Embedded Software Development with Projectional Language Workbenches

Markus Voelter
independent/itemis

Oetztaler Strasse 38, 70327 Stuttgart, Germany
voelter@acm.org, http://voelter.de

Abstract. This paper describes a novel approach to embedded software development. Instead of using a combination of C code and modeling tools, we propose an approach where modeling and programming is unified using projectional language workbenches. These allow the incremental, domain-specific extension of C and a seamless integration between the various concerns of an embedded system. The paper does not propose specific extensions to C in the hope that everybody will use them; rather, the paper illustrates the benefits of domain specific extension using projectional editors. In the paper we describe the problems with the traditional approach to embedded software development and how the proposed approach can solve them. The main part of the paper describes our modular embedded language, a proof-of-concept implementation of the approach based on JetBrains MPS. We implemented a set of language extensions for embedded programming, such as state machines, tasks, type system extensions as well as a domain specific language (DSL) for robot control. The language modules are seamlessly integrated, leading to a very efficient way for implementing embedded software.

Keywords: embedded systems, language extension, domain-specific languages, code generation, projectional editing

1 Problems and Challenges in Embedded Systems Development

Embedded software is usually based on the C programming language. There are several reasons for that. There is significant experience with C because it has been used for a long time. New systems often reuse existing software written in C. The language is also relatively close to the hardware, resulting in efficient machine code. Finally, a large number of compilers exist for many different target platforms and (embedded) operating systems. These compilers usually produce decent quality code.

However, this approach is not without problems. For example, C has a number of features that are considered unsafe. Examples include void pointers, unions and (certain) automatic type conversions. Coding conventions, such as the Misra-C standard [1], prohibit the use of these features. Compliance is checked with automated checkers. Of course, while editing the code, developers can use these prohibited features, but get an error message only later. 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 (similar to polymorphism) hard. This becomes a significant problem as systems become larger and more complex - modern cars, for example, have up to 70 computers and controllers, several different bus systems as well as several gigabytes of software and configuration data. It is also hard to implement good abstractions for physical quantities (speed, rpm, temperatures), so most systems represent all kinds of quantities as integers. The resolution of the integers is optimized based on the value range of the quantity (0 to 300 for a speed, for example). However, since the compiler does not know about these ranges, developers have to take care of this resolution mapping themselves: a very tedious and error prone approach. It also leads to code that is hard to read because of all the inline conversion code. Finally, the preprocessor can be used for any number of mischief, thwarting attempts to analyse source code in a meaningful way.

To address these problems, modeling tools are used in the following two ways. One use case is the declarative specification of module and component structures for example by declaring interfaces, bus messages used by the module, or shared memory regions. A generator creates a header file based on which the developer then implements the actual functionality of the module. Examples for this approach include Autosar [5] or EAST-ADL [4] and many home-grown tools. The other use case for modeling tools is support for a specific behavioral paradigm: prominent examples include state-based behavior (as represented by (real time) UML tools) and abstractions from the control systems domain, often implemented using data flow block diagrams with tools such as Matlab/Simulink [20].

While these modeling tools are important and useful, they also create significant problems. No one tool addresses all aspects of an embedded system, and t various tools used for describing a system don't integrate very well with each other. Symbols defined in one tool cannot easily be referenced from models created with another tool. A special case of this problem is that the integration of manually written C code and abstractions defined in the models is hard. Also, the tools are often proprietary and very hard to customize and adapt to the specific environment of the project, product, or organization. There is also a more fundamental problem with the approach of using models and generators to generate header files or skeletons and then filling in the application logic via manual C programming: because of C's limited ability to express abstractions, the high-level constructs described in the models cannot be directly used or type checked in manually written C code. To make this possible, the C compiler, as well as the IDE, would have to be extended. This is unrealistic with today’s tools.

There are also significant challenges with process integration of the models and code including the ability to trace model and program elements back to requirements and the ability to express product line variability. Most tools do not provide first-class support for expressing variations in models or programs. If they do, then each tool has its own paradigm. In C, preprocessor *#ifdef* statements are often used. Since variability affects all artifacts in the software lifecycle, having a different way of specifying it in each tool without a way of pulling it all together is a serious problem. As the number of variants increases all the time, a scalable and well integrated facility for treating the variability is crucial. This is true in a similar way for requirements traceability.

