Expressing Scientific Calculations in MPS

Version 1.3, Sept 8, 2010

Markus Völter
(voelter@acm.org)

Introduction

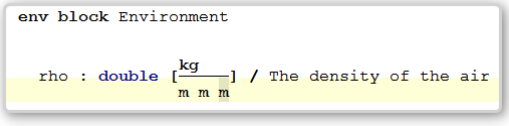
In many applications, scientific or mathematical computations are mixed with programming language code. Typically, the calculations are implemented as functions/methods, with no support for mathematical notations and physical units. Also, because the mathematical formulae are mixed implemented by means of programming languages, they are hard to maintain by non-programmers (i.e. mathematicians or scientists). Also, testing is not very well supported. Finally, the mix of programming and scientific concerns limits maintainability in general.

In this paper I describe a prototype implementations of a science/mathematics DSL based on MPS.

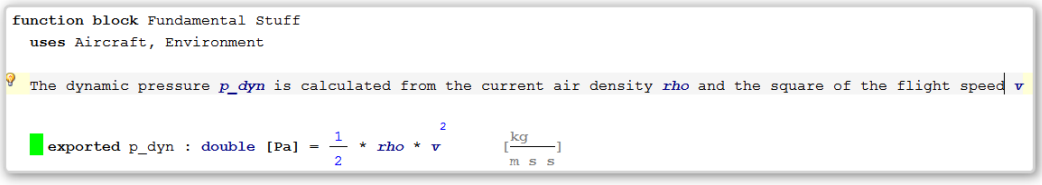
Overall program structure

The mathematical/scientific (MS) and software concerns are separated and integrated where necessary. I first describe the MS concerns.

On top level, MS programs consist of blocks. Two kinds of blocks exist: environment blocks and function blocks. The former declare quantities that are supplied from the outside. The latter declare formulae to calculate new quantities. Here is a simple example of an environment block which declares the air density as a quantity.



A function block calculates new quantities:



Using the *uses* clause, blocks can specify dependencies among each other. Only quantities defined in the used blocks (and the current block of course) are visible and can be used.

> Later Versions will be able to show a dependency graph between the blocks.

Let us look at each of these blocks in detail.

Environment Blocks

An environment block declares quantities that are input variables during the calculations

> Different means of actually obtaining these variables will be added later.

Each physical quantity has a name, a data type, a physical unit as well as a simple documentation comment.

> Later versions will support the use of subscript and superscript in symbol names, as in c_a or c_w.
>
> Data types are currently Java data types. It might be better to make these programming language-independent.

```
env block Aircraft

  v       : double [m/s ] / current aircraft speed

  A       : double [m m] / cross area of the wing

  c_a     : double [1   ] / Auftriebsbeiwert

  c_w     : double [1   ] / Widerstandsbeiwert

  n_wings : double [1   ] / Number of Wings
```

The physical units are based on SI units. It is possible to define and use derived units such as *Pa* which is N/m^2, where *N* in turn is $kg\ m/s^2$.

Function Blocks

Definition of Functions

All the interesting things are going on in function blocks. Function blocks can contain the following basic ingredients:

```
function block Fundamental Stuff
  uses Aircraft, Environment

  The dynamic pressure p_dyn is calculated from the current air density rho and the square of the flight speed v

  exported p_dyn : double [Pa] = 1/2 * rho * v^2        [kg/(m s s)]
```

Dependencies on other blocks. They determine which quantities are visible and can be used. When importing an environment block, all quantities defined in it are available. When importing another function block, only those functions marked as *exported* are visible.

Documentation Text. Documentation text is basically normal text; however, the references to symbols (e.g. the *p_dyn*, *rho* or *v* in the above screenshot) are actually real references; code completion is provided when entering a reference, and ctrl-click can be used to go to the definition of the reference.

