

# mbeddr C Cross Cuttings User's Guide

Markus Voelter

independent/itemis

**Abstract.** In this document we describe how to use mbeddr's support for cross-cutting concerns. This includes requirements and requirements traceability and support for product line variability.



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

This document is licensed under the  
Eclipse Public License 1.0 (EPL).

# Table of Contents

mbeddr C Cross Cuttings User's Guide .....	1
<i>Markus Voelter</i>	
1 Requirements .....	3
1.1 Overview .....	3
1.2 Specifying Requirements .....	3
1.3 Tracing .....	6
1.4 Other Traceables .....	6
1.5 Evaluating the Traces in Reverse .....	7
2 Variability .....	7
2.1 Overview .....	7
2.2 Feature Models and Configurations .....	8
2.3 Presence Conditions .....	9
2.4 Replacements .....	10
2.5 Attribute Injection .....	11
2.6 Projection Magic .....	11
2.7 Building Variable Systems .....	11

# 1 Requirements

## 1.1 Overview

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

## 1.2 Specifying Requirements

Requirements can be collected in instances of `RequirementsModule`, a root concept defined by the `com.mbeddr.cc.requirements` language. An example is shown in Fig. 1.2. Each requirement has an ID, a short prose summary, and a kind. (`functional`, `timing`). The kind, however, is more than just a text; each kind comes with its own additional specifications. For example, a `timing` requirement requires users to enter a `timing` specification. This way, additional formal data can be collected for each kind of requirement.

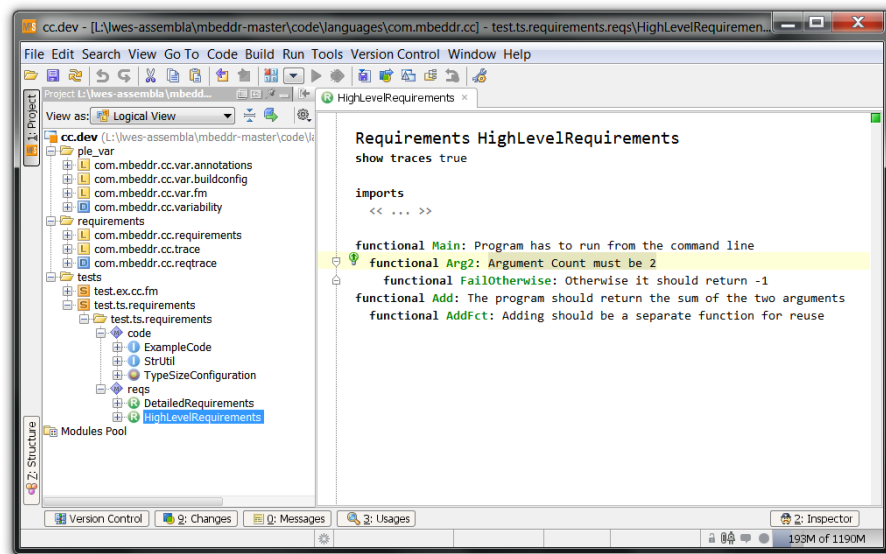


Fig. 1. Working with Requirements

In addition to the summary information discussed above, requirements can also contain details. The details editor can be opened on a requirement with an intention or with `Ctrl-Shift-D`. Fig. 1.2 shows an example.

In the details, a requirement can be described with additional prose, with the kind-specific formal descriptions as well as with additional constraints among

```
functional Main: Program has to run from the command line
functional Arg2: Argument Count must be 2
functional FailOtherwise: Otherwise it should return -1
functional Add: The program should return the sum of the two arguments
functional AddFct: Adding should be a separate function for reuse
```



```
functional Main: Program has to run from the command line
functional Arg2: Argument Count must be 2
```

Additional Constraints

none

Additional Specifications

none

Description

Some details about this requirement.  
Several lines.

