lego Mindstorms/OSEK case study discussed from the website:

http://mbeddr.wordpress.com/further-reading/

In this case study we have built an implementation of the OIL language used for configuring the OSEK operating system. Here is an screenshot with an example OIL file. It configures all kinds of aspects of the OS, including tasks, events, various flags and the threadng model.



6.5.1 A DSL for OS Configuration

In this example we build a simplified version of such a language. For now, the operating system configuration DSL essentially defines a set of tasks:

```
OS Config
-----------------------------------

task mainTask prio = 1
task eventHandler prio = 2
task emergencyHandler prio = 3
```

■ **A New Language** We create a new language **`mbeddr.tutorial.osconfig`** using the *New* → *Language* menu item on the project. Note that this language does *not* extend any other language. It is completely standaline, a completely separate DSL developed with MPS – no relationship to mbeddr.

Unfortunately because of the way MPS handles cross-model generation, you do have to create a dependency to a language so you can implement an interface (see next paragraph

below) that makes this language play well with the transformation schedule. So please create an extends dependency on the **com.mbeddr.base** language.

■ **Structure** In the language we create a new concept **OSConfig**. This is the concept that contains a complete OS configuration and should be useable as a root node inside models. This is why we set its **instance can be root** property to **true**. We also implement **IGeneratesCodeForIDE**; this is the interface that makes this concept play well with mbeddr's cross-model code generation (mentioned in the previous paragraph):

```
concept OSConfig extends BaseConcept
                 implements IGeneratesCodeForIDE
  instance can be root: true
```

We also create a new concept interface **IOSConfigContents**. We make it extend from **INamedConcept** to give each instance a name:

```
interface concept IOSConfigContents extends INamedConcept
```

We can now go back to the **OSConfig** and add child collection **contents** of type **IOSConfigContents**:

```
concept OSConfig extends BaseConcept
  instance can be root: true
  children
    IOSConfigContents contents 0..n
```

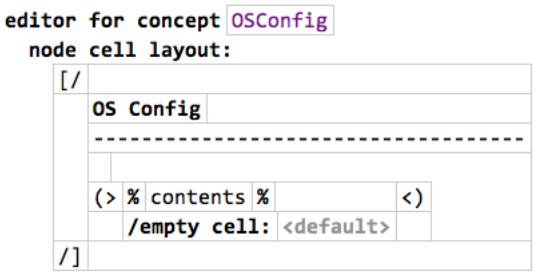
Let us now create a concrete concept that represents some kind of configuration content: the task definition:

```
concept TaskDef extends BaseConcept
                implements IOSConfigContents
  properties:
    prio : integer
```

■ **Editors** The interface gets no editor, of course. For the task definition we simply define an indent collection with the **task** keyword, the name and the priority:

```
editor for concept TaskDef
  node cell layout:
    [- task { name } prio = { prio } -]
```

Finally, the editor for the **OSConfig** itself we define a vertical list that holds the title, a line, a blank line (a constant with a space in it) and the vertical list of the **contents** child collection.



■ **Text Generator** So far all languages we have developed were extensions of a base language. The generator then was a transformation back to the base language. In case of our operating system configuration there is no base language – we have to generate text directly.

Text generators are different from the generators we have seen so far; those are actually model-to-model transformations since they map one MPS tree onto another one. For text generators we really just write into a text buffer. The language to do this is relatively simple. However, since we build text generators only for low-level base languages, this language really is good enough.

So let's get started with the text generator for **OSConfig**. Open the structure editor for the **OSConfig** concept and press the little **+** at the bottom left of the editor window and add a new Textgen conponent. Here is the code for it:

```
text gen component for concept OSConfig {
  extension : (node)->string {
    "osconfig";
  }
  encoding : utf-8

  (node, context, buffer)->void {
    append {OSConfig} \n ;
    append \n ;
    foreach c in node.contents {
      append ${c} \n ;
    }
  }
}
```

In this texgen component we specify the encoding, the file extension as well as the contents of the file. The code involving the **append** statements should be clear. Notice that we did *not* specify the name of the file (only its extension!). However, our **OSConfig** doesn't have a name. So this won't work! We have to give a name to **OSConfig**.

To do so, go back to the structure of **OSConfig** and make the concept implement **INamedConcept**. However, we don't really want to manually enter the name for each

of these things; instead, the name of the **OSConfig** should automatically be the name of the model in which it resides. You can achieve this by implementing a getter for the name in the constraints section:

```
concept constraints OSConfig {

  property {name}
    get:(node, scope)->string {
      node.model.name;
    }
    set:<default>
    is valid:<default>
}
```

We also have to implement the textgen for the **TaskDef**. It is really trivial:

```
text gen component for concept TaskDef {
  (node, context, buffer)->void {
    append {task} ${node.name} { } {() ${node.prio + ""} {}} ;
  }
}
```

You can now regenerate your model and, next to the C code, you should also get a **.osconfig** file that contains the operating system configuration.

6.5.2 Connecting to C

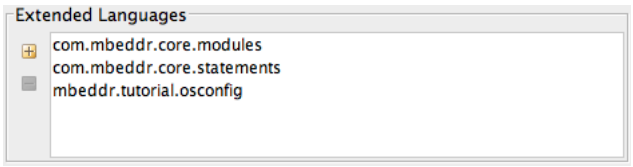
The language developed so far is completely independent of mbeddr C (even though for technical reasons it depends on **com.mbeddr.base** for the **IGeneratesCodeToIDE** interface). However, we may still want to refer to concepts defined in that language from C code.

In our example, we may want to provide a task implementation in C that corresponds to the task definitions in the OS configuration. In classical C, you may have to define a function that has the same name as the task and use a specific signature. Maybe some kind of special modifier is necessary. While we may generate such code (see below), we want to provide a better programming experience to the end user. In the code below you can see the new keyword **task**; it represent a task implementation. The name behind it (**nameTask**) isn't just a name. Instead it is a reference to a task definition in an OS config file. Pressing **Ctrl-Space** will code-complete to all available task definitions.

```
module Tasks imports nothing {

  void aHelperFunction() { }

  task mainTask {
    // here is some code that implements the task
```

```
    int8 aVariable = 10;
    aHelperFunction();
  }
}
```

■ **Another New Language** We create a new language **mbeddr.tutorial.osconfig.cimpl**. It acts as the adapter between mbeddr C and the OS configuration. To this end it extends these three languages:



■ **Structure** We create a new concept **TaskImpl** that implements **IModuleContent** so it can be used inside modules. It has a single reference to a **TaskDef** as well as a **StatementList** for the body:

```
concept TaskImpl extends BaseConcept
                 implements IModuleContent
  children:
    StatementList body  1

  references:
    TaskDef       task  1

  concept properties:
    alias = task
```

Note that we specify an alias for the concept even though it is technically a smart reference (it has exactly one reference). However, for this guy we do *not* want to trigger the special editor behavior. Instead we want the user to first create a **TaskImpl** node by typing **task** in a module and then select the reference to the **TaskDef**.

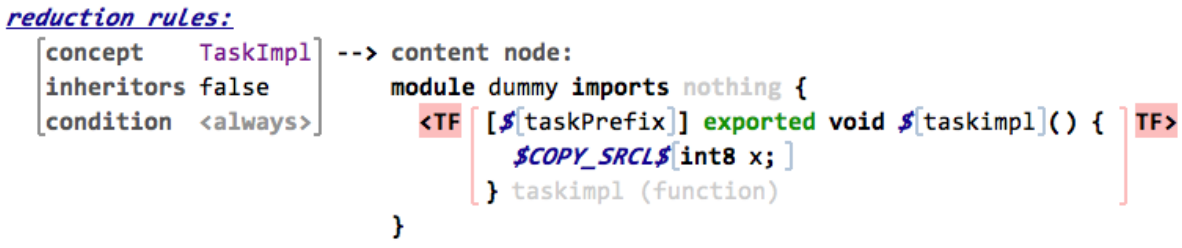
Normally we would have to define a scope for the **task** reference. However, the default behavior (which puts all the nodes in the current model into the scope) is exactly what we want so we don't have to do anything.

■ **Generator** Let us assume that the compiler for the OS we use in this example (some kind of fictional OSEK) expects task implementation function to use a special syntax: the example **mainTask** must be translated as follows:

```
task(mainTask) void Tasks_taskimpl_mainTask(void) {
  // here is some code that implements the task
  int8_t aVariable = 10;
```

```
  Tasks_aHelperFunction();
}
```

To create this function we define a generator with a reduction rule that looks as follows:



The generator uses a property macro to adapt the name, as well as a **COPY_SRCL** macro to copy in all the statements of the body. We also add a special modifier to the function. Modifiers can be added via an intention (**Alt-Enter**). The **[...]** modifier accepts arbitrary text between the brackets and then just outputs that text during text generation. We use the **taskPrefix** dummy text in the template and then replace it with the following macro expression:

```
"task(" + node.task.name + ")";
```

6.5.3 Memory Layout

Let us extend the OS config DSL with a way to define memory layouts. Here is some example code:

```
OS Config
-----------------------------------

task mainTask prio = 1
task eventHandler prio = 2
task emergencyHandler prio = 3
memory layout {
  region ram: 0..1024
  region eprom: endOf(ram)..2048
  region devices: endOf(eprom)..startOf(devices) + sizeOf(ram) * 2
}
```

Note how we have added additional contents to the os config file and also reuse C expressions within the regions – extended with new expressions to refer to the start, end and size of other regions.

■ **Yet another Language** We create another language to demonstrate how an external DSL can be extended using specific concepts from C without modifying the original OS configuration DSL. We create a new language **mbeddr.tutorial.osconfig.memory**. That new language extends **mbeddr.tutorial.osconfig** as well as **com.mbeddr.core.expressions.**

■ **Structure** The **MemoryLayout** implements **IOSContents** so it can be plugged into the **contents** collection of an **OSConfig**. It then contains a collection of **Region**s. Each region has a name (via the usual interface) and an **Expression** for the start and the end of the region. Since our language extends the C expression language we can use **Expression** here:

```
concept Region extends BaseConcept
               implements INamedConcept

  children:
    Expression startsAt 1
    Expression endsAt   1

  concept properties:
    alias = region
```

The **startOf(..)**, **endOf(..)** and **sizeOf(..)** expressions are new expressions contributed by this language. All three are essentially similar: they extend **Expression** and have a single reference to a **Region**. They also have an alias to prevent smart-reference editor behavior in the editor:

```
concept EndOfExpr extends Expression
                  implements <none>

  references:
    Region region 1 specializes: <none>

  concept properties:
    alias = endOf
```

■ **Editor** The editors for all of these concepts are straight forward based on what we have defined in the tutorials so far. A few remarks. The first one concerns the fact that when you create a new **memory layout**, between the two curlies it doesn't say **<no regions>** as would be MPS' default. We have achieved this by using an empty constant as the **empty cell** for the **regions** collection in the editor of **MemoryLayout**. Also, we have set the **editable: true** style attribute.

Second, we have used an indent child collection (**(- .. -)**) for the regions, but still get indentation and each region on a new line. We do that by setting the following style attributes on the collection cell **(-** itself:

```
indent-layout-on-new-line : true
indent-layout-indent : true
```

```
indent-layout-new-line-children : true
```

Finally, you may have noticed that there are no spaces on either side of the `..` in the region, or between the paranthesis and the region in the three new expressions. We do that by setting the `puctuation-left: true` and `puctuation-right: true` style attributes at the respective places. Marking something as "punctuation" removes the space on that side. Note that you should ever only set these attributes on constants – otherwise strange things can happen (MPS Bug!).

■ **Type System** The types of the `startsAt` and `endsAt` properties for regions must be defined. We can do this with the rule below. In both cases we express that the type of the respective property must be `uint64` or any of its shorter subtypes.

```
rule typeof_Region {
  applicable for concept = Region as r

  do {
    infer typeof(r.startsAt) :<=: new node<UnsignedInt64tType>();
    infer typeof(r.endsAt) :<=: new node<UnsignedInt64tType>();
  }
}
```

The type for the `EndOfExpr` and the `StartOfExpr` is simple as well; it is simply the type of the respective expression of the target region:

```
typeof(endOfExpr) :==: typeof(endOfExpr.region.startsAt);
```

For the `SizeOfExpr` it is a bit more interesting, since the type has to be the common supertype of the start and the end (we essentially build the difference, hence both types play a role). We can express this with the following approach:

```
rule typeof_SizeOfExpr {
  applicable for concept = SizeOfExpr as soe
  overrides false

  do {
    infer typeof(soe) :>=: typeof(soe.region.startsAt);
    infer typeof(soe) :>=: typeof(soe.region.endsAt);
  }
}
```

Notice how we define *two* typing rules for the same node (`soe`). In both we express a same-or-more-general-type relationship (`:>=:`) to the respective property. The only way how the type system engine can make *both* of these rules true is by assigning the common supertype of the two properties to `typeof(soe)` – which is exactly what we wanted.

■ **Scopes and Constraints** First of all we have to make sure that the new expressions (**startOf(..)**, **endOf(..)** and **sizeOf(..)**) can only be used below a **MemoryLayout**. Since this is the same for all three of them we have introduced a common superconcept. The constraint is defined for that one:

```
concept constraints RegionRefExpr {
  can be child
    (node, parentNode, link, childConcept, scope, operationContext)->boolean {
      parentNode.ancestor<concept = MemoryLayout>.isNotNull;
    }
}
```

This superconcept has also gotten the reference to the target **Region** so we can define the scope for that reference generically as well:

```
link {region}
  scope:
    (... , enclosingNode, ...)->join(ISearchScope | sequence<node<Region>>) {
      enclosingNode.ancestor<concept = MemoryLayout>.regions;
    }
```

■ **Generator** We will not define a generator for this extension. The point of this tutorial was to demonstrate the integration of various languages.

6.6 Complex Numbers

Note: The code for this example can be found in the tutorial in the **mbeddr.tutorial.complex** language.

In this tutorial we add complex numbers to C. We emphasize type system and editor convenience in this tutorial. We also explain how to build type system tests. We do not add a generator for complex numbers; in other words, you cannot execute programs that contain complex numbers. Here is some example code:

```
module Dummy imports nothing {
  void someFunction() {
    complex c1 = (10, 20i);
    complex sum = c1 + c1;
    complex product1 = c1 * c1;
    complex product2 = c1 * 10;
  }
}
```

■ **A new Language** We create a new language **mbeddr.tutorial.complex** that extends the **core.expressions** language.

■ **Structure** As can be seen from the code above we have to create a new type (**complex**) as well as a new literal (**(10, 20i)**). The type is trivial. The only important thing is that it extends the **Type** concept defined by the mbeddr:

```
concept ComplexType extends Type
  concept properties:
    alias = complex
```

The literal for complex types extends **Literal**, which in turn extends **Expression**. It has two children which are in turn literals:

```
concept ComplexLiteral extends Literal
    NumberLiteral real 1
    NumberLiteral img  1
```

■ **Editor** We don't have to define an editor for the **ComplexType** because the superconcept **Type** already has an editor that shows the alias of the type by default. The editor for the literal is also straight forward based on what we have discussed in the tutorial so far:

```
editor for concept ComplexLiteral
  node cell layout:
    [- ( % real % , FE% img % i ) -]
```

However, there are interesting things to say about *entering* complex literals. You enter it by first entering the real part (10 in the above example). You then press comma. This transforms the 10 into a complex literal, and you can now enter the imaginary part (20 in the example above). Here is how this is done.

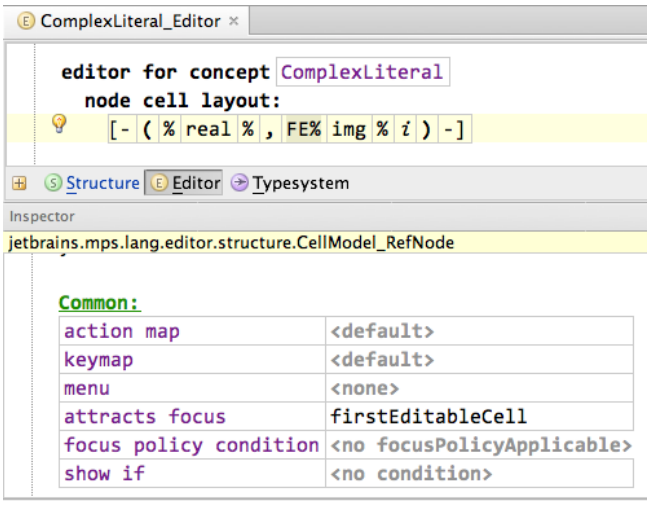
First, we specify the **dontSubstituteByDefault** property for the concept. This means that when you press **Ctrl-Space** in expression context, **ComplexLiteral** in *not* offered. This means you cannot enter it. We now have to build the press-comma-to-make-it-a-complex functionality. This is done via a right transformation (you press a key on the right of something and then run a transformation). Here is the code:

```
side transform actions makeComplexLiteral

right transformed node: NumberLiteral
  condition :
    (operationContext, scope, model, sourceNode)->boolean {
      MeetTypeHelper.isInstanceOf(sourceNode.type : IType, concept/INumeric/);
    }

  actions :
    add custom items  (output concept: ComplexLiteral)
```

```
simple item
  matching text
    ,
  description text
    <default>
  icon
    <default>
  type
    <default>
  do transform
    (operationContext, scope, model, sourceNode, pattern)->node<> {
      node<ComplexLiteral> cl = new node<ComplexLiteral>();
      sourceNode.replace with(cl);
      cl.real = sourceNode;
      cl.img;
    }
```

Here is what this means. If you have a **NumberLiteral** (that fits the condition with the **MeetType**, explained below), then we have a transformation that creates a **ComplexLiteral** if we press comma on the right side. It does so by running the **do transform** script. The script should be obvious; it creates a new node of type **ComplexLiteral**, replaces the **sourceNode** with that new node, sets the **real** part of the complex number to be the **sourceNode** (remember this is a **NumberLiteral**) and then returns the **img** part.

The idea with returning the **img** part is that the input focus is then set to that node. However, that really doesn't work (MPS Bug!). To compensate for that we set a **firstEditableCell** focus policy in the editor cell for the **img** part. This is what shows up as **FE** in the editor above:



A word on the strange condition used in the action above:

```
condition :
  (operationContext, scope, model, sourceNode)->boolean {
    MeetTypeHelper.isInstanceOf(sourceNode.type : IType, concept/INumeric/);
  }
```

Essentially what we want to do is restrict the comma-transformation to work only if the type of the expression on which we right-transform implements **INumeric**. Normally w would write this as **sourceNode.type.isInstanceOf(INumneric)**, where **.type** is a built-in operator the returns the typesystem's type for the respective node. However, the type for number literals is a bit of a challenge. If you press **Ctrl-Shift-T** on a number literal such as **10**, you get something more complex: a meet type. A meet type expresses the type system's uncertainty about the type of a node. For example, a meet type **(int8 | uint8)** means that the type could be any of the element types! This is because it a number literal such as **10** can actually can actually be a signed or an unsigned number. If we call **isInstanceOf** on a **MeetType**, this will of course fail! What we really want to know is: is any of the element types inside a meet type an instance of **INumeric** (in the case above). The **MeetTypeHelper.isInstanceOf** method performs just that function.

■ **Type System** Of course the most interesting aspect of complex numbers is the type system. The existing operators (such as **+** or ∗) have to be adapted to work with complex numbers (adding new operators that use the same symbols makes no sense, since then those would be offered in the code completion menu alongside the old ones, confusing the user).

Before we adapt the binary operators, we have to make sure we define a type for the **ComplexType** and the **ComplexLiteral**. **ComplexType** already has a type, since there is a typesystem rule for **Type** that just clones the type node:

```
rule typeof_Type {
  applicable for concept = Type as t
  overrides false

  do {
    typeof(t) :==: t.copy;
  }
}
```

This way, if you ask a type for its type (sounds strange, I know) you get a clone of the type itself. For complex literal we return a new instance of **ComplexType**:

```
rule typeof_ComplexLiteral {
  applicable for concept = ComplexLiteral as cl
  overrides false

  do {
    typeof(cl) :==: new node<ComplexType>();
  }
}
```

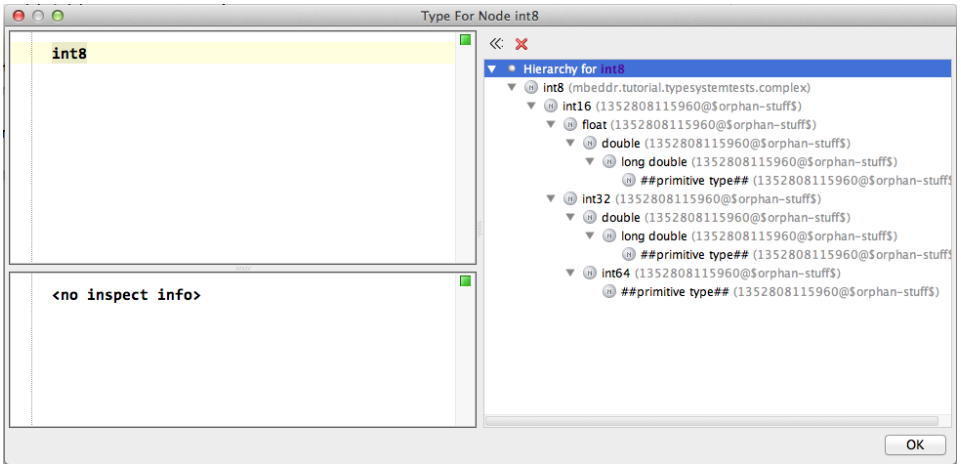
The types of the binary operators are defined differently, they use so-called overloaded operations containers. Let us look at the definition of one of these guys for dealing binary arithmetic operators in case of two complex numbers:

```
operation concepts: BinaryArithmeticExpression
  left operand type: new node<ComplexType>() is exact: true
  right operand type: new node<ComplexType>() is exact: true
is applicable:
  <no isApplicable>
operation type:
  (op, leftOperandType, rightOperandType)->node<> {
    new node<ComplexType>();
  }
```

This one defined that for any instance of **BinaryArithmeticExpression** (or subconcepts) if we use a **ComplexType** as the type of the left argumetn and a **ComplexType** as the type of the right argument, the type of the expression itself is also a **ComplexType**. We also have to look at the case where one of the two arguments is complex, and the other one isn't. The interesting question is how we represent "the other one isn't": what type does it have to be? Here is the code:

```
operation concepts: BinaryArithmeticExpression
  left operand type: new node<ComplexType>() is exact: false use strong subtyping false
  right operand type: new node<PrimitiveType>() is exact: false use strong subtyping false
is applicable:
  (op, leftOperandType, rightOperandType)->boolean {
    MeetTypeHelper.isInstanceOf(rightOperandType : IType, concept/INumeric/);
  }
operation type:
  (op, leftOperandType, rightOperandType)->node<> {
    new node<ComplexType>();
  }
```

This is the case for when there's a complex type on the left and "the other type" on the right (there's another one of these rules for the vice versa case). The "other type" has to be a **PrimitiveType** that implements (taking into account the **MeetType** stuff discussed above) **INumeric**. To understand this, take a look at the type hierarchy for **int8** in the following illustration:



As you can see, at the top of the hierarchy is the primitive type. In mbeddr C, this acts as the top type in the numeric type hierachy (note that this is the type hierarchy as defined by subtying rules and *not* the inheritance hierarchy of the respective concepts!). To make sure we actually just get numerical types and not (possible) non-numeric primtive types we add the "safety net" applicability condition.

6.6.1 Implementing a Type System Test

Since we don't have a generator, we cannot execute the code with the complex numbers. But we can write a type system test. A type syste test is a test, to be executed in MPS or on the command line, that verifies that expected type system errors are actually reported, and that no other, unexpected type system errors show up. Here is an example test case:

This root is an instance of **NodesTestCase** that is available to your model if you use the **jetbrains.mps.lang.test** language. In its **nodes** slot we create a **test node** (just press **Ctrl-Space**) and inside of that one we create an implementation module (you have to do it via the module name **ImplementationModule**, or **ImMo** using camel-case selection). Inside the implementation module you can write normal C code, including the complex numbers if you include the respective language into your model.