Finally, in large engineering environments, the integration build can take very long, up to several hours. If the concerns described above are only checked during the build, this can reduce productivity significantly. Checking these concerns earlier in a tool would improve the situation. Better modularity, analyzable dependency specification and better support for unit testing would further improve the situation.

2 Proposed Solution

This paper proposes an approach for embedded development based on a modular, incremental extension of C. Projectional language workbenches (see next section; for now, consider them a tool for flexibly defining, mixing and extending languages) are the basis for this approach. Extensions can be on arbitrary abstraction levels, thereby integrating modeling and programming and avoiding tool integration issues. Code generation to C is used to integrate with existing compilers and platforms.

Here are some examples of how C could be extended to be more suitable as an embedded programming language. These are examples only, since the paper proposes an approach where C in extended in ways that are specific to organizations, projects or platforms.

A module system. Implementing modularized, reusable software requires a module system or component infrastructure. A module should declare everything it provides or expects from its environment in a way which is analyzable for tools. Interfaces could be functions that can be called by other modules, shared memory structures or published or consumed events. A module system would be directly integrated into the programming language, abd abstractions defined in the module definitions are first class citizens in the program and can be referenced directly.

Physical Quantities. The language would support the definition of quantities, a numeric type with a unit and a value range. Conversion rules for the value ranges and units are included in the type system. The IDE can check type compatibility, calculate resulting types/units and automatically insert value range conversions.

State-based behavior. Behavior in embedded systems is often state-based [2]. An enhanced language for embedded system development should contain native abstractions for states, transitions, and events. A module that contains state-based behavior would declare published and consumed events as part of its interface.

Dataflow. Many (especially control) systems use block diagrams [3]. In a new embedded programming language, these would be represented using dataflow ports on modules. They would serve as the basis for “wiring” the modules together, potentially using the well-known box-and-line notation.

Mathematical notation and abstractions. To support the concise expression of mathematical concepts, support for mathematical abstractions and notations should be built into the language. Examples include vectors, matrices, the sum symbol, or subscript notation. Because of the complete integration into the type system, static analysis and compile-time optimizations become possible.

Middleware Integration. In many embedded domains, standardized middleware frameworks are becoming de facto standards. Examples include EAST-ADL2 [4], AUTOSAR [5] or any number of frameworks for network/bus protocols. Developers are required to interface with these frameworks all the time. Language extensions specific to the target middleware would make a developer's job significantly simpler. IDE-level checks of correct usage also make development more robust and less error prone.

Variability. First-class support for expressing variability is crucial. At least, the markup of variable sections of models/code and their relationships to configuration features must be expressed. It must be possible to show programs with or without the variability markup as well as specific variants, i.e. the with all non-selected variable sections removed (as in [26]). Because the variability expression mechanism would be native to the language, it can be analyzed and checked in the IDE – it is not just implemented as a comment or a type-system-unaware preprocessor directive.

Traceability. The language should provide a way to embed references to other artifacts using some kind of referencing scheme. At the very least, pointers to requirements in systems like DOORS or Requisite Pro are necessary.

Integration and Build. Because many engineering and integration aspects are represented directly in the language (feature dependencies, "system constants", modules, dependencies) they can be checked earlier and easier. Build and integration times can be significantly reduced.

3 Language Workbenches & JetBrains MPS

To implement extensions such as those proposed in the previous section, it must be possible to modularize the overall language as otherwise it will become complex and bloated. It must also be extendible so that new language modules can be added and language concepts in existing modules can be adapted to project- or system specifics. Projectional language workbenches form the basis for such an approach.
The term Language Workbench has been coined by Martin Fowler in 2005 [6]. In this article he characterizes it as a tool with the following properties:

1. Users can freely define languages which are fully integrated with each other.
2. The primary source of information is a persistent abstract representation.
3. A DSL is defined in three main parts: schema, editor(s), and generator(s).
4. Language users manipulate a DSL through a projectional editor.
5. A language workbench can persist incomplete or contradictory information.

Projectional editing implies that all text, symbols, and graphics are projected, well-known from graphical modeling tools (UML, ER, State Charts): the model is stored independent of its concrete syntax, only the model structure is persisted, often using XML or a database. For editing purposes this abstract syntax is projected using graphical shapes. Users perform mouse gestures and keyboard actions tailored to the graphical editing to modify the abstract model structure directly. While the concrete syntax of the model does not have to be stored because it is specified as part of language definition and hence known by the projection engine, graphical modeling tools usually also store information about the visual layout.