Close

```
functional FailOtherwise: Otherwise it should return -1
functional Add: The program should return the sum of the two arguments
functional AddFct: Adding should be a separate function for reuse
```

**Fig. 2.** Requirements editor expanded

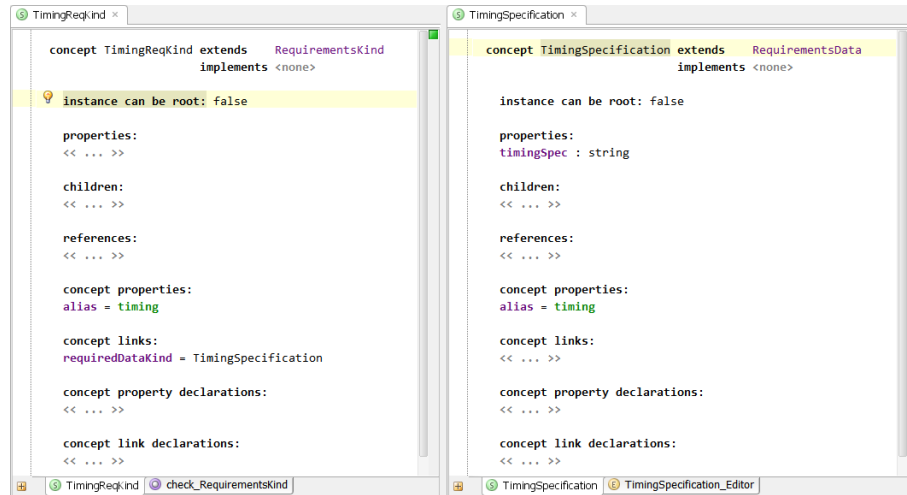
requirements. The default hierarchical structure represents refinement: child requirements refine the parent requirements. In addition, each requirement can have typed links relative to other requirement, such as the **conflicts with** shown in Fig. 1.2.

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

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

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

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



**Fig. 3.** Defining additional required specifications for a requirements kind.

*A new Additional Specification* Defining new additional specifications (such as the `timing` specification mentioned above) happens in two steps. First you have to create a new concept that extends `RequirementsData`. It should contain any additional structure you need (this can be a complete MPS DSL, or just a set of pointers to other nodes). You should also define an alias. The second step requires enforcing that a certain requirements kind also requires that particular additional specification. In the respective kind, use the `requiredDataKind` concept link to point to the concept whose instance is required. Fig. 1.2 shows an example.

```
requirements modules: HighLevelRequirements
module ExampleCode from test.ts.requirements.code imports nothing {

  int8_t main(string[ ] args, int8_t argc) { trace Requirement2
    if ( argc == 2 ) {
      return 0;
    } else {
      return 1;
    } if
  } main (function)
```

**Fig. 4.** Code with requirements traces

### 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. Fig. 1.2 shows an example of requirements traces.

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

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

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

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

### 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.

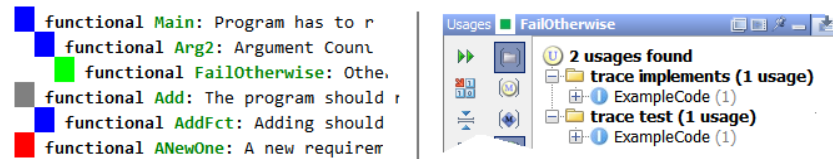
- Concepts that should act as a trace target must implement the `ITraceTarget` interface (e.g. `Requirement`)
- The root concept that contains the `ITraceTargets` must implement the `ITraceTargetProvider` and implement the method `allTraceTargets`.
- In addition, you have to create a concept that extends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

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

## 1.5 Evaluating the Traces in Reverse

The traces can be evaluated in reverse order. For example, Fig. 1.5 (left) shows how requirements can be color-coded to reflect their state. The color codes must be updated explicitly (may take a while) by the **Update Trace Stats** intention on the requirement module.

In addition, MPS Find Usages functionality has been enhanced for requirements. If the user executes Find Usages for requirements, the various kinds of traces are listed separately in the result (Fig. 1.5, right).



**Fig. 5. Left:** The requirements can be color coded to reflect whether they are traced at all (grey), implemented (blue) and testes (green). Untraced requirements are red. **Right:** The Find Usages dialog shows the different kinds of traces as separate categories.

## 2 Variability

### 2.1 Overview

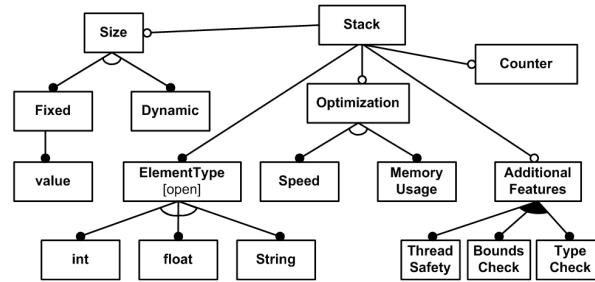
Product line engineering involves the coordinated construction of several related, but different products. Each product is typically referred to as a *variant*. The product variants within a product line have a lot in common, but also exhibit a set of well-defined differences. Managing these differences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