The green annotations are interesting. The first one, **check types**, is a test operations annotation that tells the test engine to run type system checks on this node. The second one, **node .. has error**, is used to mark a node where you expect an error to show up. Since the node has a red squiggly, and that squiggly makes sense, you mark the error as expected.

There are two more things worth mentioning: the test info node and the build configuration. The test info node is a node that points to the project that contains the test. This is so that tests can work on the command line via **ant**. When specifying this path you may want to use MPS' path variables to avoid hard-coding directories. In the example code we do use a path variable. Second, even though you're not generating executable C code, the model that contains the test cases has to contain a **BuildConfiguration**. It may be empty, but it has to exist.

■ **Running the Test Case** You can now run the test case by selecting the **Run Tests in <your model name>** option from the context menu of the model that contains the test (yes, it takes way too long. JetBrains are working on this problem!). To illustrate a test failure, I have changed one line to:

```
int8 productOfComplexAndComplex = c * c;
```

Running the tests results in a failed test, with this exception:

```
java.lang.RuntimeException: java.lang.reflect.InvocationTargetException
  at jetbrains.mps...virtual_perform_1215601182156(TypesCheckOperation_Behavior.java:21)
  at jetbrains.mps.lang.test.behavior.TypesCheckOperation_BehaviorDescriptor.
    virtual_perform_1215601182156(TypesCheckOperation_BehaviorDescriptor.java:20)
    ...
Caused by: java.lang.reflect.InvocationTargetException
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  ... 40 more
Caused by: junit.framework.AssertionFailedError: type complex is not a subtype of int8
  at junit.framework.Assert.fail(Assert.java:50)
  at junit.framework.Assert.assertTrue(Assert.java:20)
  at jetbrains.mps.lang.test.runtime.SubtreeChecker.checkNodeForErrors(SubtreeChecker.java
    :91)
  at jetbrains.mps.lang.test.runtime.SubtreeChecker.checkNodeForErrors(SubtreeChecker.java
    :39)
  ... 45 more
Process finished with exit code 0
```

While this is not really a very readable error message, it does say something about **type complex is not a subtype of int8**. The problem is that it does not report which node is the offending one. You have to figure that out for yourself. Fortunately is usually relatively simple:

- If something has a red squiggly in the test but does not have a **node .. has error** annotation, this is a problem.
- If something has *no* red squiggly in the test but *does* have a **node .. has error** annotation, this is a problem, too.

6.7 Vectors and Matrices

Note: The code for this example can be found in the tutorial in the **mbeddr.tutorial.vectors** language.

Vectors and Matrices are quite useful in many embedded or technical applications. As a consequence of their unique syntax, the projectional nature of the MPS editor is also

a nice fit. In this tutorial we look at building "strange" editors and some type system stuff. As with complex numbers, we don't have a generator. Here is some example code that uses vectors and matrices:

```
void testVectorsAndMatrices() {

  /* Vectors can be multiplied with regular numbers  */
  vector<int16, 3> aIntVectorOfSize3 = [ 1 ; 22 ; 3 + 4 ] * 512;

  /* A new operator has been introduced for the cross product  */
  vector<int16, 3> resultOfCrossProduct = aIntVectorOfSize3 x aIntVectorOfSize3;

  /* Matrices use a "table" of literals  */
  matrix<int16, 2x3> aMatrix = [ 1 + 2 | 7 + 2 | 3 ; 3 | 51 | 9 ];

  matrix<int16, 3x1> matrixWithOnlyOneCol;

  /* A matrix with only one column is a vector of the same size with a compatible base type  */
  aIntVectorOfSize3 = matrixWithOnlyOneCol;

  /* The projectional editor can also be exploited for the Transpose operator  */
  matrix<int16, 1x3> transposedMatrix = matrixWithOnlyOneCol T;
} testVectorsAndMatrices (function)
```

6.7.1 Types and Literals

■ **Structure** First of all, it is important to realize that vectors are structurally just matrices with one column. In terms of types and the permitted operators, matrices and vectors must be treated differently, but structurally they are similar. Consequently there is a **MatrixLiteral** (but no vector literal). A **MatrixLiteral** owns a collection of **MatrixLiteralCol** concepts, which in turn contain **Expression**s:

```
concept MatrixLiteral extends Literal
  children:
    MatrixLiteralCol cols 1..n specializes: <none>
  concept properties:
    alias = [
    shortDescription = matrix
```

```
concept MatrixLiteralCol extends BaseConcept
  children:
    Expression elements 1..n specializes: <none>
```

In terms of types there is a **MatrixType** and a **VectorType** which both implement the **IMatrixType** interface:

```
interface concept IMatrixType extends <none>
  properties:
    dimensionsRows : integer
  children:
    IType baseType 1 specializes: <none>
```

This interface has the number rows as well as the base type. These are important because the number of rows as well as the base type are relevant for typing: a **vector<int8,3>** is not compatible with a **vector<int16,2>**. Since the type system really just works with the type nodes, the type has to contain this information (this is in contrast to complex numbers, where every complex number is compatible with any other complex number). So then the type for vectors is defined like this:

```
concept VectorType extends Type
                   implements IMatrixType
  concept properties:
    alias = vector
```

The type for matrices adds another dimension to represent the number of columns:

```
concept MatrixType extends Type
                   implements IMatrixType
  properties:
    dimensionsCols : integer
  concept properties:
    alias = matrix
```

■ **Type System** Since we had been discussing the structure of types above let's look at the type system for the types and literals. Let us look at the typing rules for the **VectorType**. First of all, since it extends **Type**, it is automatically typed to a clome of itself:

```
rule typeof_Type {
  applicable for concept = Type as t
  overrides false

  do {
    typeof(t) :==: t.copy;
  }
}
```

Now we have to take a look at subtyping. Vectors are *covariant* regarding their base type: **vector<T,i>** is a subtype of **vector<Q,i>** if **T** is a subtype of **Q**. To make this work we have to implement a subtyping rule for **VectorType**. Subtyping rules return the collection of supertypes for any particular type. Here is the basic structure for **VectorType**:

```
subtyping rule supertypesOfVectorType {
  weak = false
  applicable for concept = VectorType as vectorType

  supertypes {
    nlist<IMatrixType> vectorSuperTypes = new nlist<IMatrixType>;
    // more stuff to be added here
    return vectorSuperTypes;
  }
}
```

To implement the covariance discussed above, we can write the following code, which should be easy to understand (**immediateSupertypes** is a built-in type system operator):

```
foreach baseSuperType in immediateSupertypes(vectorType.baseType) {
  node<VectorType> st = new node<VectorType>();
  st.baseType = baseSuperType : Type;
  st.dimensionsRows = vectorType.dimensionsRows; // same number of rows!
  vectorSuperTypes.add(st);
}
```

There is also a special case

```
if (vectorSupertypes.isEmpty && vectorType.baseType != null) {
  node<MatrixType> mt = new node<MatrixType>();
  vectorSupertypes.add(mt);
  node<VectorType> vt = new node<VectorType>();
  vectorSupertypes.add(vt);
}
```

Essentially the same supertyping rule has to be defined for **MatrixType**s. However, we want to make sure that if the matrix type has only one column, then a corresponding **VectorType** is among the supertypes:

```
if (matrixType.dimensionsCols == 1) {
  node<VectorType> vt = new node<VectorType>();
  vt.dimensionsRows = matrixType.dimensionsRows;
  vt.baseType = matrixType.baseType.copy;
  supertypes.add(vt);
  return supertypes;
} else {
  foreach superType in immediateSupertypes(matrixType.baseType) {
    // essentially same as before
  }
  if (supertypes.isEmpty && matrixType.baseType != null) {
    node<MatrixType> st = new node<MatrixType>();
    supertypes.add(st);
  }
  return supertypes;
}
```

Let us now take a look at the **MatrixLiteral**. Consider this matrix: $\begin{bmatrix} 10 & 12.1 \\ 1000 & 1000.12 \end{bmatrix}$ What is the type of that matrix? Sure, it has two columns and two rows, but what is the base type? It must be the most general of the element types (in this case a **float**, I think). Here is the code that calculates that most general type:

```
rule typeof_MatrixLiteral {
  applicable for concept = MatrixLiteral as ml
  overrides false

  do {
    var commonElementType;
    foreach e in ml.cols.elements {
      infer typeof(e) :<=: commonElementType;
    }

    node<MatrixType> mt = new node<MatrixType>();
    mt.baseType = commonElementType.copy : IType;
    mt.dimensionsCols = ml.cols.size;
    mt.dimensionsRows = ml.cols.first.elements.size;
    typeof(ml) :==: mt;
  }
}
```

The second part is simple: we create a new **MatrixType**, set its properties, and then assign this new type as the type of the matrix literal **ml**. The first part of the rule is more interesting. We first declare a new, unbound type system variable using the **var** keyword. We then iterate over all elements in all columns and write the following relationship for each of them:

```
infer typeof(e) :<=: commonElementType;
```

This expresses that the type of the element **e** must be the same or a subtype of (**:<=:**) as the **commonElementType** variable. Note how we create *one such equation for each element*. The only way how the type system engine can make *all* of these equations true (which is what the type system engine tries to do!) is to make **commonElementType** represent the common supertype of all of the element types. This neatly shows the power of the declarative, sovler-based type system engine!

There is one last thing we have to make sure: the following matrix is invalid: $\begin{bmatrix} a & b & c \\ d & e & \\ f & & \end{bmatrix}$ In other words, the number of elements in each column must be the same. Here is the non-typesystem rule that makes sure this is the case:

```
checking rule check_MatrixLiteral {
  applicable for concept = MatrixLiteral as ml
  overrides false

  do {
    node<MatrixLiteralCol> firstCol = ml.cols.first;
```

```
    if (ml.cols.any({~it => it.elements.size != firstCol.elements.size; })) {
      error "all columns must have the same size" -> ml;
    }
  }
}
```

By the way: to render a type (or any other element) nicely in textual output (such as error messages), you should override the **getPresentation** behavior. This is essentially MPS' **toString()** equivalent.

```
concept behavior VectorType {

  public string getPresentation()
        overrides BaseConcept.getPresentation {
    "vector<" + this.baseType.getPresentation() + ", " + this.dimensionsRows + ">";
  }
}
```

■ **Editor** The editors for the two types is are trivial and not worth discussing. What is interesting is the editor for the **MatrixLiteral**:

```
editor for concept MatrixLiteral
  node cell layout:
    [> $ custom cell $ ^(> % cols % /empty cell: <default> <) $ custom cell $ <]
```

It consists of two custom cells (about which we'll talk in a moment) and a horizontal collection (**(>..<)**) of the colums. A column has its own editor, discussed below. To make sure the columns have some space between them we use a trick. We add a white, vertical bar as the list separator:

```
list element:
separator              |
separator constraint   noflow
separator style        <no parentClass> {
                         text-foreground-color : white
                         padding-left : 1 spaces
                         padding-right : 1 spaces
                       }
```

The editor for the **MatrixLiteralCol** is simply a vertical list of all the expressions. We specify two style properties to make sure each expression is horizontally centered, and that the whole vertical list is centered with regards to the line in which the matrix lives:

```
horizontal-align: center
default-baseline: collection center
```

Let us now look at the two custom cells. These obviously handle the "big brackets" on the left and right side of the matrix. When you use the **custom cell** cell type, you have to actually return the cell object from the inspector using an expression like **new OpeningBracketCell(node)**. The real drawing of the cell happens in the class. Here is the basic outline of a custom cell implementation:

```
public class OpeningBracketCell extends AbstractCellProvider {

  private node<> myNode;

  public OpeningBracketCell(node<> node) {
    this.myNode = node;
  }

  public EditorCell createEditorCell(EditorContext context) {
    EditorCell_Basic result = new EditorCell_Basic(context, this.myNode) {
      ...
    };
    return result;
  }
}
```

Essentially you implement a subtype of **AbstractCellProvider** and implement the **createEditorCell** method to return a suitable cell. In many cases it is good enough to return an anonymous subclass of **EditorCell_Basic** that overrides a few methods accordingly. Here is how it works for the opening bracket cell:

```
EditorCell_Basic result = new EditorCell_Basic(context, this.myNode) {
  public void paintContent(Graphics g, ParentSettings parentSettings) {
    g.setColor(Color.BLACK);
    EditorCell_Collection parent = this.getParent();
    int x = getX();
    int y = parent.getY();
    int height = parent.getHeight();
    g.fillRect(x, y, 2, height);
    g.fillRect(x, y, 4, 2);
    g.fillRect(x, y + height - 2, 4, 2);
  }

  public void relayoutImpl() {
    this.myWidth = 4;
    this.myHeight = 10;
  }

  public boolean isFirstCaretPosition() {
    return true;
  }
};
```

There are a few things that can be done to improve the editing experience for the user; for example, the following intention lets the user select **Add New Column** from the intentions menu (**Alt-Enter**) to add a new column.

```
intention addNewMatrixCol for concept MatrixLiteralCol {
  error intention : false
```

```
  available in child nodes : true
  child filter : <all child nodes>

  description(editorContext, node)->string {
    "Add New Column";
  }

  execute(editorContext, node)->void {
    node<MatrixLiteralCol> currentCol = node.ancestor<concept = MatrixLiteralCol, +>;
    node<MatrixLiteralCol> newCol = new node<MatrixLiteralCol>();
    currentCol.elements.forEach({~it => newCol.elements.add new(<default>); });
    currentCol.add next-sibling(newCol);
    editorContext.select(newCol.elements.first);
  }
}
```

A few words about it. This intention is available not just on a **MatrixLiteralCol** but also on all of its children in the tree. This means that when it is invoked, the **node** variable may not actually be a **MatrixLiteralCol**, but some kind of **Expression** instead. The first line in the **execute** body compensates for that by retrieving the current parent column. We then create a new column, add a new (empty) expression to it for each expression in the current column and then add the **newCol** as a next sibling to the current one. We then set the focus into the first element of the new column.

6.7.2 Overriding the Existing Operators

The existing operators must get new type system rules (similar to the complex numbers). In the example we have done it only for the **MultiExpression**. Here is the code for the case where a matrix is multiplied with a skalar value:

```
operation concepts: MultiExpression
  left operand type: new node<MatrixType>()
        is exact: false use strong subtyping false
  right operand type: new node<PrimitiveType>()
        is exact: false use strong subtyping false
is applicable:
  (op, leftOperandType, rightOperandType)->boolean {
    MeetTypeHelper.isInstanceOf(rightOperandType : IType, concept/INumeric/);
  }
operation type:
  (op, leftOperandType, rightOperandType)->node<> {

    // determine the least common supertype between the base type
    // of the  matrix and the primitive on the other
    // side of the binary operator
    set<node<>> nodes = new hashset<node<>>;
    nodes.add(leftOperandType : IMatrixType.baseType);
    nodes.add(rightOperandType);
    set<node<>> leastCommonSupertypes =
         typechecker.getSubtypingManager().leastCommonSupertypes(nodes, false);

    // create a matrix type or a vector type
```

```
    // depending on what's on the left side
    node<IMatrixType> resultType =
         leftOperandType : IMatrixType.cloneForBaseType(leastCommonSupertypes.first : IType
    );
    return resultType;
  }
```

As before, we use the **MeetTypeHelper** to find out whether the right side is actually a numeric primitive type. Inside the body we use the **typechecker** to determine the least common supertype between the base type of the matrix and the primitive on the other side of the binary operator since this will become the basetype of the result type. We then create a **MatrixType** or a **VectorType** depending on whether the original non-primitive type was a matrix or a vector; this is handled correctly by **cloneForBaseType** defined for **IMatrixType**. Here's the implementation for a **MatrixType**:

```
public node<IMatrixType> cloneForBaseType(node<IType> baseType)
     overrides IMatrixType.cloneForBaseType {
  node<MatrixType> res = new node<MatrixType>();
  res.dimensionsRows = this.dimensionsRows;
  res.dimensionsCols = this.dimensionsCols;
  res.baseType = baseType;
  res;
}
```

Similar rules have to written for the other cases (i.e. matrix/vector and matrix/vector or primitive and matrix/vector). Also, the code has to be generalized for other operators, not just multiplication. This is left as an exercise for the reader.

6.7.3 Adding new Operators

We also want to add additional operators (cross product **x** and transposition **T**).

■ **Structure** The **CrossProductExpression** extends **BinaryArithmeticExpression**; this way it plugs in with all kinds of existing facilities regarding editor support and typing. Here is the definition:

```
concept CrossProductExpression extends BinaryArithmeticExpression
  concept properties:
    alias = x
    priolevel = 2000
    shortDescription = cross-product
```

The alias is used as the editor by default. The **prioLevel** determines the operator precedence. 2000 is the same as the regular multiplication. **shortDescription** is what is shown in the code completion menu behind the actual **x** symbol.

The **MatrixTransposeExpr** is a unary expression, it has an alias of **T**, a **prioLevel** of 4000 and a **shortDescription** of **transpose**. It has the **dontSubstituteByDefault** flag to prevent entering it directly.

```
concept MatrixTransposeExpr extends UnaryArithmeticExpression
  concept properties:
    alias = T
    priolevel = 4000
    shortDescription = transpose
    dontSubstituteByDefault
```

■ **Editor** The editor for the cross product is trivial and inherited from the **BinaryArithmeticExpression**. For the **MatrixTransposeExpr**, the editor is shown below:

```
editor for concept MatrixTransposeExpr
  node cell layout:
    [> % expression % T <]
```

It is essentially a horizontal list with the **expression** and the **T** next to each other. To get the superscript, you have to set the **cell layout** property of the collection cell to **superscript** and you have to add the **script-kind: superscript** style property to the **T**.

To be able to enter the **T** on the right side of an expression you need to write a right transformation. Here is the code:

```
side transform actions makeTransposeExpr

right transformed node: Expression
  condition :
    (operationContext, scope, model, sourceNode)->boolean {
      sourceNode.type.isInstanceOf(MatrixType);
  }

  actions :
    add custom items  (output concept: MatrixTransposeExpr)
      simple item
        matching text
          T
        description text
          <default>
        icon
          <default>
        type
          <default>
        do transform
          (operationContext, scope, model, sourceNode, pattern)->node<> {
            node<MatrixTransposeExpr> n = new initialized node<MatrixTransposeExpr>();
            sourceNode.replace with(n);
            n.expression = sourceNode;
```

```
            PrioUtil.shuffleUnaryExpression(n);
            n;
          }
```

This is a normal right transform as discussed earlier; what is interesting is the call to **`PrioUtil.shuffleUnaryExpression(n)`**. This reshuffles the tree to take care of precedence as expressed in the **`prioLevel`** property. It has to be called as part of any left or right transformation that involves expressions.

■ **Type System** The cross product is a binary operator, so it needs an overloaded operations container that computes the resulting type:

```
operation concepts: CrossProductExpression
  left operand type: new node<VectorType>()
        is exact: false use strong subtyping false
  right operand type: new node<VectorType>()
        is exact: false use strong subtyping false is applicable:
  (op, leftOperandType, rightOperandType)->boolean {
    leftOperandType : VectorType.dimensionsRows ==
          rightOperandType : IMatrixType.dimensionsRows;
  }
operation type:
  (op, leftOperandType, rightOperandType)->node<> {
    set<node<>> nodes = new hashset<node<>>;
    nodes.add(rightOperandType : IMatrixType.baseType);
    nodes.add(leftOperandType : IMatrixType.baseType);
    set<node<>> leastCommonSupertypes =
         typechecker.getSubtypingManager().leastCommonSupertypes(nodes, false);
    node<VectorType> vt = new node<VectorType>();
    vt.baseType = leastCommonSupertypes.first : IType;
    vt.dimensionsRows = rightOperandType : IMatrixType.dimensionsRows;
    return vt;
  }
```

Finally, the transpose expression needs to calculate the resulting type, which is the same type as the expression on which it is called, but with the dimensios exchanged:

```
rule typeof_MatrixTransposeExpr {
  applicable for concept = MatrixTransposeExpr as mte
  overrides true

  do {
    when concrete (typeof(mte.expression) : MatrixType as mteType) {
      node<MatrixType> mt = new node<MatrixType>();
      mt.baseType = mteType.baseType.copy;
      mt.dimensionsCols = mteType.dimensionsRows;
      mt.dimensionsRows = mteType.dimensionsCols;
      typeof(mte) :==: mt;
    }
  }
}
```

Note how we use **`when concrete`** to express that this typing rule may only be executed once the type of **`mte.expression`** has successfully been calculated.

6.8 Registers

Note: The code for this example can be found in the tutorial in the **mbeddr.tutorial.registers** language.

Some processors have special-purpose registers: when a value is written to such a register, a hardware-implemented computation is automatically triggered based on the value supplied by the programmer. The result is then stored in the register. If we want to run code that works with these registers on the PC for testing, we face two problems: first, the header files that define the addresses of the registers are not valid for the PC's processor. Second, there are no special-purpose registers on the PC, so no automatic computations would be triggered.

6.8.1 Example

We can solve this problem with a language extension that allows us to define registers first class and access them from C code (see code below). The extension also supports specifying an expression that performs the computation.

```
exported register8 ADC10CTL0 (val * 1000)

void calculateAndStore(int8 value) {
   int8 res = // some calculation with value
   ADC10CTL0 = res; // actually stores result * 1000
}
```

The extension also supports 16-bit registers. These can be accessed as a word, or by each byte separately.

```
exported register16 ADC10_16 (val)
  low byte suffix = _L (val)
  high byte suffix = _H (val)

void calculateAndStore(int8 lowByteVal, int8 highByteVal) {
  ADC10_16.L = lowByteVal;
  ADC10_16.H = highByteVal;
}
```

When the code is translated for the real device, the real registers are accessed using the processor header files. In testing we use generated **struct**s to hold the register data and insert the expression into the code that updates the struct, simulating the hardware-based computation. A configuration item in the build configuration is used to decide the two cases:

```
Build Configuration for mbeddr.tutorial.main.newLanguages

Build System:
  ...

Configuration Items
  ...
  registers emulated

Binaries
  ...
```

6.8.2 Implementation

The basic setup for the register definition is simple: **Register**s are **IModuleContent** that have a name an the value expression. We provide a new expression **val** that is restricted via a constraint to be used only in register definitions. Type checks make sure that the expression is a **int8** or **int16**, respectively. We also define expressions to access registers from regular C code (as in **ADC10CTL0 = res;**). However, there are a few more interesting things that are worth discussing.

■ **References to IModuleContent** To reference registers from C code, there is a **RegisterRefExpr** that extends **Expression**. It references a register:

```
concept RegisterRefExpr extends Expression
                        implements IModuleContentRef
  references:
    Register register 1
```

Note the implementation of the **IModuleContentRef**. This is necessary to support mbeddr's cross-model code generation. Every reference that points to an **IModuleContent** has to implement this interface. The interface requires the implementation of two behavior methods. Both deal with references that point to module contents that potentially reside in a different model:

```
public void rebindToProxy(node<> proxyElement)
  overrides IModuleContentRef.rebindToProxy {
  this.register = proxyElement : Register;
}

public node<> referencedModuleContent()
  overrides IModuleContentRef.referencedModuleContent {
  this.register;
}
```

■ **.L and .H suffixes** When accessing a 16-bit register, it is possible to access each byte separately (see the 16-bit example above). Access to the low byte happens via the

`.L` suffix, access to the high byte happens via `.H`. Users can simply press `.H` or `.L` on the right side of a 16-bit register reference.