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.

The following list shows the benefits of the approach:

- In projectional editing, no grammar or parser is used. Editing directly changes the program structure (AST). Thus, 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.
- Notations are more flexible than ASCII/ANSI/Unicode. Graphical, semi-graphical and textual notations can be mixed and combined. For example, a graphical tool for editing state machines can embed a textual expression language for editing the guard conditions on transitions.
- 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 languages, even when they are combined.
- Because the model is stored independent of its concrete notation, it is possible to represent the same model in different ways simply by providing several projections. Different viewpoints [7] of the overall program can be stored in one model; editing can be viewpoint or aspect specific. 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) [8] or feature dependencies in the context of product lines [9].

As a side effect, 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: 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. They do this by applying the technology known from modeling tools (projection) to editing any notation.

JetBrains MPS

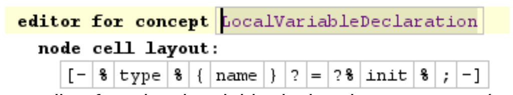
JetBrains' Meta Programming System is an open source projectional language workbench [10]. Defining a language starts by defining the abstract syntax, the editor for the language concepts is specified in a second step. Lastly the generator is defined. It outputs text (for a low-level language) or it 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.

Editing the tree as opposed to "real text" needs some accustomization. Without specific adaptations, every program element has to be selected from a drop-down list and "instantiated". However, MPS provides editor customizations to enable editing that resembles modern IDEs that use automatically expanding code templates. In some cases though, the tree shines through: Consider changing a statement like *int i = j+k;* to *int i = (j+k)*2;* you cannot simply move the cursor to the left of *j* and insert a left parenthesis. Rather, you have to select the + operator (the root node of the expression on the right) and use a *Surround with Parens* refactoring. Using (as the hotkey for this refactoring creates an editing experience very similar to "real" text.

Language definition, extension and composition with MPS

I have described language creation, extension and composition in MPS in a separate paper [15]. This section shows an example as a short summary. MPS, like other language workbenches, comes with a set of DSLs for language definition, a separate DSL for each language aspect. Language aspects include structure, editor, type system, generator as well as support for features such as quick fixes or refactorings.

Defining a new language starts by defining the language structure (aka meta model). This is very much like object oriented programming as language elements are represented as concepts that have properties, children and references to other concepts. The second step is the editor for the language concepts. An editor defines how the syntax for the concepts should look like - it constitutes the projection rules. Figure 1 is an example.



```
editor for concept LocalVariableDeclaration
  node cell layout:
    [- % type % { name } ? = ?% init % ; -]
```

Figure 1: Defining an editor for a local variable declaration statement (as in *int i = 2*2;*)

Next is the definition of the type system. For example, the *type* property of a *LocalVariableDeclaration* must be compatible with the type of its *init* expression.

At this point, the definition of the language and the basic editor, as well as the type system are finished. However, to use the new *LocalVariableDeclaration* statement, the user has to bring up the code completion menu in the editor, select the concept *LocalVariableDeclaration* and use tab or the mouse to fill in the various properties (*type, name, init*). A couple of editor customizations are necessary to make sure users can "just type" the declaration. I refer to [11] for details on how this works.

I already alluded to the relationship between object oriented programming and language definition in MPS. This analogy also holds for language extension and specialization. Concepts can extend other concepts, and subconcepts can be used polymorphically. Languages can extend other languages, too, and the sublanguage can contain subconcepts of concepts in the base language or can override the translation rules (generators) of concepts defined in the base language. Concept interfaces are also available. Using the Adapter pattern [12], unrelated concepts can be made to fit together. To use a *B* in places where an *A* (or subtypes) is expected, an adapter *BAdapter* that extends *A* and contains or references a *B* is necessary. As shown in [11], this approach supports embedding of completely unrelated languages.

Languages also define translation rules to lower-level languages or to text. MPS includes an incremental translation engine that reduces program code as long as translation rules are available for the program elements. At the end, text generators output regular program text that can be fed into a compiler.

