Embedded Software Development with Projectional Language Workbenches

Markus Voelter
independent/itemis
Ötztaler Strasse 38
70327 Stuttgart
email: voelter@acm.org
http://www.voelter.de

The Limitations of C and Modeling Tools

Embedded software is usually implemented in C. The language is relatively close to the hardware, allowing efficient machine code. A large number of compilers exist for many different target platforms and (embedded) operating systems. However, using C also its problems. For example, it has a number of features that are considered unsafe, such as void pointers, unions and (certain) automatic type conversions. Coding conventions, such as Misra-C [1], prohibit the use of these features. Another problem with C is its limited support for meaningful abstraction. For example, there is no first class support for interfaces or components, making modularization and replacement of functionality hard. This becomes a significant problem as systems become larger and more complex. Finally, the preprocessor can be used for any number of mischief, thwarting attempts to analyze source code in a meaningful way.

Modeling tools are used to address these problems. However, these also create problems, specifically regarding semantic integration among models and among models and code, as well as "mechanical" tool integration issues. Also, most tools do not provide first-class support for expressing variations in models or programs (think: product lines). If they do, each tool has its own paradigm. In C, *#ifdefs* are often used. Since variability affects all artifacts in the software lifecycle, handling it differently in each tool is a problem. As the number of variants increases all the time, a scalable and well integrated facility for treating variability is crucial. This is true in a similar way for requirements traceability.

Extensible Languages, Projectional Editing and MPS

This article illustrates an alternative approach for developing embedded software based on a modular, incremental extension of C. Extensions can be at arbitrary abstraction levels, integrating modeling and programming, and avoiding tool integration issues. Code generation to C is used to integrate with existing compilers and platforms. To keep the language manageable, extensions (such as those illustrated below) are separated into language modules. Modules can be reused from a library or can be developed specifically for a project or platform. Projectional language

workbenches, such as the open source JetBrains MPS, form the basis for the approach. Projectional editing implies that all text, symbols, and graphics are projected, just like in graphical modeling tools. The model is stored structurally, often using XML. For editing purposes the structural information is projected suitable notations. Users perform mouse gestures and keyboard actions tailored to the notation to modify the abstract model structure directly. Projectional editing can also be used for a syntax that is textual or semi-graphical (mathematical notations for example). However, since the projection looks like text, users expect interaction patterns and gestures known from "real text" to work (such as cursor movements, inserting/deleting characters, rearranging text, selection). A projectional editor has to "simulate" these interaction patterns to be usable. MPS does this very well.

Projectional editing has many advantages: Because editing changes the tree directly and no parsing is used, projectional editors can handle unparseable code. Language composition is easily possible, because composed languages cannot result in ambiguous grammars, a significant issue in classical parser-based systems. Graphical, semi-graphical and textual notations can be mixed and combined. Because projectional languages by definition need an IDE for editing (it has to do the projection!), language definition and extension always implies IDE definition and extension. The IDE will provide code completion, error checking and syntax highlighting for all language modules, even if they used in combinations. Finally, because the model is stored independent of its notation, it is possible to represent the same model in different ways simply by providing several projections. It is also possible to store out-of-band data, i.e. annotations on the core model/program. Examples of this include documentation, pointers to requirements (traceability) [2] or feature dependencies in the context of product lines [3].

Language workbenches deliver on the promise of removing the distinction between what is traditionally called programming and what is traditionally called modeling. This distinction is arbitrary anyway: developers want to express different concerns of software systems with abstractions and notations suitable to that particular concern, formally enough for automatic processing or translation, and with good IDE support. Projectional language workbenches deliver on this goal in an integrated, consistent and productive way.

JetBrains' Meta Programming System is an open source projectional language workbench [4]. Defining a language starts by defining the abstract syntax, the editor for the language concepts is specified in a second step. Lastly the generator is defined. It outputs text (for a low-level language) or it transforms higher-level code into code expressed in lower level languages. The higher-level to lower-level generators are not text generators, they transform abstract syntax trees.

The Modular Embedded Language

http://mbeddr.com hosts the modular embedded language (MEL). It defines several extensions to C useful for embedded programming. The target environment is the OSEK operating system [5], we use Lego Mindstorms for demo purposes. Mindstorms makes the system accessible and fun; OSEK makes it relevant because it is used a lot in automotive systems. OSEK is basically a set of C APIs together with a

configuration file, called OIL file. In the proof-of-concept, all these artifacts are generated from the MEL programs. Note that none of the language extensions described below are specific to OSEK - only the code generator is. By plugging in different code generators, different platforms can be targeted.