Structurally, the `LowByteRefExpr` and the `HighByteRefExpr` are `UnaryExpression`s that wrap the `RegisterRefExpr` discussed earlier. So if the user presses `.H` or `.L` on the right side of a `RegisterRefExpr`, a right transformation has to be triggered that wraps in in the respective unary expressions:

```
right transformed node: RegisterRefExpr
  condition :
    (operationContext, scope, model, sourceNode)->boolean {
      sourceNode.register.isInstanceOf(Register16) &&
          sourceNode.register : Register16.allowCharAccess;
    }

  actions :
    add custom items  (output concept: LowByteRefExpr)
      simple item
        matching text
          .L
        do transform
          (operationContext, scope, model, sourceNode, pattern)->node<> {
             node<LowByteRefExpr> lbe = new node<LowByteRefExpr>();
             sourceNode.replace with(lbe);
             lbe.expression = sourceNode;
          }
```

The transformation intercepts `.L` on the right side, creates a new `LowByteRefExpr`, adds the original node as its `expression` and then replace the source node with the new `LowByteRefExpr`.

■ **Multiple Generators** As metioned above, the registers and the register access has to be translated differently depending on whether we generate for the real system or for a unit test running on a PC. Here is the fundamental approach:

- We create a single generator (in terms of MPS) that contain two mapping configurations, one called `registersSimulated` and another one called `registersRealWorld`.
- The respective transformation rules are called from either one of them.
- We define a configuration item that allows users to specify in the build configuration which way the registers should be translated.
- Each of the mapping configurations has an applicability condition that queries this configuration item and then activates/deactivates the respective mapping configuration.

Let us look at the configuration item first. Here is the code:

```
concept RegisterConfigurationItem extends BaseConcept
                                  implements IConfigurationItem
  properties:
    trafo : RegisterTransformationKind
```

The concept implements **IConfigurationItem** so it can be used in the **configuration items** slot of the build configuration. It also has a property of type **RegisterTransformationKind** which is an enumeration that has two literals:

```
enumeration datatype RegisterTransformationKind
  ...
  value emulate     presentation emulated     (default)
  value realworld   presentation realworld
```

Here is the respective query in the **registersRealWorld** mapping configuration:

```
mapping configuration registersRealWorld
top-priority group false

is applicable:
  (genContext)->boolean {
    node<IConfigurationItem> rc = BCHelper.expectBCConfigItem(
      genContext.inputModel,
      genContext,
      "com.itemis.smartmeter.cextension/main.registersRealWorld",
      concept/RegisterConfigurationItem/
    );
    if ( rc != null ) {
      return rc : RegisterConfigurationItem.trafo.is(< realworld >);
    } else {
      return false;
    }
  }
```

We call a helper method **BCHelper.expectBCConfigItem** to find a configuration item of type **RegisterConfigurationItem**. We pass in the input model, the generation context, as well as a string that described which generator is trying to find the configuration item (this is important for error reporting). We then check whether the **trafo** property has the **realworld** enum value. A similar query (using **simulated**, obviously) is used in the applicability condition for the **registersSimulated** mapping configuration.

Note that this is an example of where a generator is designed to be able to cope with both options. In case there is an existing generator and you want to override it *without changing the existing generator*, then a different appraoch is required: the overriding generator has to have a *strictly before* $<$ priority relative to the existing generator to make sure it "reduces away" things before the existing generator has a chance to reduce things.

■ **Generating Header File Access** For the real-world translation, we assume that

the register names are defined as **#define**s in an included header file. So the code generator can generate references to **#define**. In this example, however, we use a differnet appraoch: we generate text directly. While this can be considered cheating, it is often useful, for example, when generating calls to implicit functions (i.e. functions that are available in C even though there is no prototype in a header the declares them. **printf** is an example.). Here are the reduction rules:

```
reduction rules:
  concept    RegisterRefExpr                                          --> <T [| $[registerName] |] T>
  inheritors false
  condition  (node, genContext, operationContext)->boolean {
               !(node.parent.isInstanceOf(HalfRegRefExpr));
             }

  concept    HighByteRefExpr  --> <T [| $[registerName] |] T>
  inheritors false
  condition  <always>

  concept    LowByteRefExpr   --> <T [| $[registerName] |] T>
  inheritors false
  condition  <always>

  concept    Register  --> <T  T>
  inheritors true
  condition  <always>
```

The register declarations themselves are reduced to an **EmptyModuleContent**, i.e. nothing. The **HighByteRefExpr** and the **LowByteRefExpr** are reduced to an instance of **ArbitraryTextExpression**: this thing can mix generated text with nodes (see next paragraph). Finally, the **RegisterRefExpr** is also reduced to a "piece of text", but only if it is not owned by a **HalfRegRefExpr**, the superconcept of **LowByteRefExpr** and **HighByteRefExpr**.

Let us take a closer look at the **ArbitraryTextExpression**. It is essentially a horizontal collection of parts; these are either strings or nodes. In the example we generate a string **registerName** whose value we then change via a property macro (as usual). The **ArbitraryTextExpression** provides two additional goodies, both accessible via the inspector. First, you can specify a type for it, so the type system in the IDE (or the generator) is satisfied. Second, you can specify a to-be-included header file. This header file is then included in the generated header file. This is useful for **malloc**, for example, where you just generate the text **malloc** and then use **stdlib.h** as the to-be-inclded header file.

6.9 Metadata

Note: The code for this example can be found in the tutorial in the **mbeddr.tutorial.metadata** language.

6.9.1 Example

Imagine you want to enforce architectural constraints of the kind: only modules in the Driver Layer are allowed to write certain members of a **struct**, and the Application Layer is allowed to read them. You may also want to specify a maximum read frequency.

Here is how this could look in mbeddr. We first define a few roles:

```
Access Specification
--------------------------------
role DriverLayer
role ApplicationLayer
```

We can then specify permissions on **struct** members:

```
exported struct EngineStatus {
  int8 speed; [ApplicationLayer: read
               DriverLayer: write]
  int8 rpm; [ApplicationLayer: write
             DriverLayer: read]
};
```

We can also specify which role a module plays, and if it assigns to a member for which it has only **read** permissions, an error is reported. Here is an example of a permission violation (the error message is **role ApplicationLayer cannot write this member** – somehow I cannot screenshot that):

```
[module role: ApplicationLayer]
module UsingModule imports DataDefinition {

  EngineStatus es;

  void dummy() {
    es.rpm = 10;
    es.speed = 100;
  } dummy (function)

}
```

Note how this language adds additional metadata and constraints to existing language concepts without changing these. Also, the example implements with language extensions a task that is typically handled by external tools and/or external meta data, resulting in much tighter integration.

6.9.2 Implementation

■ **Annotations** The implementation of this feature is mainly based on annotations and a non-typesystem rule. The annotations allow adding the permissions and the module role specification to struct members and modules, respectively. Let us look at the case for struct member. Here is the annotation concept:

```
concept AccessSpec extends NodeAttribute

  children:
    Permission permissions 0..n

  concept properties:
    role = accessSpec

  concept links:
    attributed = Member
```

Note how this concept extends **NodeAttribute**. This means it can be attached to instances of other nodes without declaring **AccessSpec**s as children of that other node's concept. This way the language that defines the target node does not have to be modified. The concept link **attributed** specifies the concept to which this thing can be attached, in this case it is **Member**s. We also specify the **role**. In our example, we specify that the **AccessSpec** child lives in the **@accessSpec** child slot under a **Member**.

The editor for **AccessSpec** embeds the attributed node (i.e. the **Member** it is attached to), and then, on the right side, renders the permissions.

```
editor for concept AccessSpec
  node cell layout:
    [- [> attributed node <] (> % permissions %          <) -]
                               /empty cell: <constant>
```

An intention is used to be able to attach an **AccessSpec** to a **Member**. The code should be obvious to understand:

```
intention addAccessSpec for concept Member {
  error intention : false
  available in child nodes : false

  description(editorContext, node)->string {
    "Add Access Spec";
  }

  isApplicable(editorContext, node)->boolean {
    node.@accessSpec == null;
  }

  execute(editorContext, node)->void {
    node.@accessSpec.set new(<default>);
  }
}
```

A similar annotation is created to mark modules with the role/layer in which they reside. Take a look at the code to see the details.

■ **The Check** To report the errors if a module writes a member for which i does not have the permission we have implemented a non-typesystem rule for **ImplementationModule**s. While the code is not completely trivial, it should be easy to understand. It simply reports an error if a struct member is assigned to, and the member only has a **read** permission for the role declared by the module.

```
checking rule check_AccessRestrictions {
  applicable for concept = ImplementationModule as immo
  overrides false

  do {
    if (immo.@moduleRole.isNotNull) {
      node<RoleSpec> moduleRole = immo.@moduleRole.role;
      nlist<AssignmentExpr> assignments = immo.descendants<concept = AssignmentExpr>;
      foreach ae in assignments {
        if (ae.left.isInstanceOf(AbstractDotExpression)) {
          node<Member> member = ae.left : AbstractDotExpression.member : MemberRef.member;
          if (member.@accessSpec != null) {
            node<Permission> permission = member.@accessSpec.permissions.
```

```
                findFirst({~it => it.role == moduleRole; });
            if (permission.activity.is(< read >)) {
              error "role " + moduleRole.name + " cannot write this member" -> ae.left;
            }
          }
        }
      }
    }
  }
}
```

Obviously, there should be additional checks, such as: if we assign to a member that has a permission specified from within a module that has *no* role specification, then we get an error: the user has to decide the role of the module so we can verify if the write access is allowed or not.

6.10 What if it doesn't work?

In this section we discuss various approaches of figuring out what's going on if you get stuck.

6.10.1 Finding other People's Usages

When you use mbeddr, there are already a whole lot of languages available for you to learn how to build things: the mbeddr C implementation, all the default extensions, the languages in this tutorial, as well as MPS' own BaseLanguage implementation. It is very likely that any problem you face in terms of language design and implementation has already been implemented in one of these examples.

So when thinking about a new language or a language extension, it is a good idea to first think about which existing language (or extension) is structurally similar. Once you've found such an analogue, you should write down an example using that analogue language and press **Ctrl-Shift-S** on the respective concept. This gets you to the definition of the respective concept (remember that the sources are always available. You can then look around and see how things are implemented.

If you don't know what a particular language implementation construct does you can press **Ctrl-Shift-S** on the *language implementation code*. This way you get to the definition of the concepts used for implementing languages. You can then use **Find Usages** to find all instances of that concept. This way you can navigate to all places

where JetBrains or the mbeddr team has used that particular concept. Based on this set of example uses of a particular language implementation concept it is often relatively easy to find out what the respective concept means or does.

6.10.2 println-Debugging

A pragmatic way of understanding scopes, type system rules or behaviors is to use println-debugging: you output program data onto the console. MPS defines two aliases (`sout` and `serr`) to write `System.out.println("")` and `System.err.println("")` statements in procedural code. Using those you can output nodes or concepts – their `toString` methods are reasonably useful to understand things. You may also want to call `getPresentation` on nodes to get a nicer rendering.

There is one more thing to know, however. By default, when you launch MPS, you don't see the console output because there is no console window anywhere. Here is what you should do to change this:

- On Windows, take a look at the `mps.bat` shell script in the MPS installation directory. By default, it uses `javaw`, i.e. the version of the Java runtime that does not show a console. Change that to `java`, and you will get one.
- On the Mac, don't start MPS by double-clicking on the application icon. Instead, use a console and navigate into the `MPS-x.y.app` directory. There you can invoke `./mps.sh`. Starting MPS this way will give you a console.

On that console you will get (typically way too many) log messages, errors and exception stack traces. It will also show your own `sout` and `serr` output.

6.10.3 Debugging MPS in MPS

It is possible to debug MPS in MPS. You can create an `MPS` debug configuration and then debug the execution of MPS on the level of the DSLs that are used for language definition. However, this is so slow (in terms of startup time and the performance of the to-be-debugged second instance of MPS) that we never use it.

6.10.4 Debugging Transformations

In MPS, having several chained transformations is normal, so MPS provides support for debugging the transformation process. This support includes two ingredients. The first one is showing the mapping partitioning, the second one traces all intermediate transformation steps.

■ **Mapping Configurations** For a given model, MPS computes the order in which transformations apply and reports it to the user. This is useful to understand which transformations are executed in which order and to debug transformation priorities. Let us investigate a simple example C program that contains a message definition and a **report** statement. The **report** statement is transformed to **printf** statements.

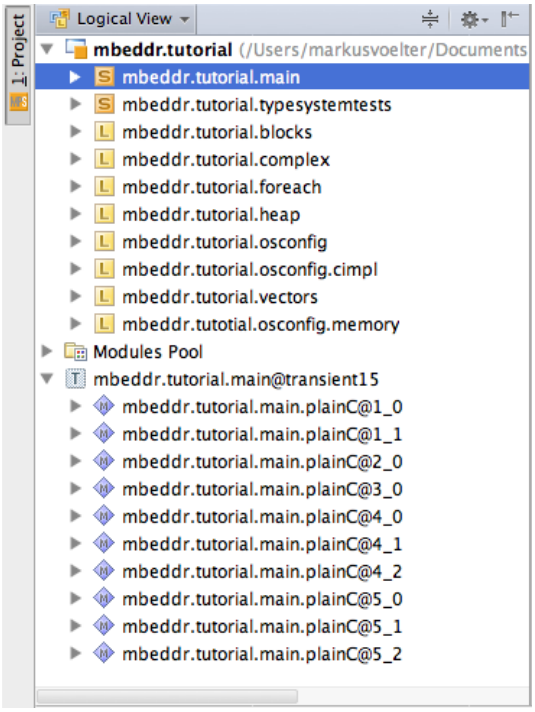
```
module Simple imports nothing {

  messagelist messages {
    INFO aMessage() active: something happened
  }

  exported int32 main(int32 argc, int8*[] argv) {
    report (0) messages.aMessage();
    return 0;
  }
}
```

Below is the mapping configuration for this program:

```
[ 1 ]
com.mbeddr.core.modules.gen.generator.template.main.copyInImportedModules

[ 2 ]
com.mbeddr.core.buildconfig.generator.template.main.dependencyGraph
com.mbeddr.core.modules.gen.generator.template.main.removeCommentedAndEmptyCode

[ 3 ]
com.mbeddr.core.util.generator.template.main.gswitchAndDectabb

[ 4 ]
com.mbeddr.core.buildconfig.generator.template.main.runConfigItemPreprocessors
com.mbeddr.core.make.generator.template.main.main
com.mbeddr.core.modules.generator.template.main.main
com.mbeddr.core.util.generator.template.main.blockAndLog
com.mbeddr.core.util.generator.template.main.flags
com.mbeddr.core.util.generator.template.main.namedStructInit
com.mbeddr.core.util.generator.template.main.rangeExpression
com.mbeddr.core.util.generator.template.main.rangeFor
com.mbeddr.core.util.generator.template.main.reportingDoNothing
com.mbeddr.core.util.generator.template.main.reportingPrintf
com.mbeddr.core.util.generator.template.main.withStatement

[ 5 ]
com.mbeddr.core.buildconfig.generator.template.main.desktop
com.mbeddr.core.buildconfig.generator.template.main.nothing
com.mbeddr.core.modules.gen.generator.template.main.main
com.mbeddr.mpsutil.graph.generator.template.main.main
```

This particular model is generated in five phases, each one executing a set of transformations whose priorities indicate that they can be executed in parallel without harm to each other.

■ **Inspecting intermediate Models** By default, MPS runs all generators until everything is either discarded or transformed into text. While intermediate models exist, they are not shown to the user. For debugging purposes though, these intermediate, transient models can be kept around for inspection. You can switch on **`Save Transient Models`** by clicking on the small, grey **`T`** at the right side of MPS' bottom status bar or by selecting the respective option in the **`Build`** menu. The transient (i.e. intermediate) models are then shown in the project tree on the left at the very bottom of the project explorer:

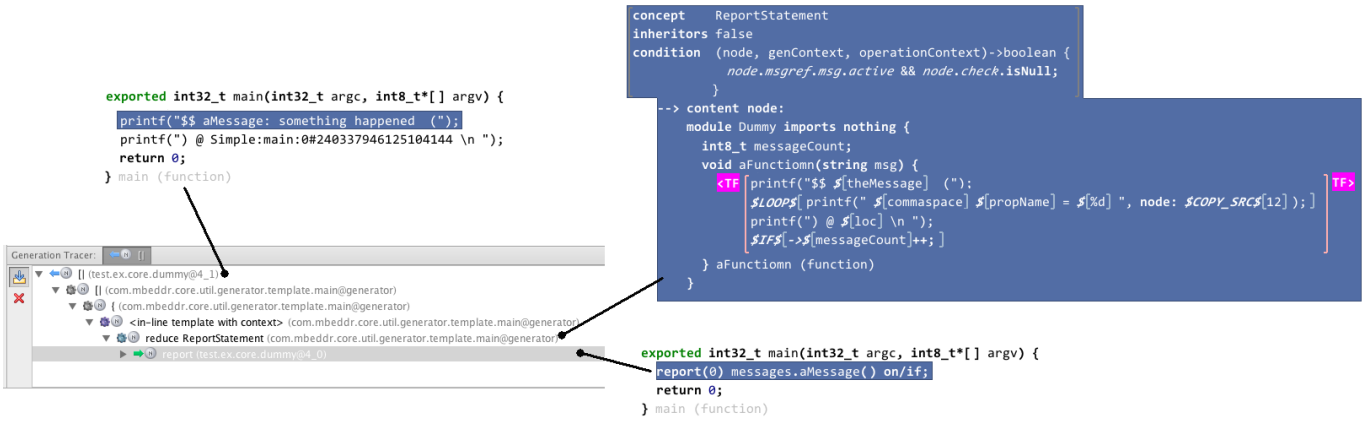


Each transformation phase (shown in the mapping configuration) is represented by one or more transient models. The suffix of the intermediate models corresponds to the phases. As an example, here is the program after the **`report`** statement has been transformed:

```
module Simple imports nothing {

  exported int32 main(int32 argc, int8*[] argv) {
    printf("$$ aMessage: something happened ");
    printf("@ Simple:main:0#240337946125104144 \n ");
    return 0;
```

```
  }
}
```

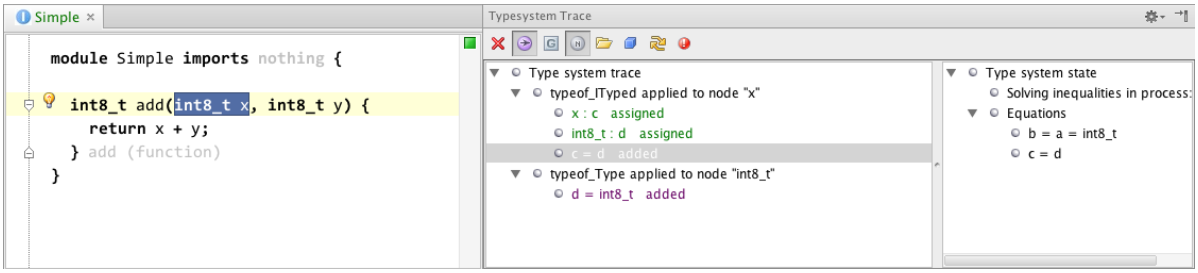
MPS also supports tracing an element through the intermediate models. Fig. ?? shows an example. Users can select a program element in the source, target or an intermediate model and trace it to the respective other ends of the transformation by using the **Show Transformation Traceback** item from the **Language Debug** submenu in the context menu of the to-be-traced element. The generation tracer also shows the transformation rules involved in the transformation.



6.10.5 Debugging Type Systems

MPS comes with two facilities. First, pressing **Ctrl-Shift-T** on any program element will open a dialog that shows the type of the element. If the element has a type system error, that dialog also lets the user navigate to the rule that reported the error.

The second facility is much more sophisticated. For any program node, MPS can show the so-called type system trace (the figure below shows a simple example). Remember how the MPS type system is relies on a solver to solve the type system equations associated with program elements (specified by the language developer for the respective concepts). So each program has an associated set of type system equations. Those contain explicitly specified types as well as type variables. The solver tries to find type values for these variables such that all type system equations become true. The type system trace essentially visualizes the state of the solver including the values it assigns to type variables as well as which type system rules are applied to which program element.



This example shows the solver state for the **`Argument x`**. It first applies the rule **`typeof_ITyped`** (**`Argument`** implements **`ITyped`**) which expresses that the type of the element (type variable **`c`** is the same as the element's **`type`** property (type variable **`d`**). It then applies the **`typeof_Type`** rule to the argument's type itself. This rule expresses that the type of a **`Type`** is a clone of itself. Consequently, the type variable **`d`** can be set to **`int8_t`**. In consequence this means that the type variable **`c`** (which represents the type of the **`Argument`**) is also **`int8_t`**.

The above example is rather trivial, type system traces can become quite involved. In most cases we rely on using println-debugging for type system problems by adding **`System.err.println`** statements at the respective places in the type system rules.

6.11 Language Evolution

If you define your own languages or extensions, it is likely that these change over time based on what you learn from using them. In this case you have to deal with existing programs: how do you keep them up-to-date with the changing language. In this section we discuss some strategies.

6.11.1 Backward Compatibility and Deprecation

Generally it is a good idea to make changes to languages in a backward-compatible way. This way, existing programs remain valid, while new language constructs become available. The following changes to languages are backward compatible:

- Changes in the concrete syntax (i.e. the projection rules) are always automatically backward-compatible, because, as a consequence of the projectional editor, the existing model structure is just projected differently. There's nothing to do in this case.

- Adding additional features to concepts. While the programs may report errors (in case the new feature is not optional), the program does remain structurally valid and can easily be migrated manually. Using Self-Migrating Code (see Section 6.11.2), defaults can even be supplied manually.

- Moving features to a super-concept (Pull Up refactoring) is not a problem for the code. Access to features is done via their name, and moving them up does not invalidate these references.

- Renaming features or concepts (via the respective refactorings) does break existing models. However, MPS writes a `.history` file that is used to "rewire" existing programs correctly. Make sure you check in the `.history` files, otherwise it won't work :-)

- Removing features is also backward compatible; however, the AST still contains the data for the removed feature *even though the feature declaration in the respective concept does contain that feature anymore*. MPS then shows a warning and suggests to remove that "unnecessary data" via a quick fix. MPS can also run *all* these quick fixes via the model checker.

What is *not* backward compatible is if you change the type of a reference or a child in a concept. In this case the data stored in the AST can't be interpreted anymore. Also, just simply deleting a concept while instances of that concept are still used in models is not backward compatible. These things should be avoided:

Instead of just deleting a concept you should deprecate it. In the context of mbeddr you can do this by implementing `IDeprecatedLangConcept` from the `com.mbeddr.core.base` language. A generic non-typesystem rule will warn for all uses of this interface in models.

In the former case (where you want to change the type of a feature) you should instead add a *new* feature with the new type (in terms of the editor, you can, for example, move either the new or the old one into the inspector). After a while, when all of these programs have been migrated, you can safely remove the old one.

6.11.2 Migrating Code

In order to migrate code orderly, the source (old) concept and target (new) concept must both be available. This is why we need to use deprecation in the first place, and not

just delete an old concept and create a new one. Once all instances of the old concept are gone, we can delete it. In this subsection we discuss how you can assist users to automate the migration.

Let us look at an example. Remember the **safeheap** statement:

```
concept SafeHeapBlock extends Statement
                      implements ILocalVarScopeProvider
  children:
    SafeHeapVar    vars 0..n
    StatementList  body 1
```

Let us consider changing the **SafeHeapBlock** in the following way: instead of storing the variables in a separate collection **vars**, we just want to put them as **LocalVariableDeclaration**s at the beginning of the implementation block. Since **SafeHeapVar** extends **LocalVariableDeclaration**, this is possible in terms of subtyping. We'll look at various ways of handling this migration in the remainder of this section.