Why language extensions

Compared to using libraries and frameworks, language extension has a couple of advantages. First of all, the syntax can be adapted. C does not have a very flexible syntax, so internal DSLs are not easily possible. Also, for all the language extensions created, MPS provides IDE support: syntax coloring, code completion as well as error reporting. Much more meaningful static checks and analyses can be implemented (in the IDE) based on the language because language modules can be limited in their expressive power, making semantic analyses feasible. Finally, all kinds of optimizations can be performed during C code generation.

In short, language extension provides all the benefits of (what's traditionally known as) modeling, but without all the integration and tooling headaches. The perceived disadvantage of language extension - mainly the effort of implementing the extensions - is addressed very effectively by MPS. As discussed in the Evaluation section, the effort for building language extensions is very reasonable.

4 Proof of Concept

A group of people (Bernhard Merkle, Alexander Bernauer, myself) are working on a proof-of-concept, the modular embedded language (MEL). It contains several of the C extensions described above. The system is a Lego Mindstorms 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. It is shown in Figure 2.

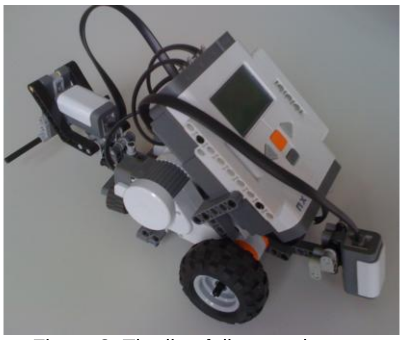


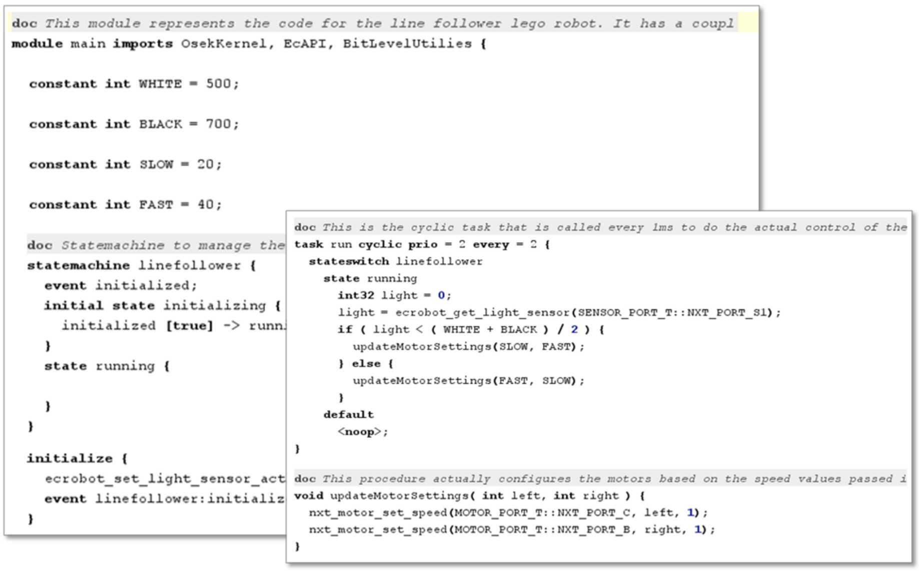
Figure 2: The line follower robot

The software is implemented in C based for the OSEK operating system [14] for Lego Mindstorms [13]. Using Mindstorms makes the case study acessible and fun; OSEK makes it relevant to real-world software because it is used a lot in automotive systems. It 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. For testing and debugging, a win32 target is available.

Core Language. The core of the MEL is a relatively complete implementation of the C programming language in MPS. The core language 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 variables, typedefs, structs and functions - all the elements that can be declared on top level in C programs. Module contents can be *exported*. Modules can declare dependencies to other modules which makes their exported contents visible to depending module.