The program described below is a simple line follower robot. It uses a single light sensor to follow (one side of) a thick black line. It keeps track of the turning line by changing the speed of motors that drive the two main wheels.

Core Language. The core of the MEL is a relatively complete implementation of C. It supports variables, constants, enums, structs, functions, most of C's statements and expressions as well as the type system including pointers. No headers files are used (they are only generated during text generation at the end of the translation chain). Instead the language provides the concept of a module. Modules are like a namespace and contain all the elements that can be declared on top level in C programs. Modules can declare dependencies on modules making their *exported* contents visible to depending module.

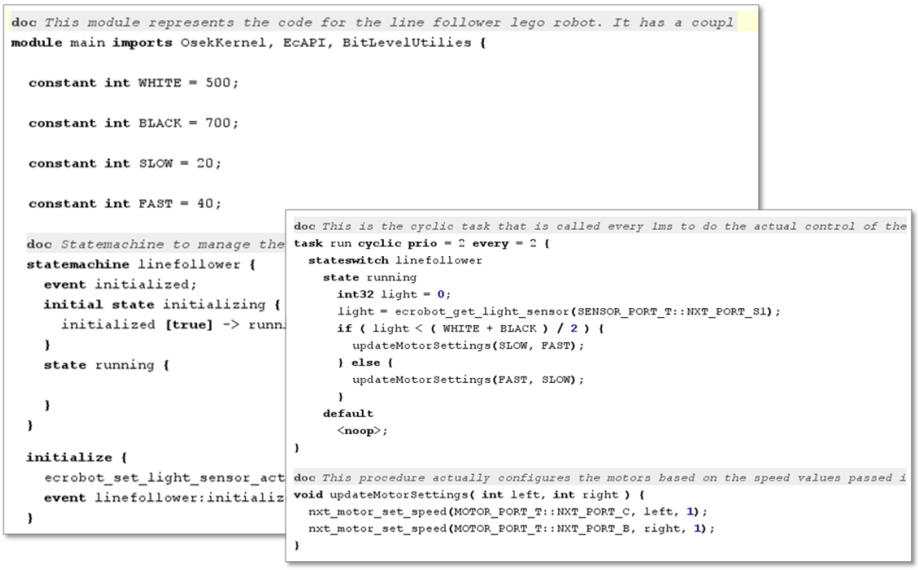

Figure 1: Code for the simple line follower robot expressed with the MEL

Figure 1 shows a screenshot of a basic line follower program. The module *main* uses three external API modules. It contains constants as well as a *statemachine* which manages the two phases of the program: *initialization* and *running*. The module also contains an *initialize* block (executed at program startup) which performs the initialization of the light sensor. The cyclic *run* task is responsible for reading the sensor and adjusting motor speeds. It is called every two system ticks. What the *run* task actually does is state-dependent by virtue of the *stateswitch*; if the linefollower state machine is in the *running* state it reads the light sensor and updates the motor settings. This state is entered by signalling *linefollower:initialized* at the end of the

setup task. Finally, the module contains the *updateMotorSettings* function which actually drives the motors.

Tasks. Tasks capture behaviour that is executed at specific times, such as *at startup* or *cyclic*. Tasks are like functions, but they have no arguments and no return type. Translation is target platform specific, for OSEK, tasks are mapped to a *void* function and a couple of entries in the OIL file.

Statemachines. The state machine language module supports the declaration of state machines (with states, events, transitions, guard conditions as well as entry and exit actions), a statement to fire events into a state machine as well as the *stateswitch*, a construct for implementing state-dependent behavior in tasks or functions.

The statemachine language module extends the core languag, and like functions or tasks, state machines can be contained in modules. The *event*... keyword and the *stateswitch* are statements, restricting their used to a statement context. Actions are statement lists, so every valid C statement can be used in them.

The generator is implemented as a model-to-model transformation that maps these construct down to the core language ("plain C"). The state machine becomes a variable (to hold the current state) as well as a function containing a switch/case statement to "execute" the transitions. States and events result in constants. The *event* statement calls this function passing in a constant representing the actual event. The *stateswitch* results in a couple of *if* statements querying the current state variable.

linefollower	*initializing*	*paused*	*running*	*crash*
initialized	true running			
bumped			true crash	
blocked			true paused	
unblocked		true running	true crash	

Figure 2: Editing the state machine as a table embedded in the program code

Projectional editing makes it possible to edit state machines using several notations. In addition to the textual notation above, we also support a table notation as shown in figure 2. A graphical notation is planned for the future.