■ **Intentions** An intention can be provided for the user to automatically execute a migration. The applicability condition of the intention would inspect the model to see if a migration is still required. If so, the **execute** section of the intention would perform the respective migration, by, for example, setting a default value for a new property, by moving the contents from one feature into another one or by completely replacing the source node with another one. Here is the intention for our example case:

```
intention migrateStructure for concept SafeHeapBlock {
  error intention : false
  available in child nodes : false

  description(editorContext, node)->string {
    "Migrate Structure";
  }

  isApplicable(editorContext, node)->boolean {
    node.vars.isNotEmpty;
  }

  execute(editorContext, node)->void {
    if (node.body.statements.isEmpty) {
      node.body.statements.addAll(node.vars);
    } else {
      node.vars.forEach({~it =>
        node.body.statements.first.add prev-sibling(it); });
    }
  }
}
```

We allow the intention only if there are still **SafeHeapVar**s in the **vars** collection. In the **execute** block we distinguish between the case where the body is empty (and we just add the variables) or where it is not empty (in which case we add them to the beginning using the **add prev-sibling** construct). Note that adding nodes to a new collection

(**body.statements**) removes them from the original one **vars**, because a node can only be owned by one container collection.

■ **Self-Migrating Code** In addition to intentions, MPS also supports quick fixes. The difference between the two is that a quick fix is always associated with a failing type system rule (inference rule or non-typesystem rule), whereas an intention is just associated with a concept. If the type system rule fails, the user is offered (in the **Alt-Enter** menu) to run the quick fix. In addition, quick fixes can also be executed automatically – leading to self-migrating code.

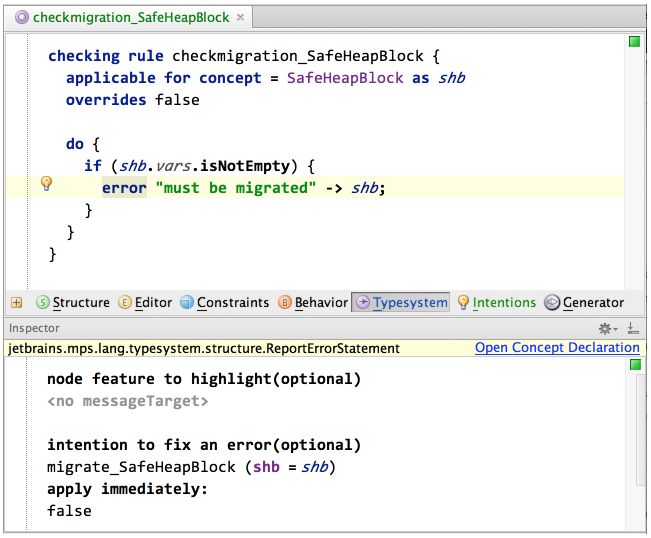
For our example, let us write a non-typesystem rule that detects that a migration is still necessary:

```
checking rule checkmigration_SafeHeapBlock {
  applicable for concept = SafeHeapBlock as shb
  overrides false

  do {
    if (shb.vars.isNotEmpty) {
      error "must be migrated" -> shb;
    }
  }
}
```

We can now implememt a quick fix that resolves the issue. The actual implementation of the quick fix is of course the same as the **excute** block in the intention shown above, but the structure of quick fixes is a bit different. Create a new quick fix (in the typesystem aspect):

```
quick fix migrate_SafeHeapBlock

arguments:
  node<SafeHeapBlock> shb

description(node)->string {
  "Migrate SafeHeapBlocks";
}

execute(node)->void {
  if (shb.body.statements.isEmpty) {
    shb.body.statements.addAll(shb.vars);
  } else {
    shb.vars.forEach({~it => shb.body.statements.first.add prev-sibling(it); });
  }
}
```

The key to make this work is to associate the quick fix with the **error** statement in the non-typesystem rule shown earlier. You do this by selecting the respective **error** statement, and then, in the inspector, selecting our new quick fix as the **intention to fix an error** (yes, bad naming...). We have to make sure we pass values for all the arguments declared for the quick fix:

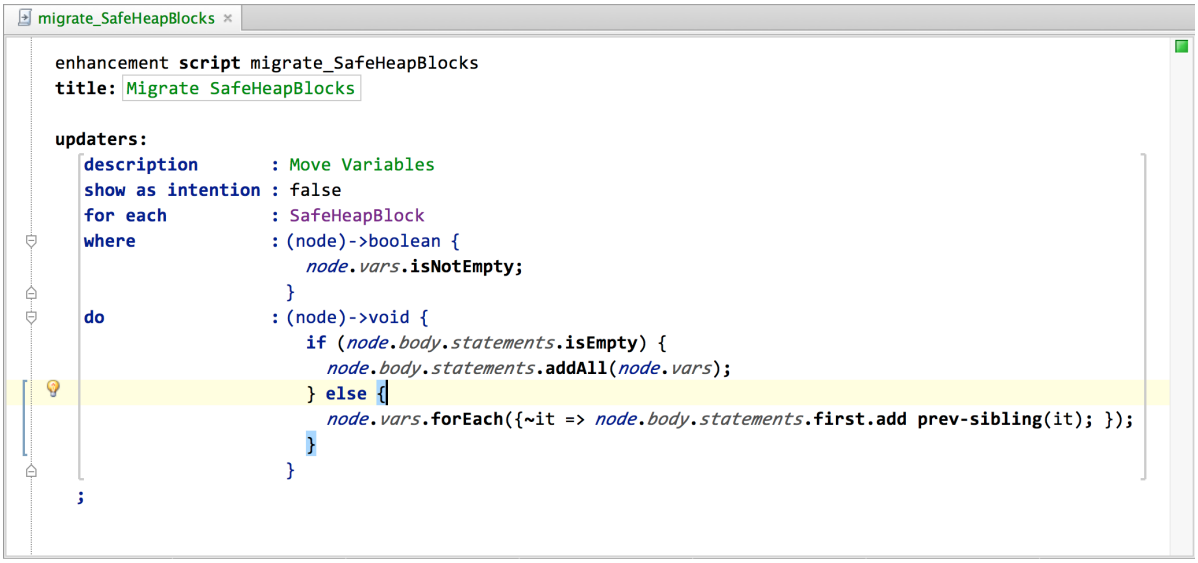


As it stands now, we have achieved essentially the same as the intention before: if the check fails, the quick fix is shown in the **Alt-Enter** menu (quick fixes can be recognized by the red light bulb symbol). However, we can make the quick fix execute automatically by setting the **apply immediately** flag at the point where we associate the quick fix with the **error** statement (see screenshot above). As soon as you open the root that contains a **SafeHeapBlock** that needs to be migrated, the migration is performed automatically.

It is a matter of taste, I guess, whether you want programs to change without users doing so explictly. I guess for cases where the user does not *see* the change (only internal restructurings are performed), and automatic quick fix is perfectly ok. If the quick fix performs a visible change, then maybe requiring the user to do it manually is a better idea. Based on that argument, the example we have shown here is a bad one.

■ **Migration Scripts** Both approaches shown so far address each instance of a to-be-migrated concept separately. In the case of intentions this is obvious; but also for the automatically executed quick fixes, these are only executed if the user opens the root that contains something that needs to to be migrated, or if the user runs the model checker and chooses to **Apply Quick Fixes** afterwards. Migration scripts are different, since they migrate a whole model/solution/project.

For our example, we create a new migration script. To be able to do so, we first add a **scripts** aspect to the **mbeddr.tutorial.heap** language. In that new aspect we add a **Language Migration Script**. Here is the code:



Once again, the actual implementation is the same as before. As you can see, one migration script can contain several updaters, and each updater has an applicable concept, a condition, as well as the code to be executed.

To execute such a script, select it from the **`Tools -> Scripts -> Enhancement Scripts -> <script name>`**.

Part III

Reference Documentation

7 mbeddr.core — C in MPS

7.1 mbeddr core: 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.

7.1.1 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.

7.1.2 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 (**Im- plementationModule**) with an exported function. You can toggle the *exported* flag with the intention **Toggle Export**. The second module (**ModuleUsingTheExportedFunction**) imports the **ImplementationModule** with the **im- ports** keyword in the module header. An importing module can access all exported contents defined in imported modules.

```
module ImplementationModule imports nothing {

  exported int32 add(int32 i, int32 j) {
    return i + j;
  }
}

module ModuleUsingTheExportedFunction imports ImplementationModule {

  int32 main(int8 argc, string[ ] args) {
    int32 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 7.6.

7.1.3 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 5.1. 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).

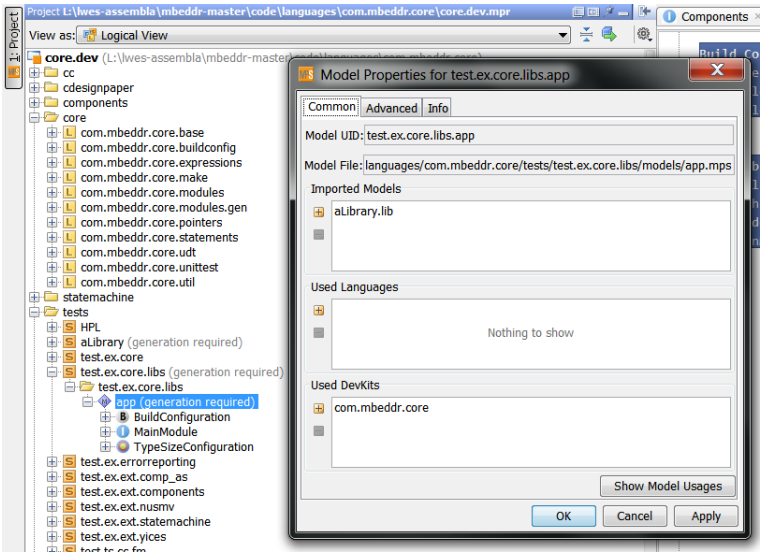
> **Note:** The build language is designed to be extended for integration with other build infrastructures. In that case, other targets (than **desktop**) would be provided by the language that provides integration with a particular build infrastructure.

■ **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 **`IConfiguration-Item`** 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 **`BuildConfiguration`**s.

7.1.4 Unit tests

Unit Tests are supported as first class citizens by mbeddr C. A **TestCase** implements **IModule- Content**, so it can be used in implementation modules alongside with functions, **struct**s 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 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:

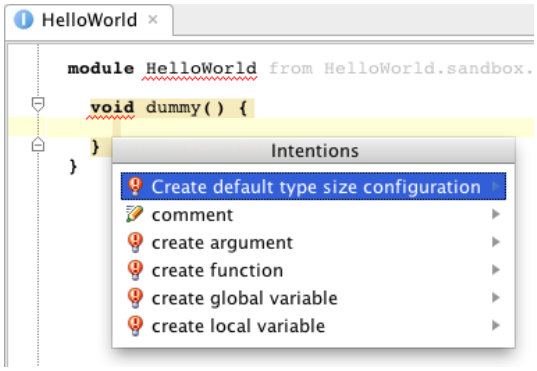
```
runningTest: running test @AddTest:test_testAddInt:0
runningTest: running test @AddTest:test_testAddFloat:0
```

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 x, int8 y, int8 z) {
  assert(0) x + y == z;
}
```

7.1.5 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**, **int16**). 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` → `char`
- `int16` → `short`
- `int32` → `int`
- `int64` → `long long`
- `uint8` → `unsigned char`
- `uint16` → `unsigned short`
- `uint32` → `unsigned int`
- `uint64` → `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.

7.1.6 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;
  }
}
```

7.1.7 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**, ..., **int64**) that can represent the number.

```
module LiteralsApp imports nothing {

  exported test case testLiterals {
    int32 intFromHex = hex<aff12>;
    assert(0) intFromHex == 720658;

    int32 intFromOct = oct<334477>;
    assert(1) intFromOct == 112959;

    int32 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.

7.1.8 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 x = 10;
    int32* xp = &x;
    assert(0) *xp == 10;

    int32[ ] anArray = {4, 5};
    int32* 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);
}
```

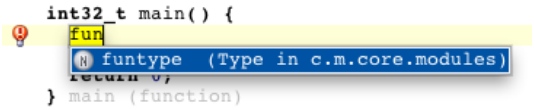
You have to include the external module **stdlib** in your implementation module with **imports stdlib**. You can then call **malloc** or **free**:

```
module BasicPointer imports stdlib {
  ...
  exported test case mallocTest {
    int8* mem = ((int8*) malloc(sizeof int8));
    *mem = 10;
    assert(0) *mem == 10;
    free(mem);
  }
}
```

■ **Function Pointers** In regular C, you define a function pointer type like this: **int (*pt2Fun) (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, int32)=>(int32)** 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 add(int32 a, int32 b) {
    return a + b;
  }

  int32 minus(int32 a, int32 b) {
    return a - b;
  }

  exported test case testFunctionPointer {
    // function pointer signature
    (int32, int32)=>(int32) pt2Function;

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

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

To initialize a function reference to "nothing", please use the **noop** expression.

Function types can be used like any other type. This is illustrated in the next example. The typedef **typedef (int3_t, int32)=>(int32) 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, int32)=>(int32) as ftype;

  int32 add(int32 a, int32 b) {
    return a + b;
  }

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

  int32 doOperation(ftype operation, int32 firstOp, int32 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, int32)=>(int32) as ftype;

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

  int32 doOperation(ftype operation, int32 firstOp, int32 secondOp) {
    return operation(firstOp, secondOp);
  }
}
```

There are also two helpful intentions: one extracts a **typedef** with the respective function type from an existing **Function**. The other one, when called on a function type, can create an exemplary function.

7.1.9 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

.

```
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;
  }
}
```

7.1.10 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.

7.1.11 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;**).

7.1.12 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[3] array = {1, 2, 3};

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

    int8[2][2] array2 = {{1, 2}, {3, 4}};
    assert(1) array2[1][1] == 4;
  }
}
```

7.1.13 Structs and Unions

■ **Initialization** We support initialization expressions for **structs** and **unions**. Both are quite close to regular C. Here is the one for **structs**:

```
struct Point {
  int8 x;
  int8 y;
};

Point p = {
  10,
  20
};
```

The one for **unions** is a bit different from regular C, since it does mot require the leading dot before the referenced member:

```
union U {
  int8 m1;
  boolean m2;
};

U u1 = {m1 = 10};
U u3 = {m2 = true};
```

■ **The with Statement** Just like Pascal, mbeddr C supports a **with** statement for **structs** that avoids repeating the context expression[1]. Here is an example:

```
struct Point {
  int8 x;
  int8 y;
};

with (aPoint) {
  x = 10
  y = 20
};
```

The **with** statement resides in the **core.util** language.

7.1.14 Reporting

Reporting (or logging) is provided as a special concept. It's designed as a platform-independent reporting system. With the current generator and the **desktop** setting in

[1]Note that since the generator generates code that evaluates the context expression several times, the context expression must be idempotent. An error is reported if that is not the case.

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 the primitive types are supported (an error message is flagged in the editor if you use an unsupported type).

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 actualArgCount) active:
        No argument has been passed in, although an arg is expected
  }

  int32 main(int8 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.

■ **Counting Message Calls** Sometimes it is useful to be able to find out if and how often a message was actually reported. This is particularly useful in testing when you

want to find out whether a message (which is supposed to report some kind of error) was actually called (and hence the error actually occured). To count message calles, you have to mark the particular message as **count**ed via an intention:

```
message list CalcMessages {
  WARN ppcfailed(int8 operation, int8 ppc) active (count): ppc failed
}
```

You can then use the **messagecount** expression to retrieve the count:

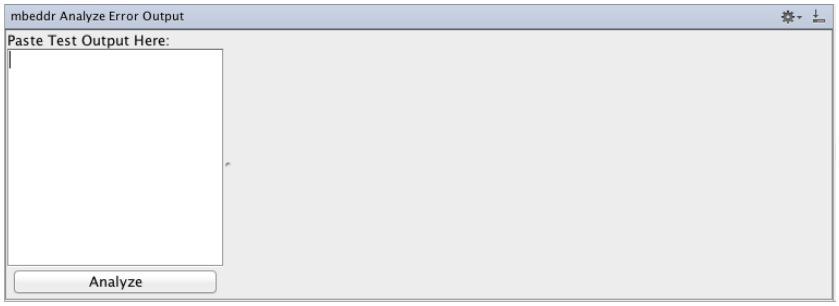
```
exported test case testComputer {
  computer.add(1, 1);
  assert(0) messagecount(CalcMessages.ppcfailed) == 0;
}
```

■ **Finding the Node that reported a Message** Reporting is used to report failed assertions in tests and other problems with program execution. Efficiently finding the node that reported a message (eg. a failed assertion) is important. There are two ways to do this. The first one includes the location string. Here is the default output of a report message:

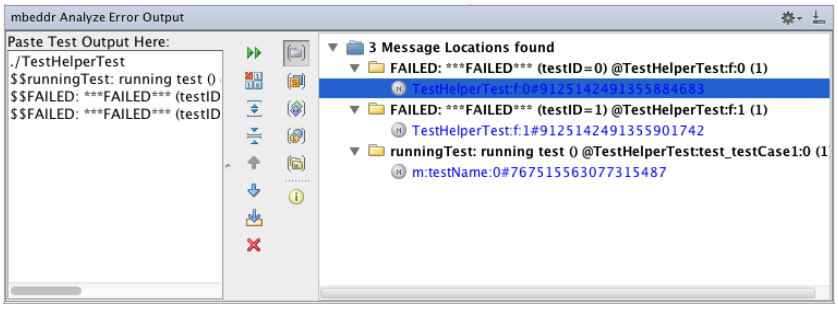
```
./TestHelperTest
$$runningTest: running test () @TestHelperTest:test_testCase1:0#767515563077315487
$$FAILED: ***FAILED*** (testID=0) @TestHelperTest:f:0#9125142491355884683
$$FAILED: ***FAILED*** (testID=1) @TestHelperTest:f:1#9125142491355901742
```

Each message starts of with the **$$**. Next is the error message and the list of message parameters. After the @ sign we the message shows the message location. The format is **module:content:index**, where the **index** is also shown in the respective **report** or **assert** statement (the number in parentheses). Based on this information, the location can be found manually.

However, mbeddr also comes with a tool that simplifies finding the message source. Using *Analyze* → *mbeddrAnalyzeErrorOutput*, arbitrary test program output can be analyzed. After selecting the menu item, the following view opens:



You can then paste arbitrary text that contains report messages into the text area (for example the example above). Pressing the **Analyze** button will find the nodes that created the messages (using the Node ID; this is the long number that follows the # in the message location).



You can then click on the node to select it in the editor (you should play a bit with the options buttons on the left; in particular the one with the grey folder (top right) is useful, since it shows the actual error message).

You can update the error output text in the text area at any time; press the button with the two green arrows to refresh the found nodes on the right.

■ **Logging Expressions** For debugging purposes it is often useful to log expressions without manually creating a message and a report statement. To this end, the log expression can be used. Here is some example code:

```
exported test case testLogExpressions {
  int8 x = 3;
  int8 y = log:0<x>;
  int8 z = log:1<3 + log:2<x>>;
  int8 zz = 3 * log:3<x>;
}
```

The log expression **log:n<...>**, where **n** is the index of the log expression reported in the output, can be attached to any expression *except literals* via an intention or a **log** left transformation.

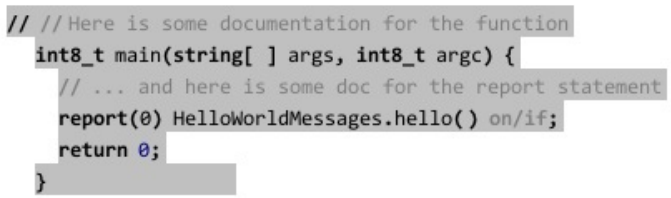
7.1.15 Assembly Code

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

7.1.16 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.



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 Find References, and refactorings may affect the code. However, type system errors are not reported inside commnted code. Also, commented program elements cannot be refernced (existing references are marked as invalid). 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, as well as many other useful things[2].

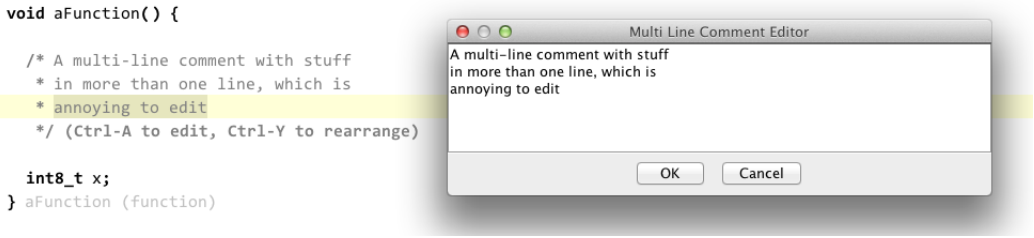
■ **Documentation** mbeddr supports comments. There are several kinds of comments that can be used: single line statement comments, multi-line statement comments and element documentations.

[2]However, let us know if we have to support more commentable things.

Single line statement comments are comments that can be used *only* in statement context. For example, in a function body, you can type `//` and enter arbitrary text. 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[3], 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, just like in normal textual programming). Second, as mentioned above, they can only be used in statement context. A better solution in many cases is to use the element documentation. It is structurally 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

[3]This is a problem with MPS' editor and will be fixed in future release.

element documentation, as well as many other concepts such as states in state machines or ports in components.

■ **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):

```
exported test case gotoTest {

  /* Here is a multiline comment.
   * It has some TODO(cat) where I need to do something.
   */ (Ctrl-A to edit, Ctrl-Y to rearrange)

  int8_t x = 0;
  goto ende;
  fail(0);
  /* Here is some documntation with a TODO comment.  */ (Ctrl-A to edit, Ctrl-Y to rearrange)
  label ende:
  assert(1) x == 0;
} gotoTest(test case)
```

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.

7.1.17 Function Modifiers and pragmas

In C, functions and variables often have modifiers that are either "processed away" by macros or can be understood by some proprietary compiler. In any case, one has to be able to mark up functions and variables with such modifiers. In mbeddr this is possible by by selecting the **`Add Modifier`** intention on functions and global variables.

To create a new kind of modifier, you have to create a concept that extends **Prefix**.

While the preprocessor is not supported in general, **#pragma**s are supported top level and in function bodies with the usual syntax.

7.1.18 Opaque Types

In C it is possible to define an empty **struct** (as in **struct o;**) and then use this type *only as a pointer*, even though it is not really defined. mbeddr supports this via a first class concept **opaque o;**, which can also just be used as a pointer.

7.2 Command Line Generation

mbeddr C projects can be generated from the command line via **ant** – and through **ant**, they can be included in CI servers. To generate the necessary ant build file (**build.xml**), mark the respective project in the mbeddr/MPS IDE and select **(Re-)Generate and build file for Project** from the **Code** menu. This creates two files, both in the project root directory (next to the **.mpr** file): a **build.xml** and a **build.properties**.

The **build.xml** file contains the invocation of the MPS generator; its **build** target performs the actual generation. No changes is ever necessary in this file.

The **build.properties** contains local settings. You have to specify the MPS directory (e.g., the **MPS2.4.app** directory on a Mac) and a temp directory where MPS can create some temporary files.

To run the generator, invoke **ant** in the project directory.

7.3 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**.

7.3.1 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.

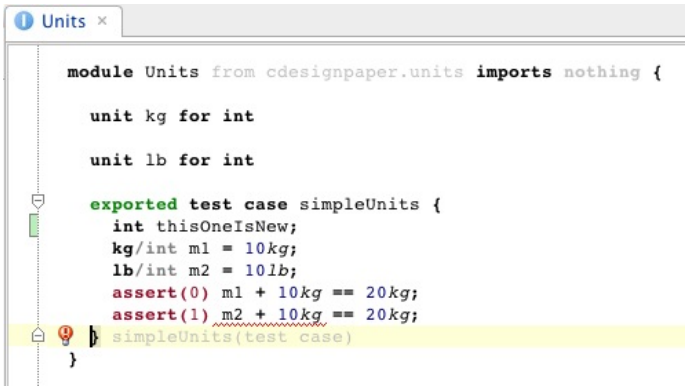
Make sure the **.history** files are *not* added to the **gitignore**! These are important for MPS-internal refactorings.

■ **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.

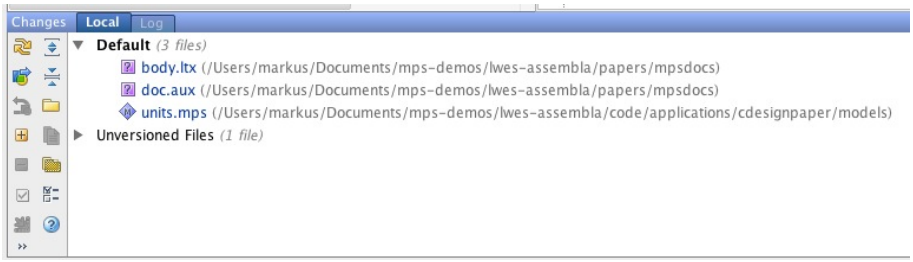
7.3.2 Committing Your Work

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. The screenshot below 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 (see figure below) 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.



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$
```

7.3.3 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.