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

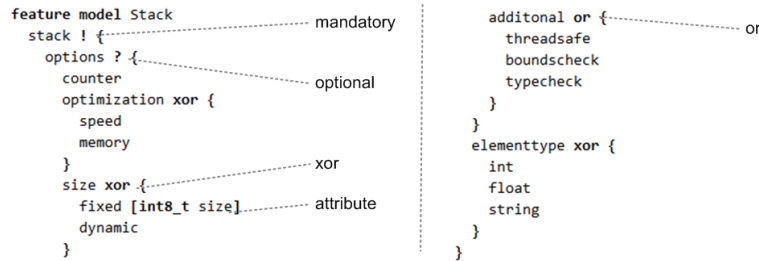
- `com.mbeddr.cc.var.fm` supports the definition of feature models. Feature models are a well-known formalism for expressing variability on a high-level, independent of the realization of the variability in software.
- `com.mbeddr.cc.var.annotations` allows the connection of implementation artifacts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

## 2.2 Feature Models and Configurations

**Defining a Feature Model** Feature models express configuration options and the constraints between them. They are usually represented with a graphical notation as shown in Fig. 2.2.



**Fig. 6.** An example feature model



**Fig. 7.** An example feature model in mbeddr

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

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



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

Fig. 2.2 shows the textual notation for feature models used in mbeddr. Note how the constraint affects all children! We had to introduce the intermediate feature **options** to separate the mandatory stuff from the optional stuff. Features can have configuration attributes (of any type!). Children and attributes can be added to a feature via an intention. You can also use a surround intention to wrap a new feature around an existing one.

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

Let us now look at how to define a product variant as a set of selected features. Fig. 2.2 shows two examples:

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

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

**Fig. 8.** Two valid configurations of the feature model

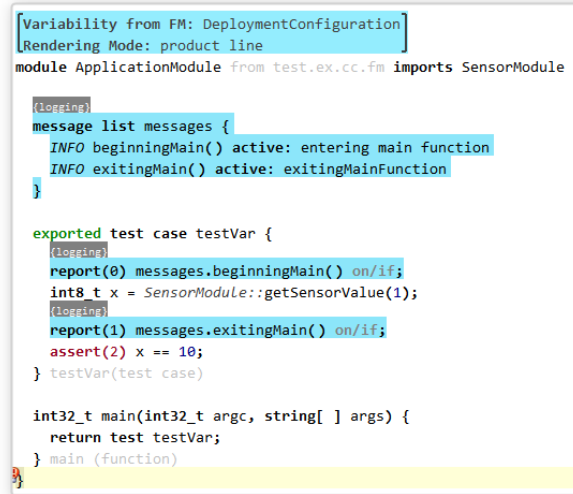
Note that, if you create invalid configurations by selecting feature combinations that are prohibited by the constraints expressed in the feature model, errors will be shown.

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

### 2.3 Presence Conditions

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

For example the two `report` statements and the message list in Fig. 2.3 are only in the program, if the `logging` feature is selected. `logging` is a feature defined in the feature model FM that is referenced by this root node.



```

[Variability from FM: DeploymentConfiguration]
[Rendering Mode: product line]
module ApplicationModule from test.ex.cc.fm imports SensorModule {

  (logging)
  message list messages {
    INFO beginningMain() active: entering main function
    INFO exitingMain() active: exitingMainFunction
  }

  exported test case testVar {
    (logging)
    report(0) messages.beginningMain() on/if;
    int8_t x = SensorModule::getSensorValue(1);
    (logging)
    report(1) messages.exitingMain() on/if;
    assert(2) x == 10;
  } testVar(test case)

  int32_t main(int32_t argc, string[ ] args) {
    return test testVar;
  } main (function)
}

```

Fig. 9. C program code with presence conditions

To use presence conditions, the root node (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 node 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).

## 2.4 Replacements

A presence condition is basically like an `#ifdef`: the node to which it is attached will *not* be in the resulting system if the presence condition is *false*. But sometimes you want to *replace* something with something else if a certain feature condition is met. You can use replacements for that. Fig. 2.4 shows an example.