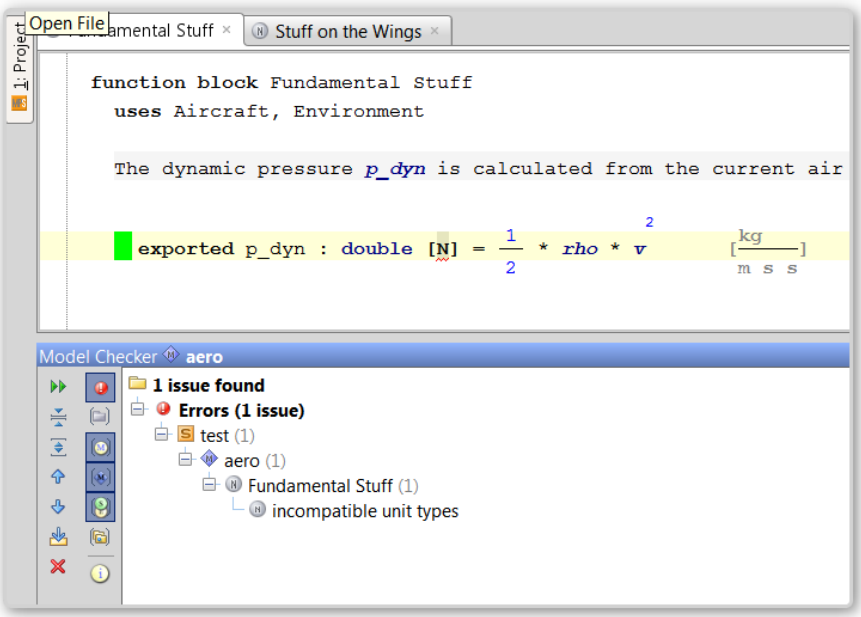
> In a later version, much more sophisticated facilities for entering text and integrating it with formulas will become available.

Functions. The functions themselves also have a name, a Java data type, and a physical quantity. They can optionally be exported to make the symbols visible in referencing blocks. Formulas also contain the actual calculation; notice now mathematical symbols can be used here (more examples below).

> More mathematical symbols (and better layout) will be added later.

Unit Checking

Functions declare an expected unit (*Pa* in the example below). The system calculates the actual unit of the formula (at the end of the line, in grey), and if the two don't match, an error is reported:



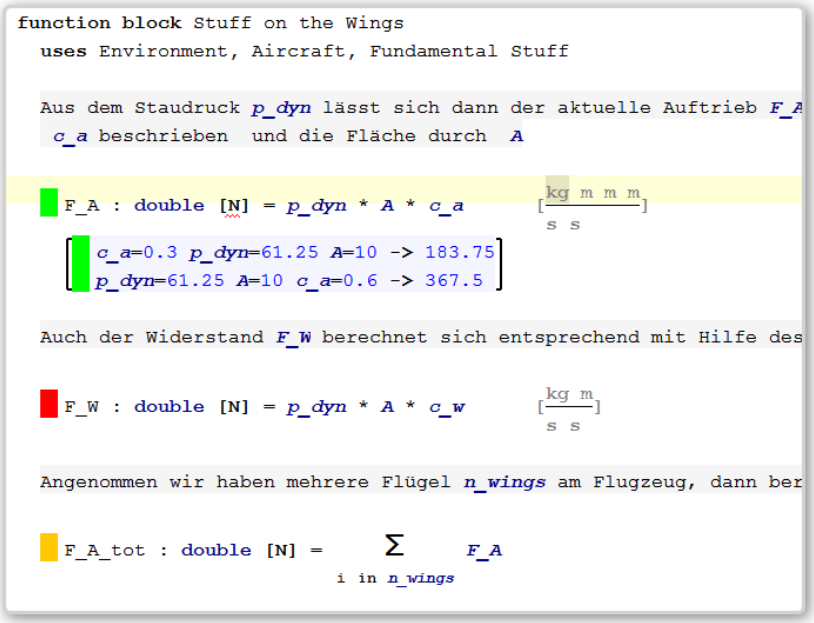
Testing

Testing of the physical formulas is important - in the editor itself, as well as in the generated code. It is possible to associate tests with any formula. As can be seen from the screenshot below, the tests can be hidden or shown.

Each test is evaluated in real time; a green bar showing that it succeeded, a red bar shows an error (Note that the tests are actually evaluated by an interpreter directly in the tool - no code generation required!)