7.3.4 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".

7.4 Debugging

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

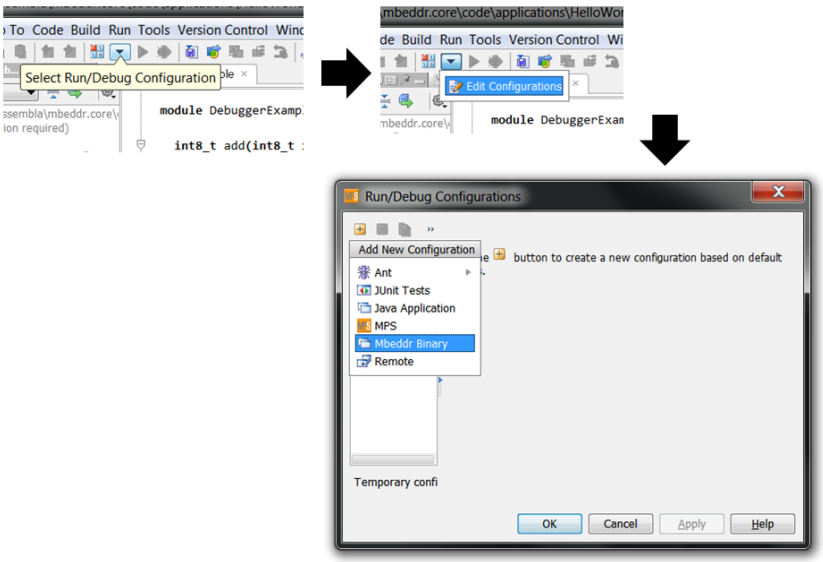
Note: We assume the default configuration, where we use **gdb** as the debug backed. Debugging on some target device is not yet supported.

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.

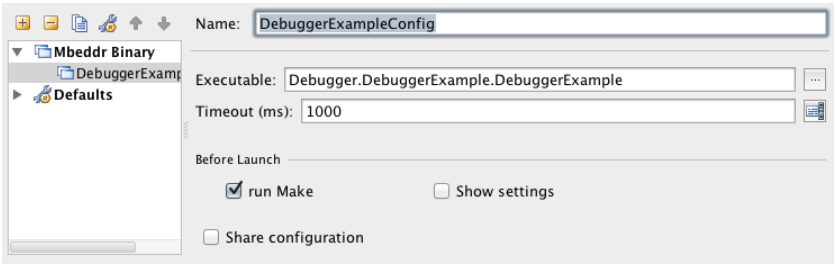
```
module DebuggerExample from Debugger.Example imports nothing {

  int8 add(int8 x, int8 y) {
    return x + y;
  }

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

  int32 main(int32 argc, int8*[ ] argv) {
    return test testAdding;
  }
}
```

7.4.1 Creating a Debug Configuration

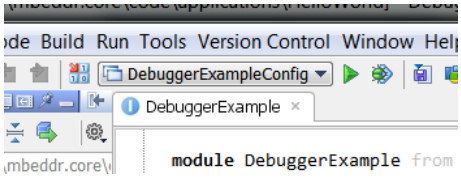
We start out by creating a new debug configuration as shown in the figure below:



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). **`Timeout`** controls in *ms* how long the debugger should wait for a connection setup to the underlying C debugger (increase this value, if your experience problems) and **`run Make`** creates a executable with debug symbols in it (necessary for the debugger).



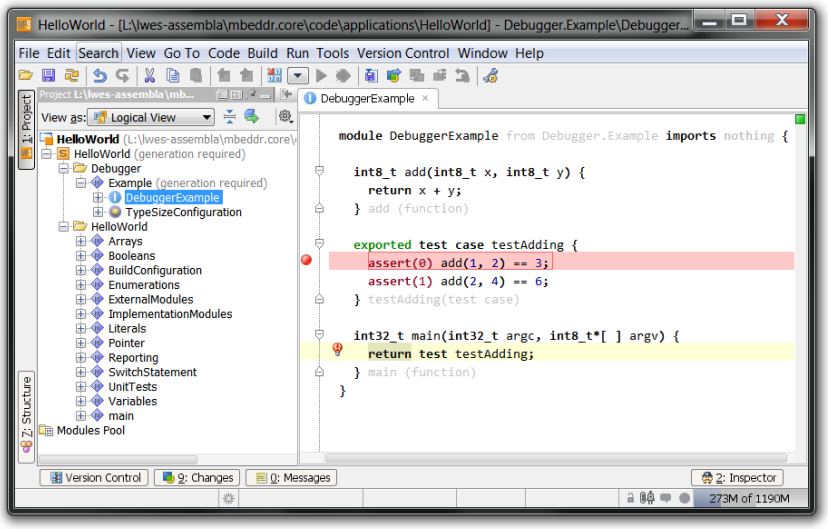
The launch configurations drop down at the top of the MPS screen should now show this new configuration:



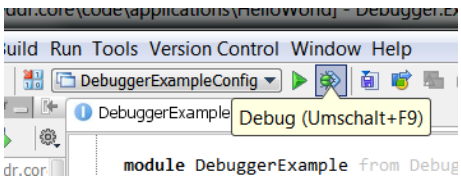
7.4.2 Running a Debug Session

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**.

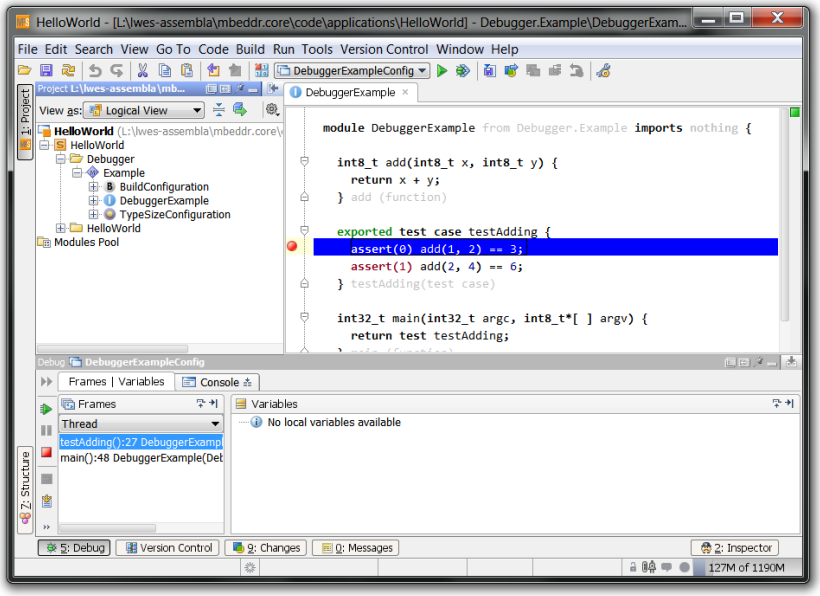
Now set a breakpoint in the first line of the test case. You can set breakpoints by clicking into the gutter of the editor. The result should look like this:



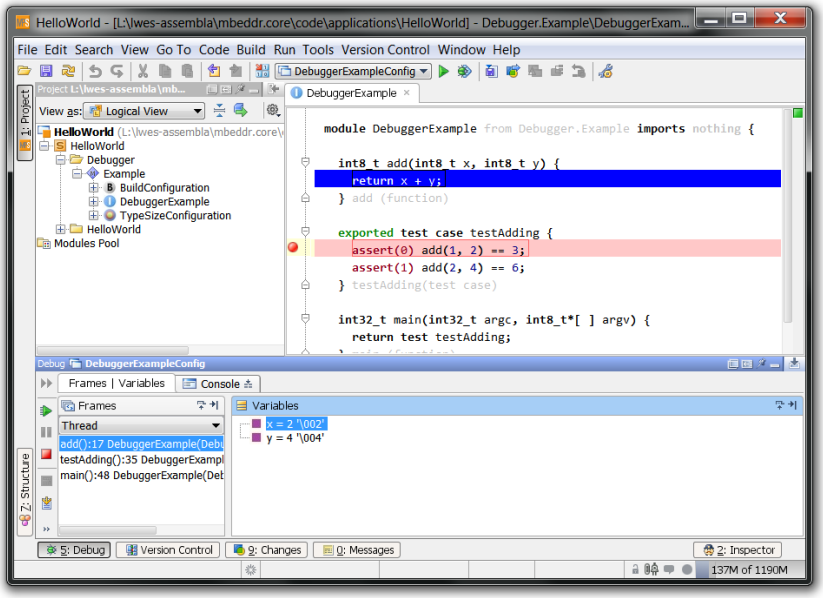
Next, run the previously created debug configuration by pressing **Shift-F9** or by selecting the debugger button in the MPS title bar (see next figure).



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**.



7.5 Data flow analyses

mbeddr comes with support for data flow analyses for the C core. Currently we can detect uninitialised and unused variables, unused assignments, missing return statements and dead code. Additionally we have implemented a constant detection analysis to detect variables which are constant through the program flow but are not defined as constants, and an analysis to track the values of variables. If you decide to use these two analyses you should add com.mbeddr.core.dataflow in the model properties as a used language. The data flow analyses are realised with the MPS data flow framework (for further information please take a look at the MPS Dataflow Cookbook and the dataflow section of the MPS user guide). If you want to inspect the data flow graph, which is used for the analyses, you can right click a node in your mbeddr C program and select **Show Data Flow Graph** from the **Language Debug** context menu. If you decide that the provided data flow information is not useful, we have introduced an annotation to disable the analyses. This annotation can be attached to e.g. a C function with the **Toggle Do Not Analyze Data Flow Annotation** intention. Additionally you have the possibility to disable all or a specific analysis in the **Analyze** menu of the main menu.

This section provides a short overview of these analyses and explains the warning or error messages with the help of some examples.

7.5.1 Uninitialized Variables

Writing and reading variables is modelled with MPS data flow read and write instructions.

```
void uninitializedVariablesExample() {
  int32 a = 1;
  int32 b;
  int32 c;
  if (rnd) {
    c = 1;
  } if
  a++;
  b++;
  c++;
} uninitializedVariablesExample (function)
```

The usage of an uninitialized variable is detected if there is no write instruction before a read instruction of the corresponding variable in the data flow graph. In the example function the reference to variable **a**, used in the post increment expression, is not marked with a warning or error message, as the variable has been initialised in the variable declaration. On the other hand the references to the variables **b** and **c** could lead to errors. The reference to variable **b** is marked with an error message, as the analysis detects that the variable is not initialized. The reference to variable c is marked with a warning message indicating that the variable may not have been initialised.

7.5.2 Unused Variables

Variables which are never used can be detected by looking for variables for which neither a read nor a write instruction exists in the data flow graph. If such a variable is detected the variable declaration is marked with a warning message. To detect variables as unused which are initialized (a write instruction exists in the data flow graph) but never read, the data flow graph is searched for write instructions with no consecutive read instruction. In this case, the initializer of the variable is marked with a warning message. The following figure contains an example C function with an unused variable and a redundant initializer.

```
void unusedVariablesExample() {
  int32 a;
  int32 b = 1;
} unusedVariablesExample (function)
```

7.5.3 Unused Assignments

Unused assignments can be detected by searching for write instructions with no consecutive read instruction of the same variable in the data flow graph. The C function **unusedAssignmentsExample** is a simple example for an unused assignment.

```
void unusedAssignmentsExample() {
  int32 a = 1;
  if (a == 1) {

  } if
  a = 0;

} unusedAssignmentsExample (function)
```

The `int32` variable `a` is initialised with the value `1`. Afterwards it is read in the condition of the if statement. Therefore the assignment is used. In an assignment expression the value `0` is assigned to the variable. As the variable is never used again, this assignment can be marked with a warning message, indicating the superfluous assignment.

7.5.4 Missing Returns

To represent return statements in the data flow graph a return instruction is provided by the MPS data flow framework. To detect C functions with missing return statements the last instruction of the data flow graph is used. If any path to this instruction through the data flow graph with no return instruction exists, a return statement is missing. The following figure shows an example C function with a missing return.

```
boolean missingReturnsExample(boolean parameter) {
  if (parameter) {
    return false;
  } if

} missingReturnsExample (function)
```

7.5.5 Dead Code

Dead code regions can be detected by searching the data flow graph for program flow paths which are not reachable from the first instruction of the data flow graph. An example is given in the following figure.

```
void deadCodeExample() {
  int32 i;
  if (1 == 2) {
    // dead code
    i = 1;
  } else if ( 1 == 1 ) {
    // live code
    i = 2;
  } else {
    // dead code
    i = 3;
  }

} deadCodeExample (function)
```

The first condition will always be false, therefore the then part of the if statement can not be reached. As the condition of the else if part is always true, the else part can never be reached too. Therefore there are two dead code regions in the example C function, which are marked with error messages.

7.5.6 Constant Detection

With a constant detection analysis we can detect variables, which have a constant value for all executions of the program, but are not declared as a C `const`. Currently only assignments which assign a statically evaluatable expressions to a variable are used for this analysis. If an not evaluatable expression is assigned to a variable, it is assumed that this variable is not a constant. The following figure shows an example C function for the constant propagation analysis.

```
void constantPropagationExampe() {

  int32 a = 1;
  int32 b = 1;

  if (rnd) {
    b = 2;
  } if
} constantPropagationExampe (function)
```

The value assigned to the variable a is 1 for all executions of the program, throughout the complete program. Therefore the variable declaration is marked with a warning message. To disable this message, use the Toggle Const intention with the variables's type node selected. If the value of a variable changes while executing the program, in the above C function the value of b can be either 1 or 2, the variable declaration is no longer marked as a constant.

7.5.7 Constant Propagation

Similar to the constant detection analysis, we track the values of variables through the program flow. This analysis is used to mark expressions with warning messages, if the evaluation of the expression at this point in the program flow will always return the same result.

```
void valuePropagationExample() {
  int32 a = 1;
  int32 b = 2;

  if (a < b) {
    // some code
  } if

} valuePropagationExample (function)
```

In the C function above, the values of the variables a and b are tracked. Therefore the expression `a < b` can be statically evaluated to true, which is indicated as a warning.

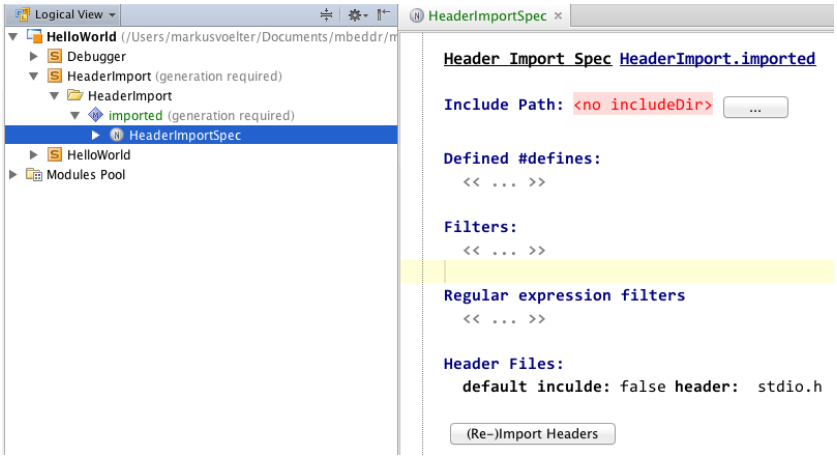
7.6 Importing existing Header Files

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 7.1.2, 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.

7.6.1 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:



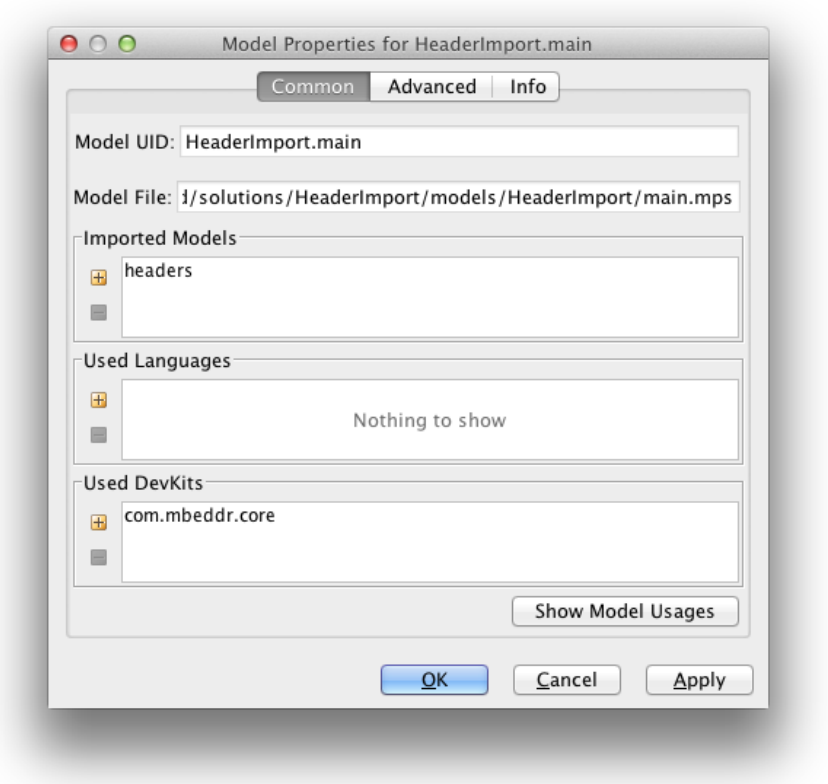
In the **path** property we set the path to a directory that contains the set of header files we want to import. Note that you can use the **...** button to bring up a directory selection dialog. You can also use MPS' path variables in the path specification, using the familiar **${the.var.name}** notation[4].

Once you have specified the path, you can press the **(Re-)import Headers** button. This will import the headers in the specified directory. A result dialog will report any problems with the import.

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.

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:

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



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 main(int32 argc, int8*[] 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**.

7.6.2 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[5].

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

[5]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.

■ **Header Files** This sections shows all header files located in the directory defined in the **`Include Path`** property. Sometimes a header file does not include some other header file explicitly, although it should. The compiler seems to not care ... mbeddr, does care, however. By marking a header file as **`default include`**, this header will be added to the **`import`** section of all other headers.

7.6.3 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 **`#ifdef`**s 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) 10 + SOME_OTHER_DEFINE + 3
```

8 mbeddr.ext — Default Extensions

8.1 Physical Units

Physical Units are new types that, in addition to specifying their actual data type, also specify a physical unit (see figure below). 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 correct type checks for types with units. The type system also performs unit computations to, for example, handle an **speed = length/time** correctly.

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.

```
derived unit mps = m s^-1 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;
}
```

To use physical units use the **com.mbeddr.physicalunits** devkit in your model.

8.1.1 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/m/ length;
int8/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/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/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/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/m/ someLength;
int8/m/ result = 10 m + someLength;
```

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

```
int8/m/ someLength = 10 m;
int8 justSomeValue = stripunit[someLength];
int8/m/ someLength = introduceunit[justSomeValue -> m];
```

8.1.2 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:

```
Unit Declarations
derived unit N = kg m s^-2 for force
derived unit Pa = N m^-2 for pressure
```

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

8.1.3 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:

```
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
```

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/F/ tempInF = 10 F;
int8/C/ tempInC = [tempInF -> C];
```

8.1.4 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/#global/ aGlobalVariable = 10#global;
int8/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/#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;
```

8.2 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.

8.2.1 Basic Interfaces and Components

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

■ **Interfaces** There are two kinds of interfaces. Client/Server interfaces contain a set of operations that can be called by client components. Sender/Receiver interfaces are used to share data among components. In the rest of this section we first look at client/server interfaces and later at sender/receiver interfaces.

An client/server 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 speed, uint32 ms)
  void driveBackwardFor(uint8 speed, uint32 ms)
  void driveContinouslyForward(uint8 speed)
  void driveContinouslyBackward(uint8 speed)
  void stop()
  query uint8 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 speed, uint32 ms) <- op dt.driveForwardFor {
      motorLeft.set_speed(((int8) speed));
      motorRight.set_speed(((int8) 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.

Runnables can be marked as **inline**. In this case, an invocation of such a runnable will be inline, no method invocation (in the generated code) will occur. At this point, inline runnables cannot return any values; in other words, they must return **void**.

A required port can be marked (via an intention) as **multiple** which means that a single required port can be connected to *several* provided ports. We call such a port a *multi-port*. The number of provided ports in can be connected to must be specified in the port:

```
exported component Subject {
  requires ObserverIf i1 [2]
}
```

Note: Currently multi ports can only be used with interfaces that have *only* **void** operations (it is not clear what to do with a collection of retuen values).