A replacement replaces the node to which it is attached with an alternative node if the condition is *true*. In the example in Fig. 2.4 the function call and the 10 are both replaced with a 42. Note that you'll get an error if you try to replace a node with something that is not structurally compatible, or has the wrong type.

```

exported test case testVar {
  (logging)
  report(0) messages.beginningMain() on/if;
  int8_t x = SensorModule::getSensorValue(1) replace if (test) with 42;

  (logging)
  report(1) messages.exitingMain() on/if;
  assert(2) x == 10 replace if (test) with 42;
}

```

Fig. 10. C program code with a conditional replacement

## 2.5 Attribute Injection

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

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

```

(valueTest)
int8_t vv = value;
(valueTest)
assert(3) vv == 42;

int8_t ww = 22 replace if (valueTest) with 12 + value;
(!valueTest)
assert(4) ww == 22;

```

Fig. 11. C program code with an attribute injection (`value` is the name of an attribute of the `valueTest` feature)

## 2.6 Projection Magic

It is possible to show and edit the program as a product line (with the annotations), undecorated (with no annotations) as well as in a given variant. Fig. 2.6 shows an example. Note that due to a limitation in MPS, it is currently not possible to show the values of attributes directly in the program in variant mode. The projection mode can be changed in the configuration annotation on the root node.

## 2.7 Building Variable Systems

Building variable systems is a little bit tricky. The problem is that you will want to build different variants at the same time. To make the C build simple, each



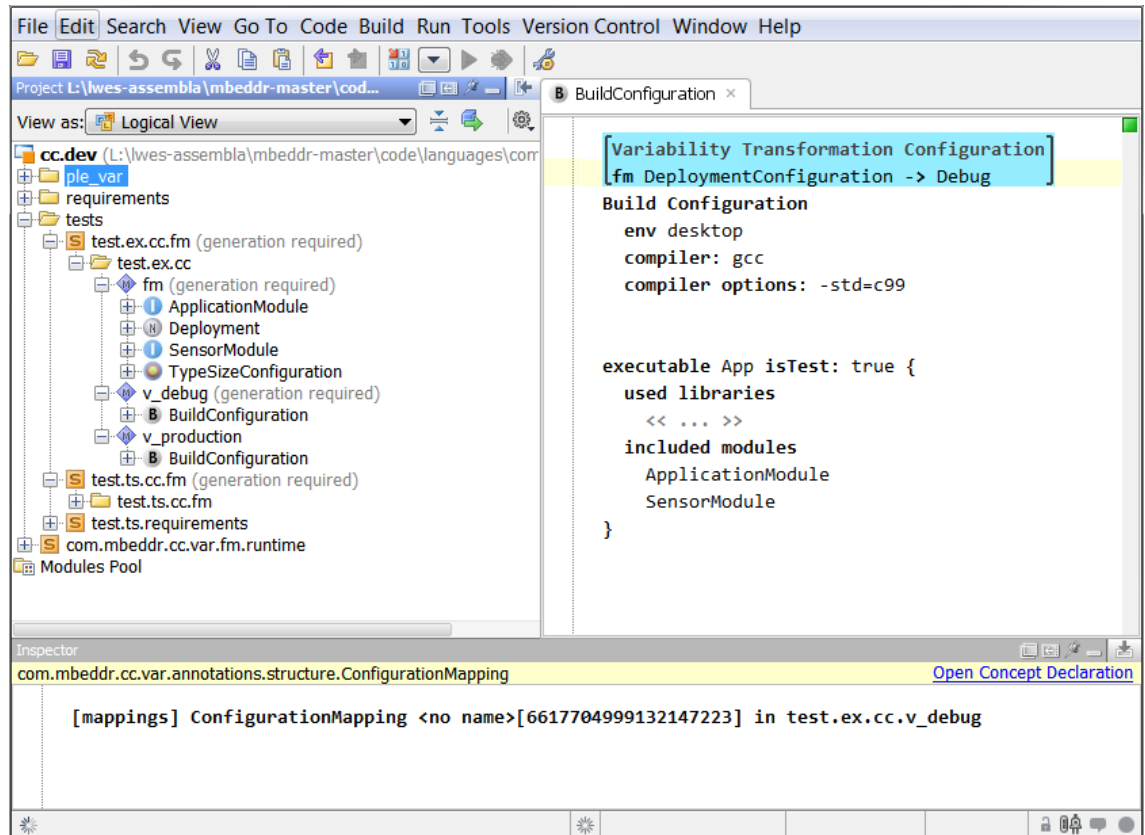
Fig. 12. C program rendered in product line mode and in two variants

variant should live in its own directory. If you want to generate into different directories with MPS, you need different models. This results in the following setup:

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

Figure Fig. 2.7 shows an example of such as setup with one product line model (fm) and two variant models v\_debug and v\_production.

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



**Fig. 13.** Model setup and transformation configuration for generating several variants at the same time.