A special kind of integer. Working with the sonar sensor to detect obstructions in the robot's path requires averaging over repeated measurements because the sensor is very sensitive. This is useful for many sensors, so an extension of the type system is included. Figure 3 A shows the declaration of a variable with an averaging type: the base type is *int*, the number of elements over which to average is 10 and the initialization value is 250. The =/ operator (see B) inserts a new measurement into to the buffer, calculating the new average and assigning it to the variable of the left of the operator. The variable can be read just like any other variable (see C).

A

```
{sonar} var avg(int, 10) currentSonar = 250;
```

B

```
{sonar} task sonartask cyclic prio = 2 every = 100 {
          currentSonar =/ ecrobot_get_sonar_sensor(SENSOR_PORT_T::NXT_PORT_S2);
          {debugOutput} debugInt(2, "sonar:", currentSonar);
        }
```

C

```
{sonar} if ( currentSonar < 150 ) {
  event linefollower:blocked
  terminate;
}
```

Figure 3: Averaging Variables

Safety. One aspect of safety is making sure that no overflows occur. For example, if a programmer assigns a value to an *uint8*, the value must not be larger than 255. If a module is marked as *safe,* all assignments and initializations are wrapped with a *checkSizeNN(...)* function call that logs an error if the value is outside the permitted range. An alternative implementation would transparently replace built-in operators with safe library functions such as *mul32*.

```
exported interface MotorControl {
  void stop(   );
  void setLeftSpeed( int8 speed );
  void setRightSpeed( int8 speed );
}

exported component Motors {
  provides motorControl : MotorControl;
}
```

```
exported component implementation MotorsNXT : Motors {

  procedure void motorControl.stop( ) {
    nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_B, 0, 1);
    nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_C, 0, 1);
  }

  procedure void motorControl.setLeftSpeed( int8 speed ) {
    nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_C, speed, 1);
  }

  procedure void motorControl.setRightSpeed( int8 speed ) {
    nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_B, speed, 1);
  }
}
```

Figure 4: An interface, a component and a component implementation

Components. Component-based development is very widespread in embedded systems. Components are the "smallest architecturally relevant building block". They encapsulate behaviour and expose all system-relevant characteristics declaratively. In MEL, a language module is available to express interfaces, components and component implementations (Figure 4).

The separation of interface, component and implementation also leads to a support for "compile-time polymorphism": since client code is written only against interfaces, the implementing component can be replaced. For example, this supports the encapsulation of platform-specific code and makes mocks for testing simple.

A domain specific language. All the facilities described so far address embedded software development generally. There was nothing in the languages that is specific to mobile robots that can drive and turn.

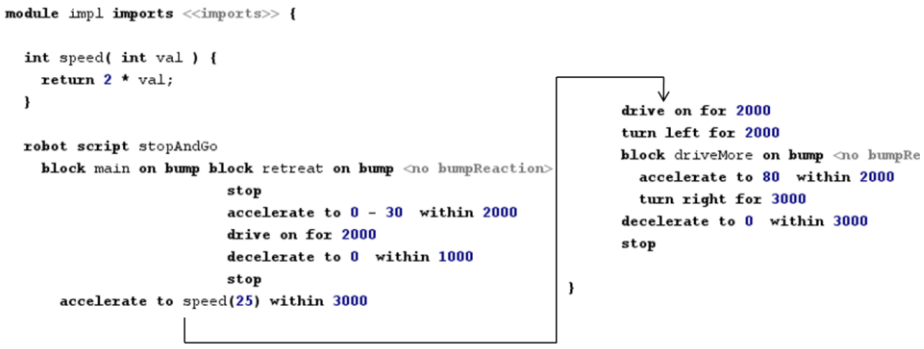

Figure 5: A robot routing script embedded in a module

Figure 5 shows a DSL for describing robot driving instructions. Since the robot routing language extends the core language, it can be embedded in a module. The *robot script* can even call C functions! This is an example of cascading languages: on top of a (relatively) general purpose language (Embedded/C/OSEK/Mindstorms), a (relatively) domain specific language (robot routing) is cascaded. The domain specific constructs are translated down to the more general purpose constructs for execution.