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 **when connected (optionalReqPort) { .. }** statement. The body of the **when connected** 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 speed, uint32 ms) <- op dt.driveForwardFor {
      when connected(log) {
        // other stuff
        log.error(...)
        // more other stuff
      }
    }
    ...
}
```

Note that if an invocation is tried on a optional port without wrapping it in a **when connected** statement, mbeddr reports and error. A quick fix is available to add the **when connected** statement. **when connected** statements can also have an **else** part (using the expected syntax).

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

```
exported component OrienterImpl extends nothing {
    int16[5] headingBuffer
    int8 headingIndex

    void orienter\_orientTowards(int16 heading, uint8 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 {
    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 main(int32 argc, int8*[ ] argv) {
  initialize defaultInstances;
  ...
}
```

An instance configuration can contain several connectors for the same required port of a given instance. If the required port is a multi port, the connector shows the **multi** modifier. If the required port is a regular to-one required port, the the later connector *overrides* the former ones, so in effect each required is connected only once. The overriding connector is marked as **override**. You can **Ctrl-Click** onto the *override* keyword to navigate to the connector that is overridden by the current **override** connector.

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

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

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

8.2.2 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.

8.2.3 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 speed, uint32 ms)
    pre(0) speed < 100
    post(1) currentSpeed() == 0
    protocol init -> init
  void driveContinouslyForward(uint8 speed)
    pre(0) speed <= 100
    post(1) currentSpeed() == speed
    protocol init -> forward
  void accelerateBy(uint8 speed)
    post(1) currentSpeed() == old(currentSpeed()) + speed
    protocol forward -> forward
  query uint8 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.

8.2.4 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 size;
  int8* 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 {
      requires PersistenceProvider pers
      provides Driver d

      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 {
    provides PersistenceProvider pp
    sequence {
      step 0: pp.isReady return false;
      step 1: pp.isReady return true;
      step 2: pp.store {
          0: parameter data: data != null
        }
      step 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. Note that optionally, a **step** can also have a body that contains arbitrary procedural code. Mocks can also have fields, just like any other component.

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.

8.2.5 More Test Support

In the context of testing, it makes sense to call runnables of component instance directly, without going through their respective triggers (polymorphic interface calls or interrupts, etc.). This is also useful for calling internal (i.e. "private") runnables that do not have any trigger.

This is why inside test cases, you can use the direct runnable call expression:

```
exported component C extends nothing {

    int8 count = 0

    internal int8 getStuff(int8 x) <= no trigger {
      count++;
      return count;
    }
}

instance configuration instances extends nothing {
    instance C c1
}

exported test case testCall {
  assert(0) $instances:c1.getStuff(10) == 1;
}
```

8.2.6 Sender/Receiver Interfaces

A sender/receiver interface does not declare operations. Instead it declares data items:

```
exported sr interface PositionProvider {
  int8 x;
  int8 y;
  uint16 alt;
}
```

A component that *provides* this interface is supposed to provide values for these items. A component that *requires* a sender/receiver interface is supposed to read these values:

```
module ComponentsSRI imports DataStructures {

  exported component GPS extends nothing {
    provides PositionProvider pos
    void update() {
      pos.x++;
      pos.y++;
    }
  }

  exported component FlightRecorder extends nothing {
```

```
    requires PositionProvider pp
    Trackpoint[1000] recordedFlight;
    uint16 count = 0;
    void timed_tick() {
      with (recordedFlight[count]) {
        x = pp.x
        y = pp.y
        alt = pp.alt
      };
      count++;
    }
  }

}
```

Components still have to be instantiated, and the ports connected. As with client/server interfaces, several required ports can connect to the same provided port.

There are two options for sender/receiver interfaces. First, an interface can be marked as **strict**. In this case, *only the provider* can write the values defined in the interface, and the requirers can only read them. Data flow is in one direction. If interfaces are not strict, then requirers can also write.

So you may ask what is the difference between providing and requiring? The difference is who owns the data. If you have one provider and two requirers, then there is *one* set of data at runtime. If one requirer writes to the data item, the other requirer will see the change.

A second option is that a data item can be marked as **atomic**. This means that it can only be set in one piece. This is especially useful when **struct**s are used as the type of a data item. For non-atomic data items, you can assign to fields of complex data items (as in **carData.position.x = value**):

```
exported struct Position {
  int8 x;
  int8 y;
};

exported sr interface CarData {
  Position position;
}

exported component Motor extends nothing {
  provides CarData carData
  internal void updateCarDataPositionX(int8 value) <- no trigger {
    carData.position.x = value;
  }
}
```

If the **position** item were marked as **atomic** only the complete **struct** could be assigned (as in **carData.position = otherPosition;**).

Both **atomic** and **strict** are necessary for certain middleware platforms. In a pure-C/no-middleware implementation, these two options are not relevant.

8.3 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**.

8.3.1 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 initial = start {
    in reset()
    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 reset()
  in 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.

```
  var int[0..100] current = 0
  var int[0..100] LIMIT = 100
  var 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 {  }
}
```

Statemachines can be marked as **strict** (by using an intention). If a state machine is triggered and there is *no transition firing*, then an error is reported. Regular state machines just silently ignore this case. Also, for strict statemachines, you have to be specify a message reference (in the inspector) that will be reported if one of these errors occurs.

8.3.2 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. State machines also have to be initialized explictly using the **sminit** statement.

```
WrappingCounter wc;

void someFunction() {
  sminit(wc);
}
```

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

```
void someFunctionCalledByADriver(int8 ticks) {
  smtrigger(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); }
```

8.3.3 The complete WrappingCounter state machine

```
module WrappingCounterModule imports nothing {

  statemachine WrappingCounter initial = start {
      in increment(int[0..10] delta)
      in reset()
      out wrapped(int[0..100] steps) -> wrapped

      var int[0..100] curr = 0
      var int[0..100] LIMIT = 100
      var int[0..100] steps = 0

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

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

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

  WrappingCounter wc;
```

```
  void someFunctionCalledByADriver(int8 ticks) {
    smtrigger(wc, increment(ticks));
  }
}
```

8.3.4 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 {
  smtrigger(wc, reset);
  assert(0) isInState(wc, start);
  smtrigger(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).

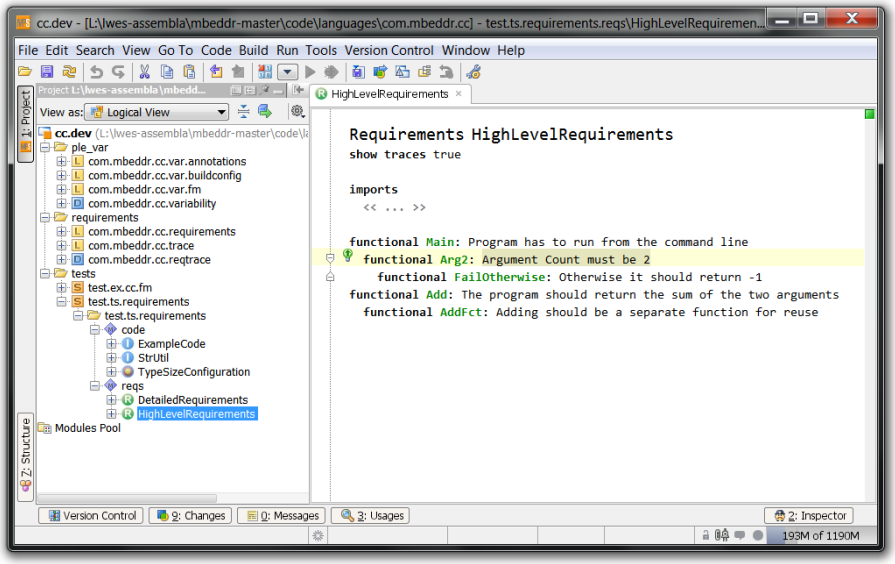
There is also a shorthand for checking if a state machine reacts correctly to events in terms of the new current state:

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

9 mbeddr.cc — Cross-Cutting Concerns

9.1 Requirements

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

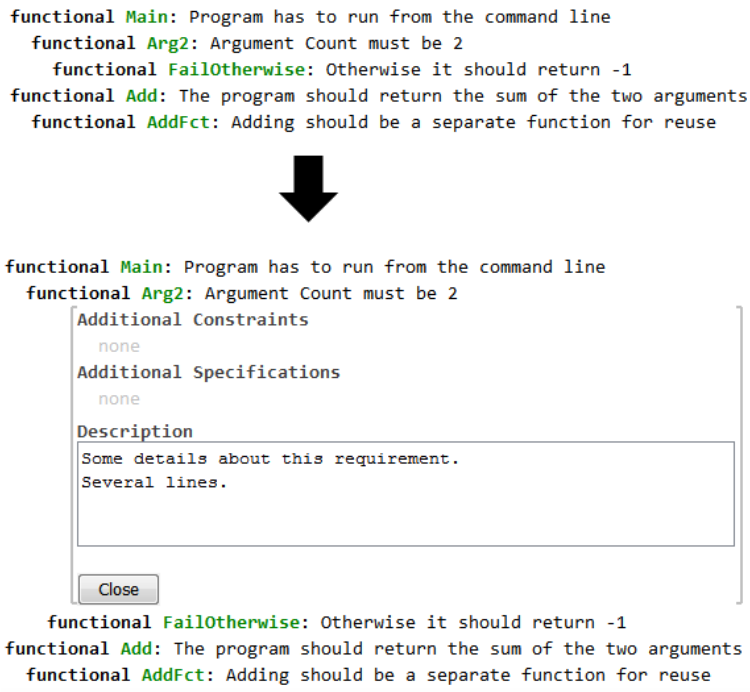


9.1.1 Specifying Requirements

Requirements can be collected in instances of **`RequirementsModule`**, a root concept defined by the **`com.mbeddr.cc.requirements`** language. An example is shown above. 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.

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**:



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 **conflicts with** or **requires also**.

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

■ **Filtering and Summarizing** It is possible to filter requirements when viewing them in a requirements module. At the top right of a requirements module you can select from a set of filters, including filtering by **summary**, **name**, or **trace status**. Filters are and'ed together by default. However, explicit **and** and **or** filters are available as

well. New filters (which may filter on project specific additional data) can be created by extending the **RequirementsFilter** concept and implementing its **matches** behavior method. As an example, here is the code for the **name contains substring** filter:

```
public boolean matches(node<Requirement> r)
  overrides RequirementsFilter.matches {
  if (this.substring == null || this.substring.equals("")) { return true; }
  r.name.contains(this.substring);
}
```

A similar facility exists for summarizing requirements. The only currently implemented summarizer counts the requirements (taking the filters into account). Custom summarizers may be used, for example, to sum up efforts.

■ **CSV Import** Requirements can be imported from a CSV file. To do this, use the **requirements.csv** language in your program and use the respective intention to attache a **CsvImportAnnotation** to your requirements module:

```
CSV Import from ${smartmeter.git.root}/planning/myCSVFile.csv
Separator: ;
Quote Char: "
Mapper: smart meter default clear before import: false
Status: Successfully imported on 2012/07/04 21:45:30

Requirements SomeRequirements        show traces true
                                     filters << ... >>
```

There, you can specify the CSV file from which to import (you can use MPS path variables there!), as well as the element separator (defaults to **;**) and the quotation character (defaults to **"**). To run the actual import, use the **(Re-)Import** intention.

The mapping from the CSV file fields to the contents of a **Requirement** node is of course project specific. Hence this mapping is modularized by an **IRequirementsMapper**. You have to create an implementing concept for your project. In the respective behavior you have to implement the following method:

```
public virtual abstract boolean map(string[] elements, node<Requirement> req);
```

The framework passes in a CSV line (as an string array) as well as a new **Requirement**. The implementation of the method has to fill the requirement with the data from the **elements** array. Two additional optional methods are available. **extractID** returns the

identifying string from the **elements** array (this is what will become the requirement ID). It defaults to the first element[1]:

```
public virtual string extractID(string[] elements, boolean dummy) {
  elements[0];
}
```

The second optional method **getParentRequirement** returns the requirement under which the newly created one should be attached. This is the way to construct requirement trees. It defaults to **null** which means that the new requirement has no parent — it is added to the requirements module on top level.

```
public virtual node<Requirement> getParentRequirement(
     node<RequirementsModule> rm, node<Requirement> req, string[] elements) {
  return null;
}
```

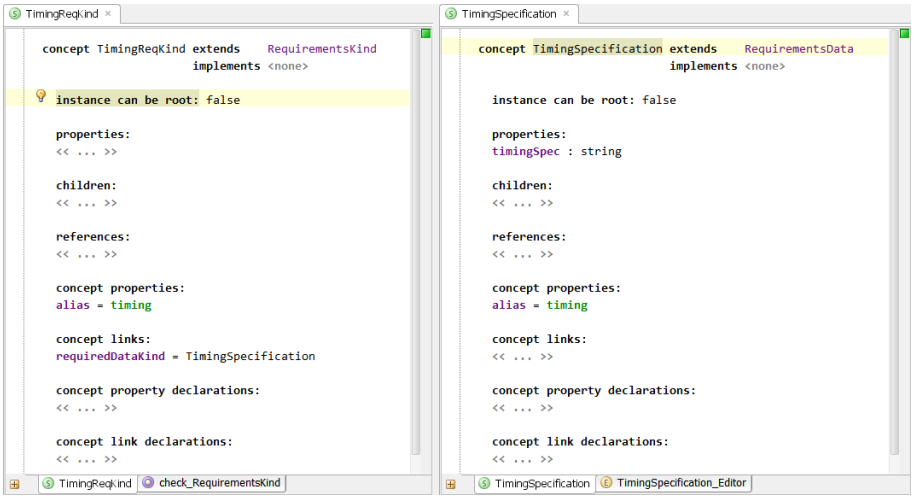
9.1.2 Extending the Requirements Language

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.

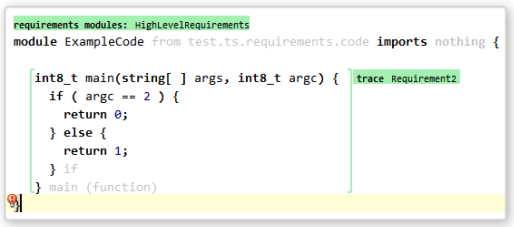
[1] The ID is important because when you reimport a CSV file, the **Requirement** nodes for existing requirements are kept the same so references to it do not break.



■ **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.

9.1.3 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. Below is 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**.

9.1.4 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 **TraceTargetProviderRefAttr**. It references **ITraceTargetProvider**s.
- Finally you have to implement a subconcept of **TraceTargetRef** that represents the actual reference to the new kind of target.

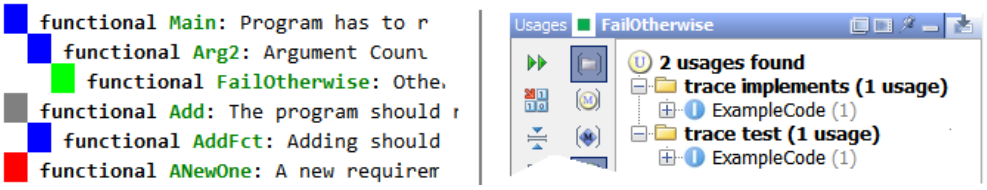
Here is how the whole system works: You attach a **TraceAnnotation** to a program element. It contains a set of **TranceTargetRef**s which in turn reference **ITraceTarget**s. To find the candidate trace targets, the scoping rule of **TranceTargetRef** ascends the tree to the current root and checks if it has something in the **traceTargetProviderAttr**

attribute. That would have to be a subtype of `TraceTargetProviderRefAttr`. It then follows the `ref`s references to a set of `ITraceTargetProvider` and asks those for the candidate `ITraceTarget`.

9.1.5 Evaluating the Traces in Reverse

The traces can be evaluated in reverse order. For example, as shown in the figure below, requirements can be color-coded to reflect their state. Traced requirements are grey, implemented ones are blue, and tested ones are green and untraced requirements are red. 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 (see below, right side).



9.2 Variability

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:

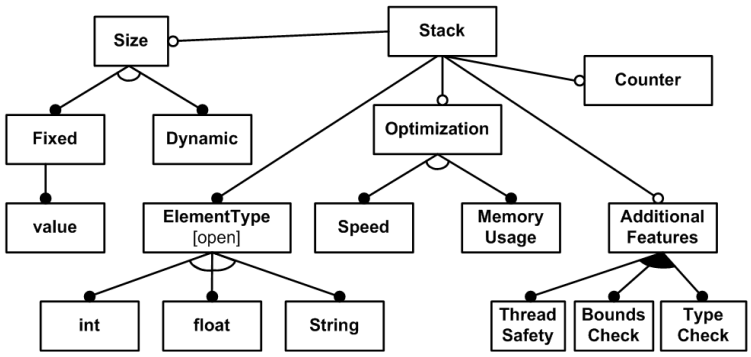
- `com.mbeddr.cc.var.fm` supports the definition of feature models. Feature models are a well-known formalism for expressing variability on a high-level, independent of the realization of the variability in software.

- **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 **#ifdef**s).

- **com.mbeddr.cc.var.buildconfig** ties the processing of annotations into mbeddr's build process.

9.2.1 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. An example is below.



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

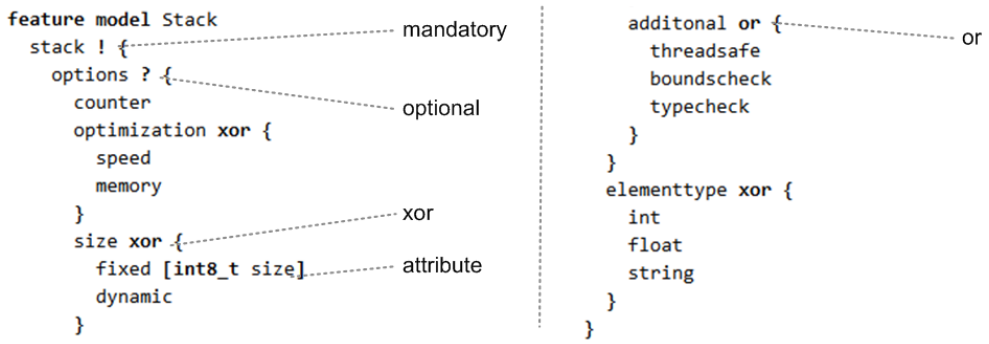
- **mandatory**, the filled circle: mandatory features have to be in each product variant. In the above example, each **Stack** has to have the feature **ElementType**.

- **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**.

The figure below 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. Below are 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
    }
  }
```

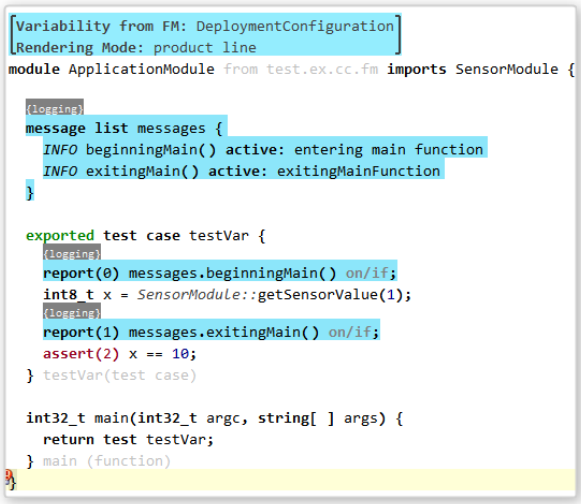
Note that, if you create invalid configurations by selecting feature combinations that are prohibited by the constraints expressed in the feature model, errors will be shown.

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

9.2.2 Connecting implementation Artifacts Statically

Static connection to artifacts means that artifacts are varied *during the generation process*. The mechanism provided by mbeddr is generic in that it can be used with arbitrary artifacts expressed in arbitraray (MPS-based) languages.

■ **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 the screenshot below 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.

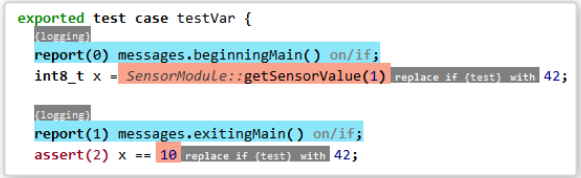


To use presence conditions, the root note (here: an implementation module) as to have a **`FeatureModelConfiguration`** annotation. It can be added via an intention if the **`com.mbeddr.cc.var.annotations`** language is used in the respective model. The annotation points to the feature model whose features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

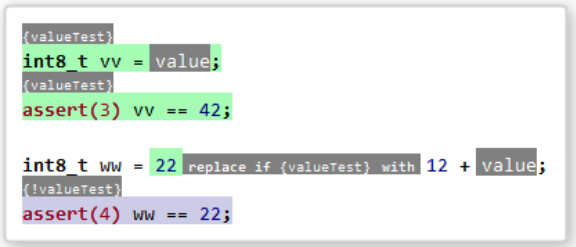
The background color of an annotated note is computed from the expression. Several annotated nodes that use the same expression will have the same color (an idea borrowed from Christian Kaestner's CIDE).

■ **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.



A replacement replaces the node to which it is attached with an alternative node if the condition is *true*. In the example in the figure below 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.

■ **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.

■ **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. The figure below 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.

■ **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 module. 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 models that contain the product line artifacts.

In each variant model you will have a build configuration that determines the variant that will be built. It does so by including a configuration item:

```
Build System:
  ...

Configuration Items
  variability mappings {
    fm DeploymentConfiguration -> Debug
  }

Binaries
  executable App isTest: true {
    ...
  }
```

This configuration item determines which configuration should be used for each feature model (**Debug** in the example). To add this item, you need to use the **com.mbeddr.cc.var.buildconfig** language in your model.

9.2.3 Implementing Variability in C code at Runtime

Runtime variability means that the code in the binary can implement all products of the product line (in some sense, it is the superset of all of them). The decision as to which alternative to run is performed at runtime. Connecting a program to a feature model this way is more invasive. The code has to be written to take the variability into account explicitly. Also, the language used to express the variable program has to provide constructs that connect to the feature models.

Note: Currently, support is provided for C statements. Support for additional languages will be provided later.

■ **Representing Feature Models and Configurations at runtime** To be able to make a variability decision at runtime, the information about the feature model and the configurations must be made available at runtime. A C data structure is used to represent the feature models. As of now, this data structure is a **struct** that has a boolean member for each feature[2].

The decision as to where (i.e. into which module) this data structure should go must be made by the developer. To do so, a new language construct is available. Consider the following feature model:

```
Variability Support FeatureModels

feature model someFM
  root ? {
    f1
    f2
    f3 [int8 attr]
  }
```

To embed the corresponding data structure into a module, use the following code (like any other module content, this element can be exported to make it visible from other (importing) modules):

```
module TestFeaturesOnly {
  feature model @ runtime for someFM
}
```

Once this has been embedded, a new type **fmconfig<..>** is available that can be used whereever types are supported (local variables, arguments, global variables, component fields). The type takes a **feature model @ runtime** as an argument:

[2] A later generator will use bit fields to reduce the size of the data in the running C program.

```
module TestFeaturesOnly {
  feature model @ runtime for someFM;
  fmconfig<someFM> currentConfig;
}
```

Variables of type **fmconfig<..>** can hold the runtime representation of a particular configuration model. Assigning a particular configuration to a variable types as **fmconfig<..>** happens via another special construct. Assuming you have a configuration model **Cfg3** for the **someFM** feature model, you can write:

```
module TestFeaturesOnly {
  feature model @ runtime for someFM;
  fmconfig<someFM> currentConfig;

  exported test case testAttribute {
    store config<someFM, Cfg3> into currentConfig;
  }
}
```

The **currentConfig** variable holds this configuration now; it can be passed around like any other data using the **fmconfig<someFM>** type.

■ **Writing Variant-Aware Code** As of now, variant-aware code relies on using a **switch**-like statement that switches over variants. In its simplest form it looks like this:

```
module TestFeaturesOnly {
  exported test case testAttribute {
    int8 x = 0;
    variant<currentConfig>(f1) { x = 1; }
  }
}
```

The **variant** statement refers to the configuration it depends on (**currentConfig** in the example) and then contains a boolean expression over features in the underlying feature model. In the example above we just check if the feature **f1** is selected in the configuration, but we could write more complex expressions as well. Also, arbitrary code can be put into the body of the statement:

```
variant<currentConfig>(f1 || f2 && !f3) {
  x = 1;
  int8 morestuff = x;
  callAFunction(x);
}
```

This example looks more like an **if** — and in fact, this is exactly how it works[3]. Using

[3]The reason why we don't just use an **if** is the support for verification described below.

an intention, more cases and a default can be added[4]:

```
variant<currentConfig> {
  case (f2) { x = 2; }
  case (f1) { x = 3; }
  default { x = 4; }
}
```

It is also possible to access the values of feature attributes from the code:

```
variant<currentConfig>(f3) { x = f3.attr; }
```

Note that using the **feature.attr** notation is only possible inside a **variant** statement.

■ **Configuring Components** A particularly useful case for runtime variability is the configuration of component instances. In this case you define a feature model for a component, and each instance has its own configuration model that determines the instance's configuration. Here is an example of a component:

```
exported component C extends nothing {

  init fmconfig<CompFeatures> cfg;

  internal int8 getNumber() <- no trigger {
    int8 x = 0;
    variant<cfg>(returnTheBigNumber) { x = 10; }
    return x;
  }
}
```

We have defined an init field for the component that is typed with an **fmconfig<..>** type that references a feature model. We can then use the well-known **variant** statement to make the code depend on features.

So far this is very similar to what we have described for non-component code earlier. The next step would be to use the **store config** to assign an instance configuration. However, **store config** is a **Statement**, and it cannot be used where instances are created. Instead, we have created special support (**CfgBig** and **CfgSmall** are two configuration models for the **CompFeatures** feature model):

```
instances exampleInstances {
  instance C cbig(config = CfgBig)
  instance C csmall(config = CfgSmall)
}
```

To use this additional construct, you have to use the **com.mbeddr.cc.var.rt.comp** language in your model.