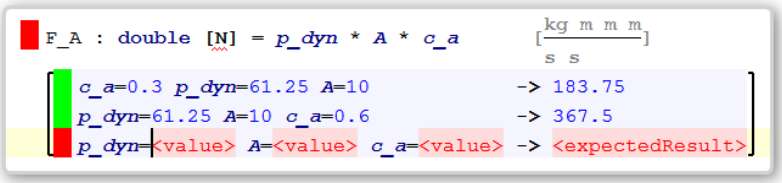
If the tests themselves are hidden, the color code is aggregated onto the formula itself: If one or more tests fail, the formula is red. If no test is defined, the formula is marked orange.

(Notice also the sum symbol, as an example of more mathematical symbols that can be used).



If you add a new test, the system automatically provides inputs for values of all the referenced symbols.

In later version, a nicely formatted table will be available for entering test values.



Integration with Java Code

Using the formulas from application code

The formulas defined above are directly integratable into Java code. Let's look at calculating the lift of an airplane:

```
import blocks Environment
              Aircraft
              Stuff on the Wings
public class TestClass extends <none> implements <none> {
  <<static fields>>

  <<static initializer>>
  <<fields>>
  <<properties>>
  <<initializer>>
  public TestClass() {
    <no statements>
  }

  public void calcLift() {
    values air = (| Environment.rho = 1.225 |);
    values planeStatic = (| Aircraft.A = 10.0, Aircraft.c_a = 0.5, Aircraft.c_w = 0.3, Aircraft.v = 100.0 |);
    double lift = Stuff on the Wings.F_A (air, planeStatic);
    System.err.println(lift);
  }

  public static void main(string[] args) {
    new TestClass().calcLift();
  }
```

At the top of this (otherwise ordinary) class we have added a so-called block import annotation (available as an intention in the Alt-Enter menu). You can then use Ctrl-Space to select the blocks whose symbols you want visible in the Java code.

You can then define *values* variables. A variable of type *values* can hold any number of value assignments to any of the environment symbols in the imported blocks. You can create any number of *values* and each can hold any combination of environment symbol value assignments. This way, different sets of input values can be stored.

When calling a function (again, available via Ctrl-Space based on the imported blocks) you can pass in any number of *values* objects to supply the input.

Note how the new language constructs are fully integrated with Java; they can be used with any Java expression (as in the assignment of *Aircraft.v*) in the following code.

```
public void calcLift() {
  values air = (| Environment.rho = 1.225 |);
  values planeStatic = (| Aircraft.A = 10.0, Aircraft.c_a = 0.5, Aircraft.c_w = 0.3 |);
  for (int i = 0; i < 100; i++) {
    values planeDynamic = (| Aircraft.v = (double) i |);
    double lift = Stuff on the Wings.F_A (air, planeStatic, planeDynamic);
    System.err.println(i + "->" + lift);
  }
}
```

The system also performs static analysis to find out if all required values are supplied, as shown in the following example, where the value for the aircraft speed is missing:

```
public void calcLift() {
  values air = (| Environment.rho = 1.225 |)
  values planeStatic = (| Aircraft.A = 10.0, Aircraft.c_a = 0.5, Aircraft.c_w = 0.3 |)
  for (int i = 0; i < 100; i++) {
    values planeDynamic = (| Aircraft.v = (double) i |)
    double lift = Stuff on the Wings.F_A (air, planeStatic);
    System.err.prin Error: no value provided for symbol Aircraft.v
  }
}
```

Of course, code generators to basic Java exist, and the programs can be executed.

Running the Unit Tests

The unit tests defined in the function blocks are executed directly by an interpreter in the tool, coloring the functions and the tests respectively. However, tests that pass in the interpreter should also pass in the compiled version.

To make the tests possible in the compiled version, the generated function modules contain the test code as well. You can run the tests the following way:

```
public static void main(string[] args) {
  test Fundamental Stuff;
  test Stuff on the Wings;
  new TestClass().calcLift();
}
```

If one fails you'll get an error message as follows:

```
tests failed for block Fundamental Stuff
sr.functionblocks.rt.main.TestFailedException: test for function p_dyn expected: 10.0, actual: 0.0
  at aero.FundamentalStuff_Implementation.test_p_dyn(FundamentalStuff_Implementation.java:26)
  at aero.FundamentalStuff_Implementation.runTests(FundamentalStuff_Implementation.java:16)
  at aero.TestClass.main(TestClass.java:25)
```