Requirements traceability and product line variability. The MEL also supports traceability to requirements as well as feature annotations to express product line variability. Arbitrary program elements can be annotated with expressions that define dependencies on features in a feature modeling tool. The expressions will be evaluated as low-level C code is generated to make sure only the "selected" parts of the program are generated. This evaluation of the feature expression can also be triggered when editing the program. As a consequence, the program can be edited

- as a whole, without feature dependency expression shown
- as a whole, with feature dependency expression
- and in any variant configuration

This makes working with variants of source code much easier, since the actual variants can be seen "live" in the editor. Figure 6 shows an example program with feature dependency annotations shown. Figure 1 at the beginning of the paper shows a variant, with feature dependency annotations hidden.

```
{sonar} task sonartask cyclic prio = 2 every = 100 {
          int s = ecrobot_get_sonar_sensor(SENSOR_PORT_T::NXT_PORT_S2);
          sonarHistory[sonarIndex] = s;
          sonarIndex = sonarIndex + 1;
          if ( sonarIndex == 10 ) {
            sonarIndex = 0;
          }
          int ss = 0;
          for ( int i = 0; i < 10; i = i + 1; ) {
            ss = ss + sonarHistory[i];
          }
          currentSonar = ss / 10;
          { {debugOutput} { debugInt(2, "sonar:", currentSonar); } }
        }

doc This is the cyclic task that is called every 1ms to do the actual control of the
task run cyclic prio = 2 every = 2 {
  stateswitch linefollower
    state running
      {bumper} int bump = ecrobot_get_touch_sensor(SENSOR_PORT_T::NXT_PORT_S3);
      {bumper} if ( bump == 1 ) {
        {debugOutput} { debugString(3, "bump:", "BUMP!"); }
        event linefollower:bumped
        terminate;
      }
      {sonar} if ( currentSonar < 150 ) {
        event linefollower:blocked
        terminate;
      }
      int light = ecrobot_get_light_sensor(SENSOR_PORT_T::NXT_PORT_S1);
      if ( light < ( WHITE + BLACK ) / 2 ) {
        updateMotorSettings(SLOW, FAST);
      } else {
        updateMotorSettings(FAST, SLOW);
      }
      {debugOutput} { debugInt(4, "light:", light); }
    {sonar} state paused
              updateMotorSettings(0, 0);
              if ( currentSonar < 255 ) {
                event linefollower:unblocked
              }
    {bumper} state crash
               updateMotorSettings(0, 0);
    default
      <noop>;
}
```

Figure 6: An program with feature-dependency annotations

Summary

The approach described in this article results in much more readable code, because the abstractions are better aligned with the program's intent. Low-level and abstract concepts can be mixed or separated, as necessary. It is possible to build language extensions specific to a platform, system or middleware, the effort is limited: for example, building the robot routing language took about 6 hours. Alternative notations such as tables or graphical notations will be possible by the end of 2010.

You can go to http://mbeddr.com for screencasts and papers [6,7,8]. The website also contains a link to MPS as well as to the MEL. All of the tools and the code are open source.

References

1. MISRA Group, Misra-C, http://www.misra-c2.com/
2. Wikipedia, Requirements Traceability, http://en.wikipedia.org/wiki/Requirements_traceability
3. Eisenecker, U., Czarnecki, K., Generative Programming, Addison-Wesley, 2000
4. JetBrains, Meta Programming System, http://jetbrains.com/mps
5. Sourceforge.net, nxtOSEK, http://lejos-OSEK.sourceforge.net/
6. Voelter, M., Product Line Engineering with Projectional Language Workbenches, http://voelter.de/data/pub/Voelter-ProductLineEngWithProjectionalLanguageWorkbenches.pdf
7. Voelter, M., Product Line Engineering with Projectional Language Workbenches, http://voelter.de/data/pub/ Voelter-EmbeddedSystemsDevelopmentWithProjectionalLanguageWorkbenches.pdf
8. Voelter, M., Solomatov K., Language Modularization and Composition with Language Workbenches, http://voelter.de/data/pub/ VoelterSolomatov_SLE2010_Language ModularizationAndCompositionLWBs.pdf

About the Author

Markus Völter works as an independent researcher, consultant and coach for itemis AG in Stuttgart, Germany. His focus is on software architecture, model-driven software development and domain specific languages as well as on product line engineering. Markus also regularly writes (articles, patterns, books) and speaks (trainings, conferences) on those subjects. Contact him via voelter@acm.org or www.voelter.de.