[4]No **break** is needed in this construct. Only one **case** will ever by executed. The first one that matches wins.

■ **Verifying the Code** As of now, no verification for the code is available. However, in the future, the following verification will be supported:

- Is an expression in a `case` actually allowed based on the constraints expressed in the underlying feature models?
- Are the various cases mutually exclusive?
- If no `default` is given, are all other cases explicitly taken care of?
- When accessing a feature attribute: is the respective feature actually selected in the current context

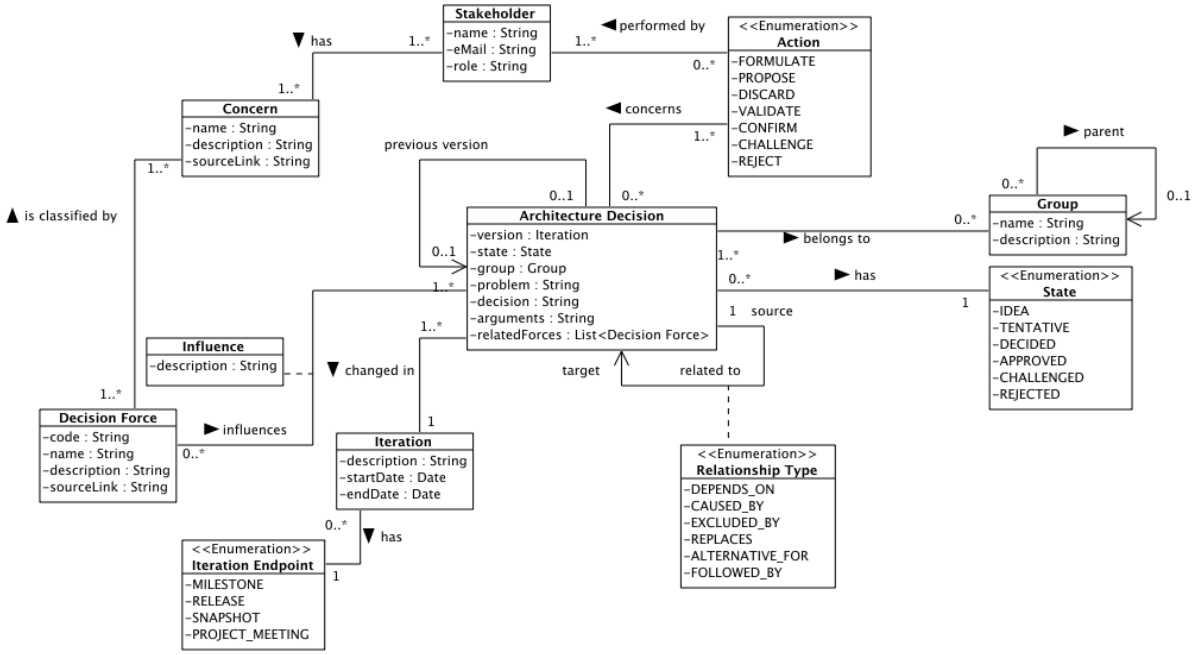
9.3 Architecture Decisions

Note: The implementation of this language is very new and not much tested yet. Expect bugs and improvements.

This extension supports the systematic documentation of (architectural) decisions. It is based on the approach by van Heesch, Avgeriou and Hillard:

> U. van Heesch, P. Avgeriou, R. Hilliard, A documentation framework for architecture decisions, Journal of Systems and Software, Volume 85, Issue 4, April 2012, Pages 795-820, Elsevier.

The following figure provides an overview over the meta model.



To use the architecture decisions, include the **`com.mbeddr.archdec`** devkit in your project.

9.3.1 Base Data

As can be seen in the above meta model, various base data are required to document architecture decisions in a meaningful way. These includes concerns, stakeholders, decision groups, and forces. These can be captured in **`Architecture Decision`** modules. The following figure shows an example.

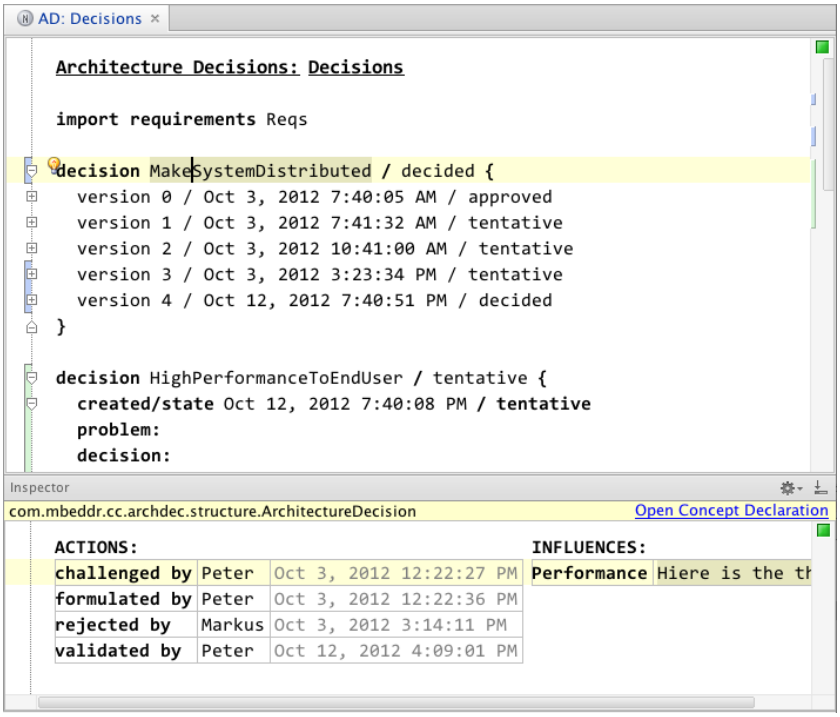
```
Architecture Decisions: GroupsAndBasics

concern Deployment : Deployment Concern
concern Development : Development Concern

group Initial: Decisions made initially in the project
group OnTheFly: Decisions made during the course of the project


force Performance:
  affects Deployment, Development

stakeholder Peter: End User (peter@company.de)
  cares about Deployment

stakeholder Markus: End User (peter@company.de)
  cares about Deployment, Development
```

9.3.2 Achitecture Decisions

The decisions themselves can be captured in **Architecture Decision** modules as well. Note that the decisions are versioned. Only the most recent version can actually be edited. Older versions are grey and read-only.

```
decision MakeSystemDistributed / decided {
  version 0 / Oct 3, 2012 7:40:05 AM / approved
  version 1 / Oct 3, 2012 7:41:32 AM / tentative
  version 2 / Oct 3, 2012 10:41:00 AM / tentative
  version 3 {
    created/state Oct 3, 2012 3:23:34 PM / tentative
    problem: How can we guarantee up time in the face of catastrophic failures
    decision: We will geographically distribute the system.
    arguments: We could have hardened the building, but distribution is more robust.
    relates to: depends on HighPerformanceToEndUser
    belongs to: Initial
  }
  version 4 {
    created/state Oct 12, 2012 7:40:51 PM / decided
    problem: How can we guarantee up time in the face of catastrophic failures
    decision: We will geographically distribute the system over several states.
    arguments: We could have hardened the building, but distribution is more robust.
    relates to: depends on HighPerformanceToEndUser
    belongs to: Initial
  }
}
```

For architecture decisions (not for their versions!) the inspector contains information about which stakeholder interacted with the decision and which forces influence it. Each influence points to a decision and can contain a short rationale.



9.3.3 Connection to Requirements

In principle, every program element can be connected to requirements by the generic tracing framework (see Section 9.1.3). However, since the connections between architecture and requirements are especially typical, first class support is provided.

To connect a decision to a requirement, you first have to import a requirements module into the architecture decision module using the **`import requirements`** construct. You can then add requirements to the influences section of a decision using the **`req`** keyword in a decision's influences.

9.3.4 Tracing to Architecture Decisions

As discussed in Section 9.1.4, other trace targets can be plugged into the existing trace mechanism. Architecture decisions are such an alternative trace target. Hence, program elements can be connected to architecture decisions via traces.

10 mbeddr.analyses — Default Formal Analyses

Besides improvements in productivity, the higher level extensions of C allow the definition of automatic and usable formal analyses. In this chapter we describe the use of these analyses.

10.1 Checking Decision Tables

10.1.1 Installation

Decision tables checking is based on the Yices 1 SMT solver:
`http://yices.csl.sri.com/download.shtml`

To be able to use these analyses, the solver has to be installed on your machine and available in the search path.

10.1.2 Available Analyses

A decision table is a more elegant and explicit way of coding two dimensional choices. Internally they are represented as nested `if`-statements. For decision tables we define two analyses:

1. **Completeness** checks if the decision table covers all possible cases.

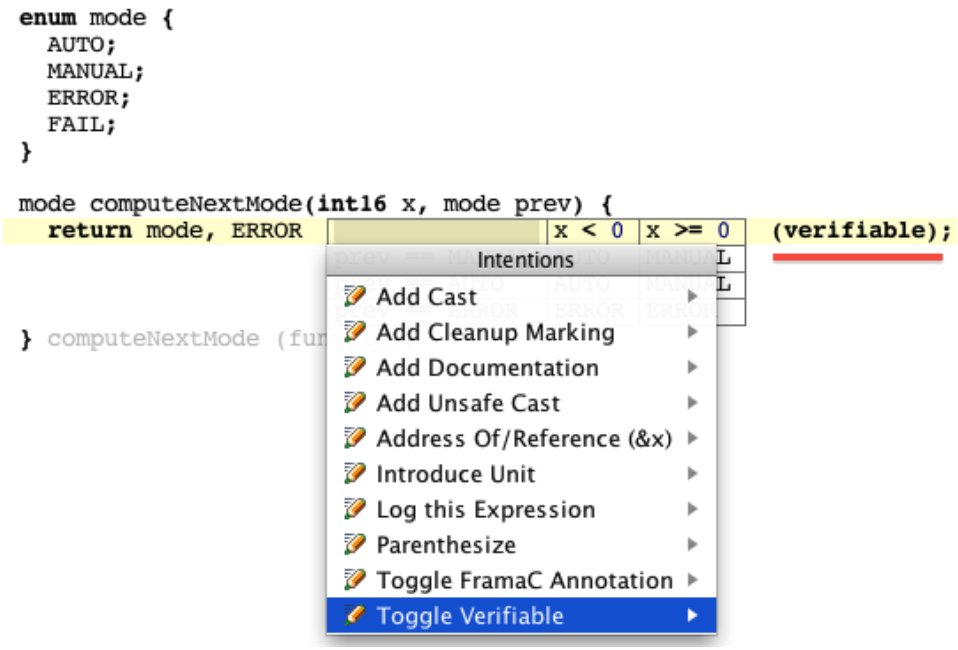
2. **Consistency** checks if there are cases for which two cells of the decision table can be active at the same time.

As long as the expressions used as conditions are "simple", the above analyses can be automatically performed by using an SMT (satisfiability modulo theories) solve.

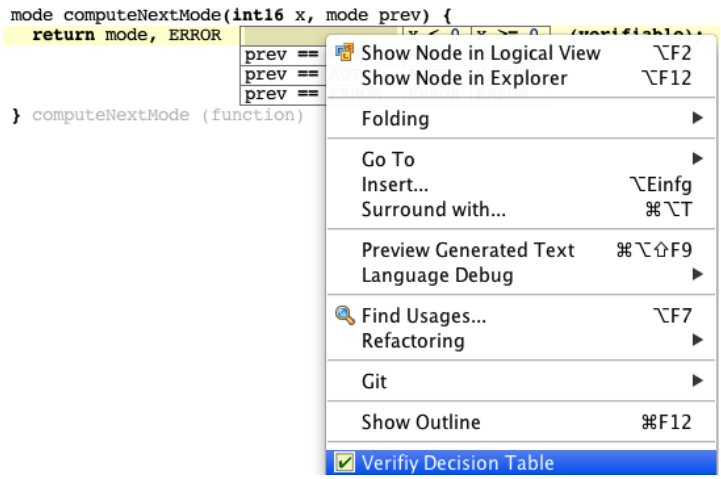
10.1.3 Performing the Analyses

The first step for analyzing decision tables is to add the devkit **com.mbeddr.analyses.dectab** to the model that contains the decision table.

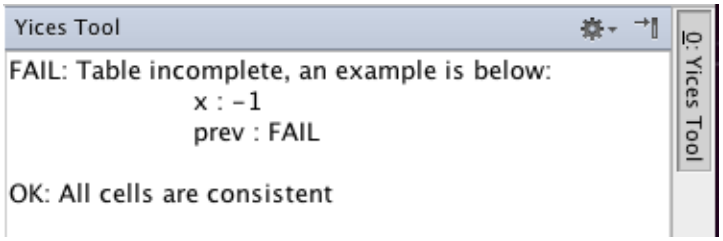
Once the devkit is added to the model, you can mark a decision table as **verifiable** by using an intention on the left-upper corner of the decision table. Once the **verifiable** flag is set it will be displayed on the right-hand-side of the table (underlined with red in the figure below).



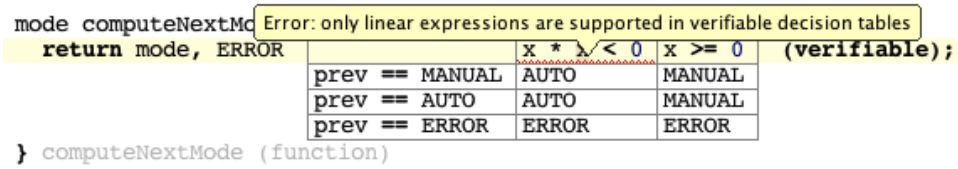
On a decision table marked as verifiable, the verification can be run with the help of a context menu entry.



Once the action is started, the corresponding Yices code is generated, Yices is started in the background and the results of the verification results will be lifted (i.e. interpreted in the context of the decision table) and presented in a separate window:

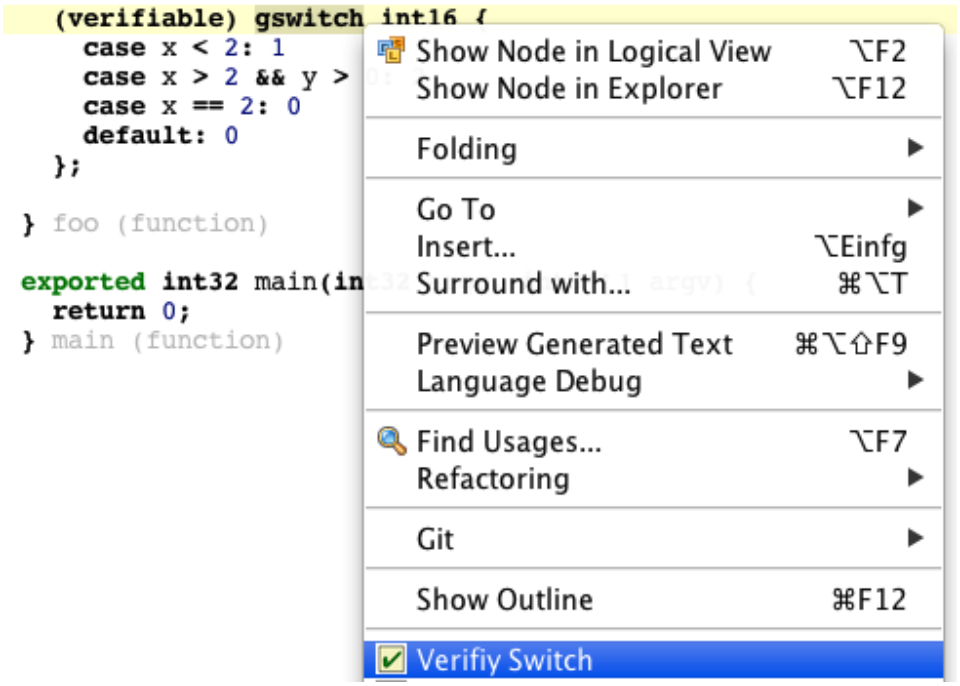


Above we had mentioned that the expressions in the decision table have to be "simple". More specifically, the expressions have to *linear*. If they are not, the table cannot be analyzed. A set of automatic checks are performed by MPS to assure that the conditions used in the top line and left column of the decision table meet the linearity limitations. If they are not, an error message is displayed with an explanation about the cause. In this case the completeness and consistency analyses cannot be run.

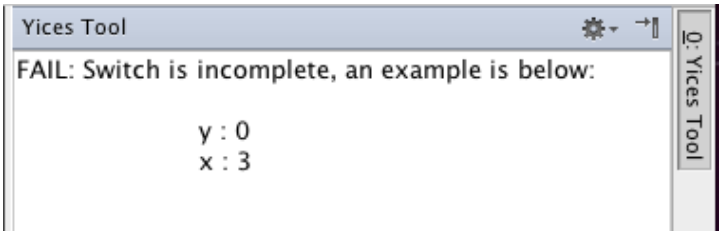


■ **gswitch** A `gswitch` (which is short for generic switch) is essentially a set of `if`-statements expressed with a more concise syntax. It can also be seen as a one-dimensional

decision table. Consequently, the same analyses are available: after marking a **gswitch** as **verifiable** (using an intention) the verification can be started:



The verification results concerning the completeness of the gswitch are shown in a window. The conditions used in the gswitch should have "the same form" as those used in the decision tables.



10.2 Model-Checking State Machines

10.2.1 Installation

State machines checking is based on the NuSMV symbolic model checker: `http://nusmv.fbk.eu/`. To be able to use these analyses, NuSMV has to be installed on your machine and available in the search path.

10.2.2 Available Analyses

mbeddr supports two kinds of model checking: automatic and custom. The automatic analyses are executed whenever any state machine is verified. The custom analyses have to be contributed manually by the user. mbeddr supports the following automatic analyses on state-machines:

Unreachable States checks whether all states can be reached somehow. Non-reachable states represent "dead-code" and are reported as errors.

Not-fireable Transitions checks whether all transitions can be fired. Transitions that cannot be fired represent dead code as well and are flagged.

Nondeterministic Transitions checks whether there are cases when more transitions can be fired simultaneously. Specifically, this means that several transitions react on the same event and have overlapping guard conditions.

Variable Bounds Integral variables in state machines are bounded explicitly. This verification checks whether all variables remain inside their defined bounds.

In addition to the above analyses, users can define other, application domain-oriented verification conditions by instantiating a well-known set of temporal logic patterns[1]. A special DSL that is easier to use than the underlying CTL specifications is available.

[1] A comprehensive catalogue of patters can be found here `http://patterns.projects.cis.ksu.edu/`

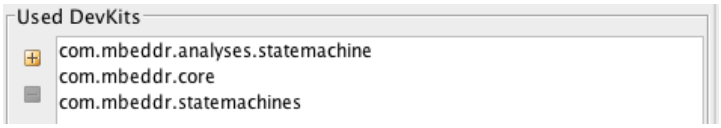
10.2.3 Performing Analyses

■ **Restrictions** We support the analysis of only a subset of the state-machines language provided by mbeddr. More precisely, we allow only the following features:

- **Data types:** all local variables, arguments of input or output events should have as type one of the following types: `enumeration`, `boolean`, `int8`, `int16`, `int32` and `bounded_int`. In particular, we do not support `floats` or `structs`.
- **No access to global state:** accessing global variables or calling global functions is not allowed. Mapping out-events to arbitrary functions is legal, though.
- **Single assignment actions:** in each effective action executed as a consequence of an event (i.e. exit action of the current state + transition action + entry action of the target state), a variable can be assigned only once.

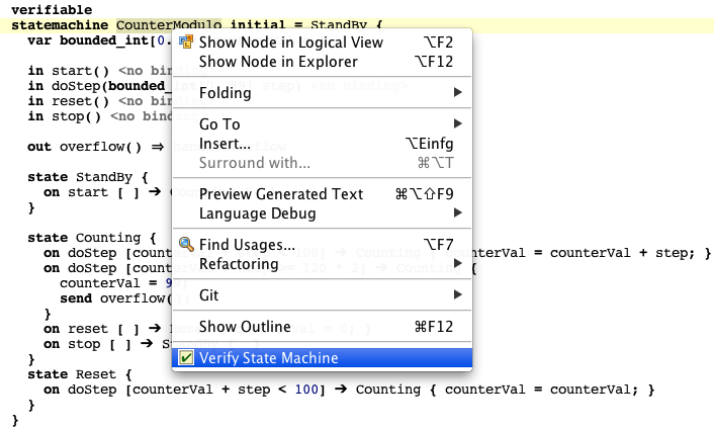
If a particular state machine does not conform to these restrictions and error is reported. No verification is possible in this case.

■ **Running the Verification** The first step for analyzing decision tables is to add the devkit `com.mbeddr.analyses.statemachine` to the model that contains the state machine that should be verified.

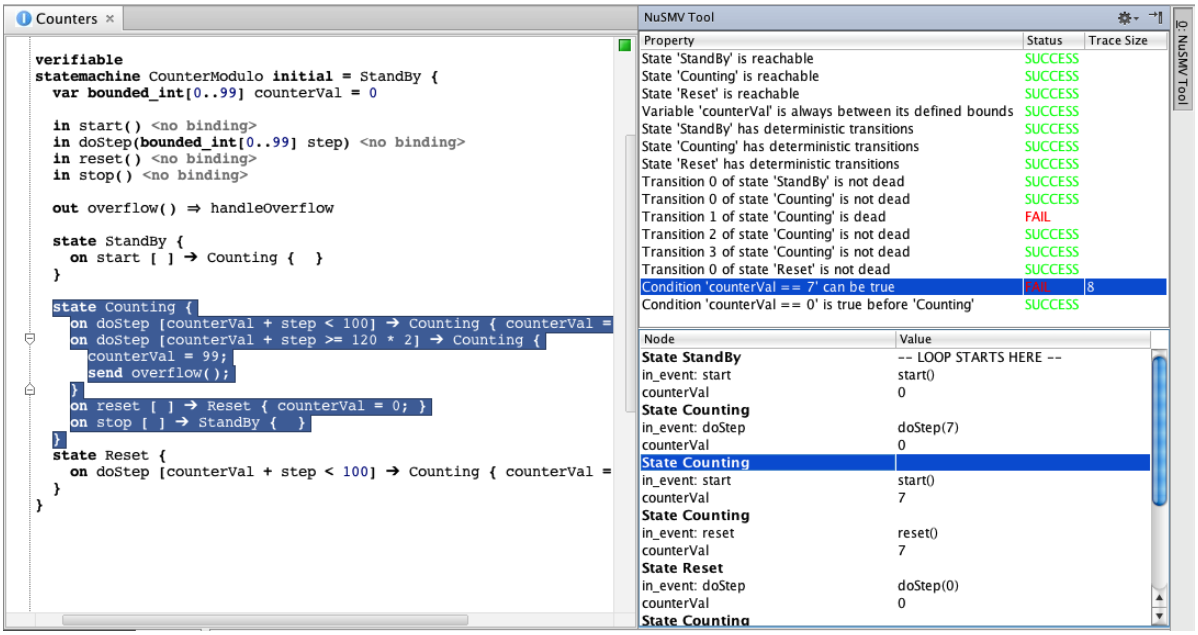


Once the devkit is added to the model, you can mark a state machine as `verifiable` by using an intention on the state machine. Once the `verifiable` flag is set it will be displayed on the top of the state-machine.

To start the verification, right-click on the state-machine node and select the `Verify State Machine` menu item. By default, only the automatic verifications will be executed. Adding custom conditions is described below.

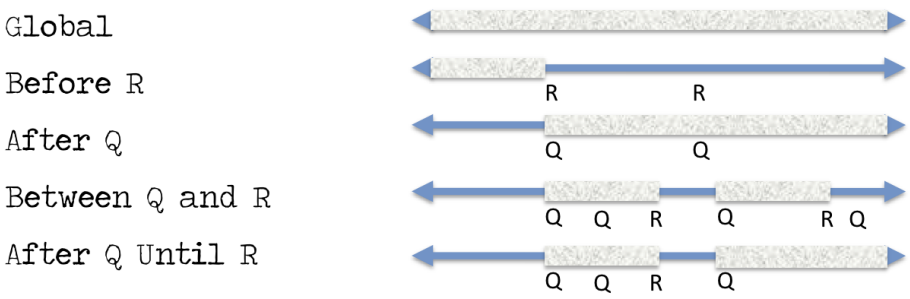


After starting the verify action, NuSMV code will be generated, then NuSMV will run (depending on the complexity of the state-machine, this might take a long time) and the results will be lifted back and displayed in a table in a separate window.

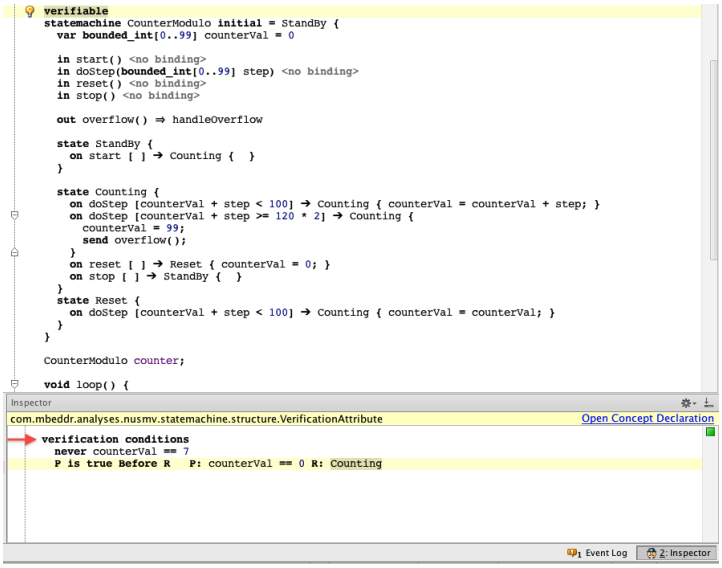


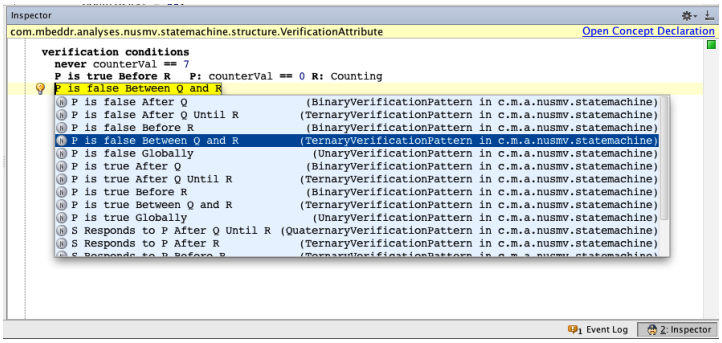
Successful verifications are marked with **Success**. Failed verifications are marked with **FAIL**. Clicking on a failed verification shows the counter example in the table below. The counter example represents an example execution of the state machine (i.e. sequence of steps) that leads to a violation of that particular verification condition. By clicking on a state in the counterexample table, the corresponding node will be highlighted in the editor.

■ **Custom Conditions** Custom defined verification conditions are defined by instantiating patterns in the Inspector view of the node **`verifiable`**. Each of these patterns has a temporal scope (i. e.,**`Global`**, **`Before`**, **`After`**, **`Between Q and R`**, **`After Q Until R`**) that restricts a certain basic property (i. e.,**`P`**, **`not P`**, **`S Responds to P`**). In the figure below we illustrate the temporal scope of the patterns.



To define a new verification condition, click on the **`verifiable`** node and in Inspector view instantiate a pattern.





10.3 Checking Interface Contracts and Protocols

10.3.1 Installation

Checking the contracts and protocols attached to component interfaces is based on the CBMC model checker: **`http://www.cprover.org/cbmc/`**. To be able to use these analyses, CBMC has to be installed on your machine and available in the search path.

10.3.2 Available Analyses

Given a component that provides interfaces that have pre- or postconditions or interface protocols, we check for each runnable in the component:

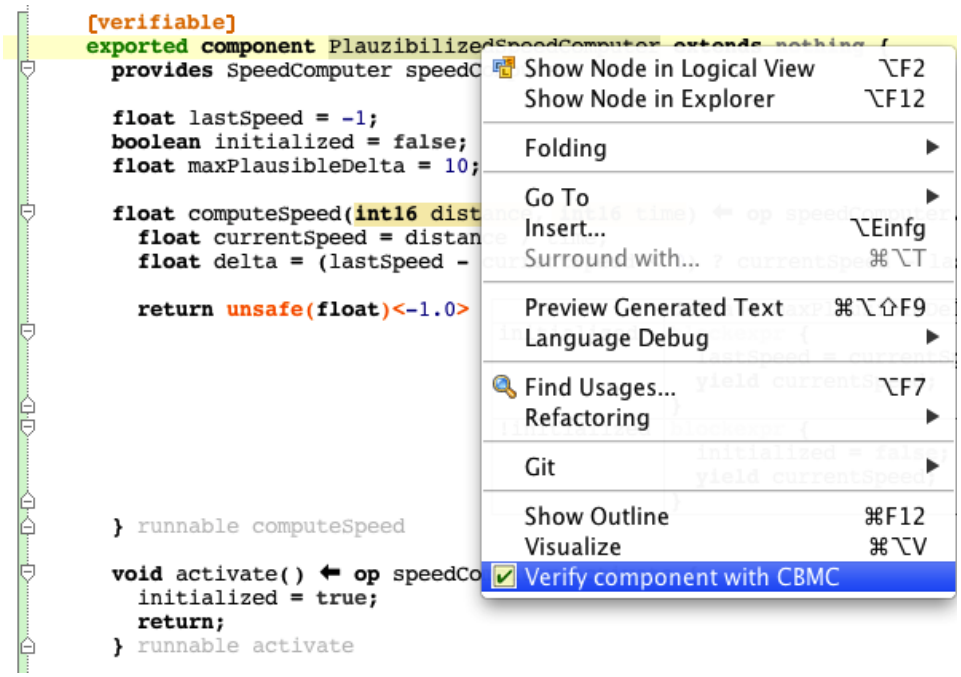
- that all preconditions are always met when the operations are called.
- that all postconditions hold when the runnable returns.
- that the clients of this component comply with the protocol expected by the interface.

10.3.3 Performing the Analyses

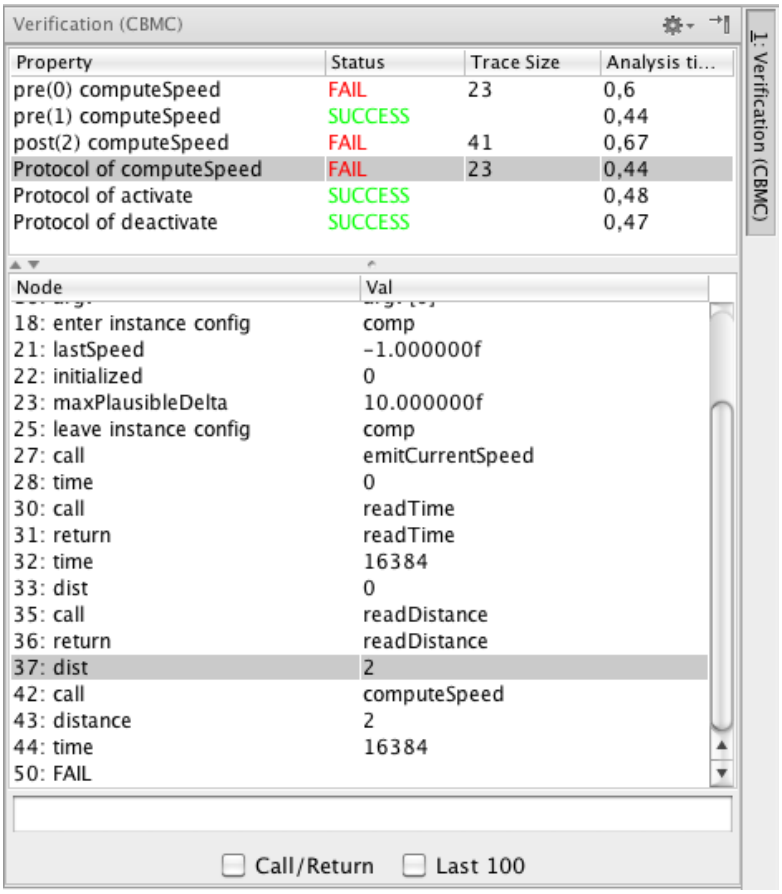
The first step for analyzing pre-/postconditions and interface protocols is to enable the checking of contracts (**`check contracts`** must be **`true`**) in the **`Build Configuration`**:

```
Configuration Items
  components {
    generation strategy: no middleware {
      wire statically: true instance config: comp
      check contracts: true
    }
  }
```

To start the verification, right-click on a component node and select the **Verify Component** menu item.



The verification result will be displayed in a window as shown below. On the upper part of the window we have the list with checked pre-/postconditions and protocols. In the case when an condition fails, if you double-click on it then a trace through the system that leads to the failure will be displayed in the lower part of the window. Often, traces are long and can be filtered by entering a text. By checking **Call/Return**, only the method calls and returns from the counterexample will be displayed. By selecting **Last 100** only the last 100 steps of the counterexample will be displayed.



10.3.4 Analyses Configuration

For each component one can configure how the analysis is performed by defining an analysis configuration. The configuration can be entered in the inspector for the `verifiable` annotation. The following parameters can be configured:

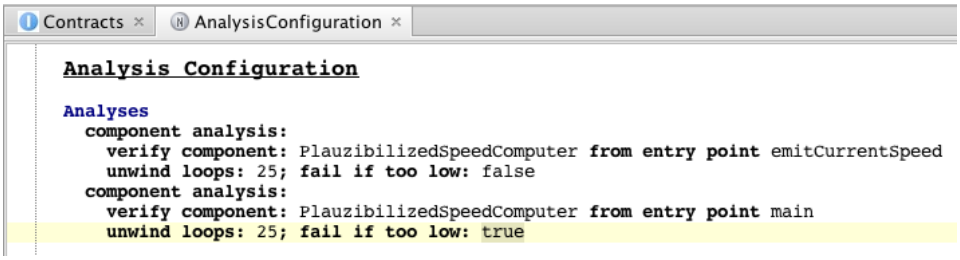
- **entry point** represents the function from where the analysis starts. By default, if none is given, the analysis starts from the `main` function. However, starting from `main` may make the verification take a long time, and a more local entry point may be advantageous.

- **loops unwinding** is the number of times the loops are unwound (unfolded)

- **unwinding assertions** when set to true, then the analysis fails if the number of unfolds is insufficient for analyzing the whole loop. If set to false, then the analysis

is performed but potentially incomplete.

In addition to specifying the configuration in the inspector of the **verifiable** flag, it can also be added as a configuration item in the build configuration. By right-clicking on the **Analysis Configuration** node itself, all analyses are performed.



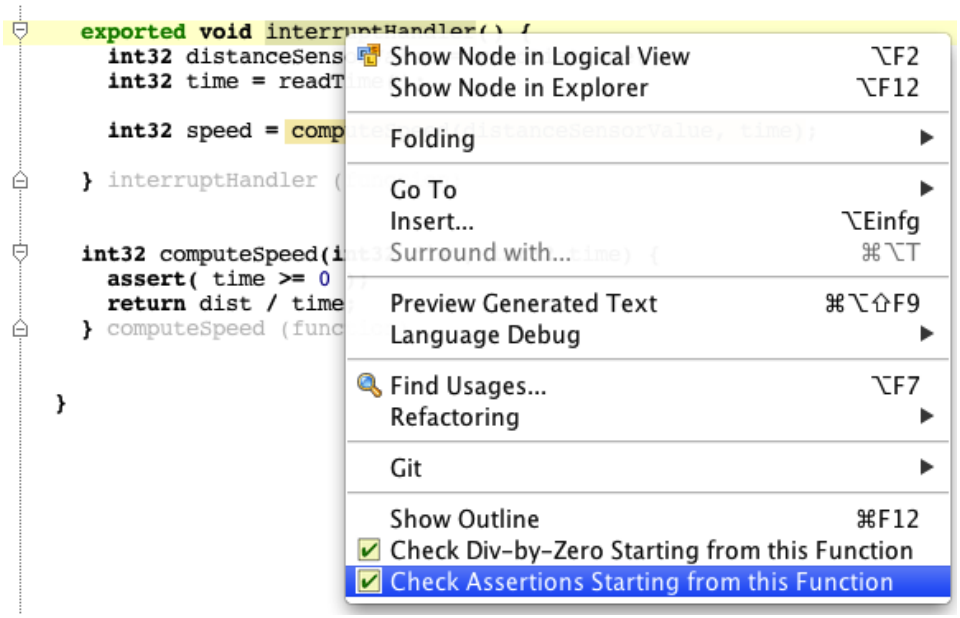
10.4 Checking Assertions and Division-by-Zero

10.4.1 Installation

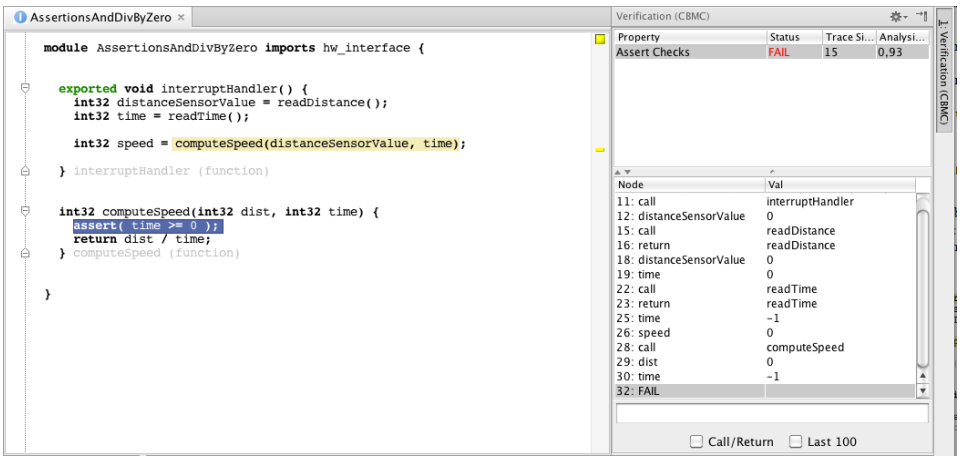
Checking assertions and the potential occurrence of division-by-zero can be done by using the CBMC model checker. Its installation has been described in the previous section (Section 10.3).

10.4.2 Performing the Analyses

To start the assertion or division-by-zero checking, right-click on a function node and select the **Check assertions starting from this function** or **Check division-by-zero starting from this function** menu item. Please make sure that the implementation module which contain the functions on which the analyses are run is contained in the **Build Configuration**.



Below is illustrated the result of checking assertions starting from a function. In the case when the assertion fails, a counterexample is shown that guides you through the system towards a state which violates the assertion. By clicking on different entries of the counterexample, the corresponding piece of the program is opened and highlighted in the editor.



10.5 Checking Product Line Variability

10.5.1 Installation

Checking the consistency of feature models and configurations is based on the Yices SMT solver[2]:
http://yices.csl.sri.com/download.shtml

To be able to use these analyses, the solver has to be installed on your machine and available in the search path.

10.5.2 Available Analyses

For feature models we can check if the cross-cutting constraints (**conflicts with**, **requires also**) are consistent or not. If they are not consistent, then the feature model cannot be instantiated (i. e.,no valid configuration is possible). For configuration models we check if they conform to the defined feature model.

Note: A number of additional verifications for feature models seem useful, such as: checking whether a presence condition is possible, checking whether an artifact remains consistent for every defined configuration, etc. Over time, we will add more verifications.

10.5.3 Performing the Analyses

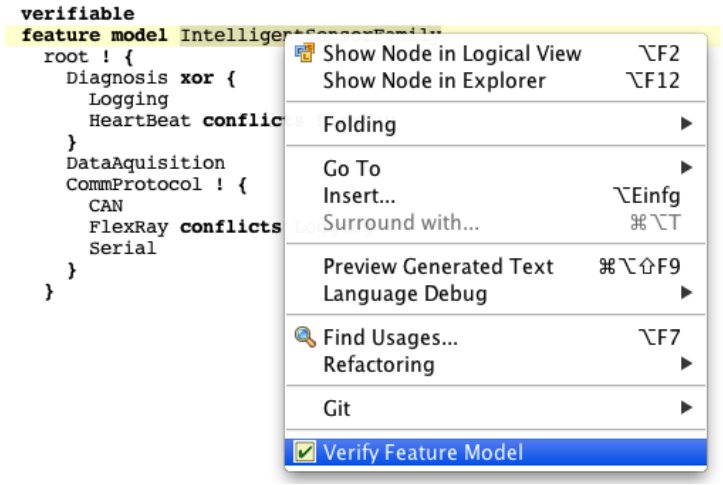
The first step for analyzing feature models is to add the devkit **com.mbeddr.analyses.fm** to the model that contains the to-be-verified feature model:

■ **Feature Models** Once the devkit is added to the model, you can mark a feature model or a configuration model as **verifiable** by using an intention. Once the

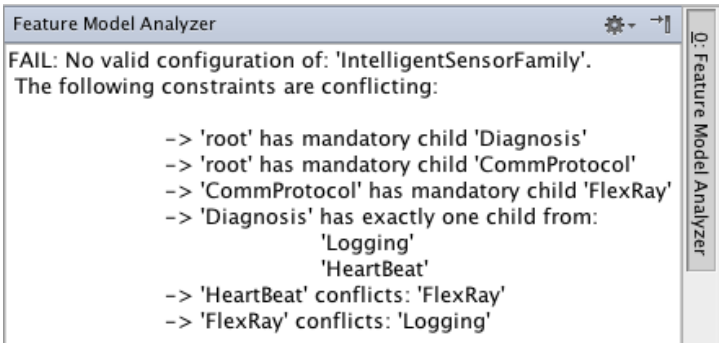
[2]This is the same tool that is also used for checking decision tables.

verifiable flag is set it will be displayed on the top of the feature model or configuration model.

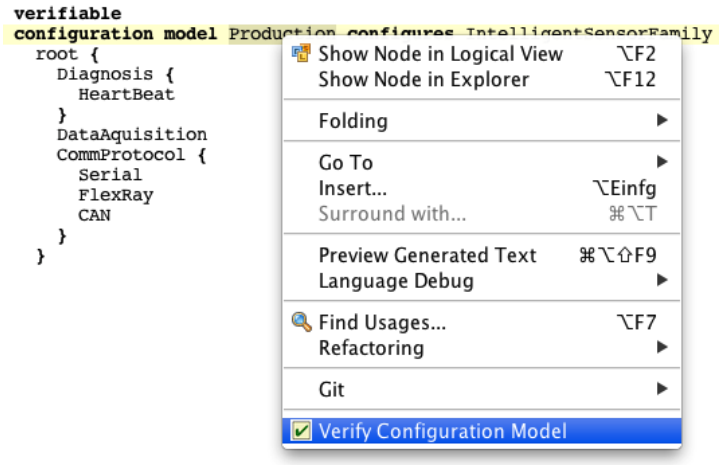
To start the verification, right-click on the feature model node and select the **Verify Feature Model** menu item.



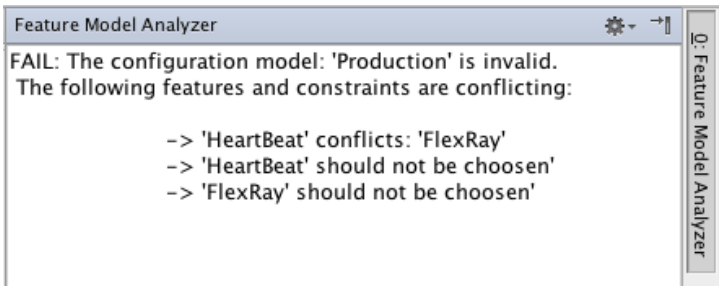
The verification result will be displayed in a window as shown below. In case the feature model is inconsistent (since it contains conflicting constraints), those conflicting constraints are listed.



■ **Configuration Models** To start the verification, right-click on the configuration model node and select the **Verify Configuration Model** menu item.



The result of the verification is displayed in a window as a debug information about which of the features selected in the configuration are conflicting with the constraints defined in the feature model.



Bibliography

[1] S. Andalam, P. Roop, A. Girault, and C. Traulsen. PRET-C: A new language for programming precision timed architectures. In *Proocedings of the Workshop on Reconciling Performace with Predictability (RePP), Embedded Systems Week*, 2009.

[2] E. Axelsson, K. Claessen, G. Devai, Z. Horvath, K. Keijzer, B. Lyckegard, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *MEMOCODE 2010*.

[3] Y. Ben-Asher, D. G. Feitelson, and L. Rudolph. ParC - An Extension of C for Shared Memory Parallel Processing. *Software: Practice and Experience*, 26(5), 1996.

[4] F. Boussinot. Reactive C: An Extension of C to Program Reactive Systems. *Software: Practice and Experience*, 21(4), 1991.

[5] M. Broy, S. Kirstan, H. Krcmar, and B. Schätz. What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry? In *Emerging Technologies for the Evolution and Maintenance of Software Models*. ICI, 2011.

[6] E. Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *LNCS*, pages 54–56. 1997.

[7] W. Damm, R. Achatz, K. Beetz, M. Broy, H. Dämbkes, K. Grimm, and P. Liggesmeyer. *Nationale Roadmap Embedded Systems*. Springer, Mar. 2010.

[8] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys 2006. ACM.

[9] C. Ebert and C. Jones. Embedded Software: Facts, Figures, and Future. *Computer*, 42(4), april 2009.

[10] A. S. Gokhale, K. Balasubramanian, A. S. Krishna, J. Balasubramanian, G. Ed-

wards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt. Model driven middleware: A new paradigm for developing distributed real-time and embedded systems. *Science of Computer Programming*, 73(1), 2008.

[11] B. Graaf, M. Lormans, and H. Toetenel. Embedded Software Engineering: The State of the Practice. *IEEE Softw.*, 20(6), Nov. 2003.

[12] K. Hammond and G. Michaelson. Hume: a domain-specific language for real-time embedded systems. In *Proceedings of GPCE 2003*.

[13] P. Liggesmeyer and M. Trapp. Trends in Embedded Software Engineering. *IEEE Softw.*, 26, May 2009.

[14] K. Loer and M. Harrison. Towards Usable and Relevant Model Checking Techniques for the Analysis of Dependable Interactive Systems. In *In Proceedings of the International Conference on Automatic Software Engineering (ASE)*, 2002.

[15] L. Palopoli, P. Ancilotti, and G. C. Buttazzo. A C Language Extension for Programming Real-Time Applications. In *6th International Workshop on Real-Time Computing and Applications (RTCSA 99)*. IEEE CS.

[16] A. G. S. Andalam, P. S. Roop. Predictable multithreading of embedded applications using PRET-C. In *Proceedings of ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE),*, 2010.

[17] H. Sheini and K. Sakallah. From Propositional Satisfiability to Satisfiability Modulo Theories. In *Theory and Applications of Satisfiability Testing - SAT 2006*, volume 4121 of *LNCS*, pages 1–9. Springer Berlin / Heidelberg, 2006.

[18] R. von Hanxleden. SyncCharts in C - A Proposal for Light-Weight, Deterministic Concurrency. In *Proceedings of the International Conference on Embedded Sofware (EMSOFT'09)*, 2009.