Note that the system does not include the preprocessor. Instead it provides first class concepts for many of the typical use cases for the preprocessor (such as constants or conditional code regions). Existing code can thus not be directly imported into the MEL. However, the system supports interfacing with code (as represented by header files) that resides outside of MEL/MPS. This is described below.



```
doc This module represents the code for the line follower lego robot. It has a coupl
module main imports OsekKernel, EcAPI, BitLevelUtilies {

  constant int WHITE = 500;

  constant int BLACK = 700;

  constant int SLOW = 20;

  constant int FAST = 40;

  doc Statemachine to manage the
  statemachine linefollower {
    event initialized;
    initial state initializing {
      initialized [true] -> runn
    }
    state running {

    }
  }

  initialize {
    ecrobot_set_light_sensor_act
    event linefollower:initializ
  }
```

```
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
      int32 light = 0;
      light = ecrobot_get_light_sensor(SENSOR_PORT_T::NXT_PORT_S1);
      if ( light < ( WHITE + BLACK ) / 2 ) {
        updateMotorSettings(SLOW, FAST);
      } else {
        updateMotorSettings(FAST, SLOW);
      }
    default
      <noop>;
}

doc This procedure actually configures the motors based on the speed values passed i
void updateMotorSettings( int left, int right ) {
  nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_C, left, 1);
  nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_B, right, 1);
}
```

Figure 3: Code for the simple line follower robot expressed with the MEL

Figure 3 shows a screenshot of the basic line follower program implemented with MEL. It contains a module *main* that uses three external API modules. The module 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 (called 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.

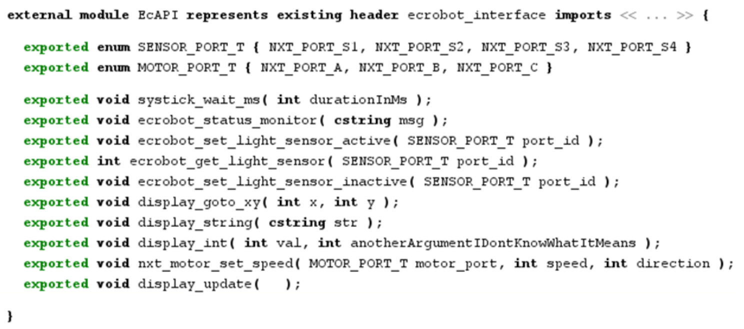
Let us look at some of the features available in the MEL.

Tasks. Tasks capture behaviour that should be executed at specific times, currently *at startup* and *cyclic* are supported. Tasks are like functions, but they have no arguments and no return type. In fact, use of the *return* keyword is prevented inside tasks.

Translation is target platform specific and can be exchanged by using a different generator. For the OSEK target, tasks are mapped to a *void* function and a couple of entries in the OIL file: for a cyclic task, a counter, an alarm and a task have to be declared. In addition, a function that drives the counter based on an interrupt is required. All of this is transparent to the programmer.

Statemachines. The MEL contains a module for state machines. It 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 statemachine as well as a *stateswitch* statement to create a *switch*-like construct for implementing state-dependent behaviour in tasks or functions.

The statemachine language module extends the core language. Like functions or tasks, state machines implement the *IModuleContent* interface so they can be put into modules. The *event*… keyword and the *stateswitch* are subtypes of the core language's *Statement* concepts, making sure they can be used in statement context. Actions are statement lists, so every valid C statement can be used in them.



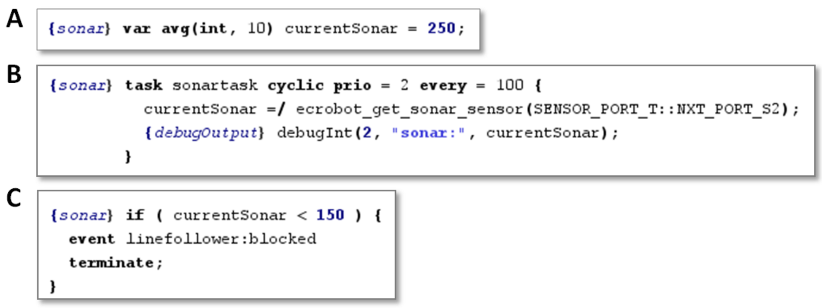
```
external module EcAPI represents existing header ecrobot_interface imports << ... >> {

  exported enum SENSOR_PORT_T { NXT_PORT_S1, NXT_PORT_S2, NXT_PORT_S3, NXT_PORT_S4 }
  exported enum MOTOR_PORT_T { NXT_PORT_A, NXT_PORT_B, NXT_PORT_C }

  exported void systick_wait_ms( int durationInMs );
  exported void ecrobot_status_monitor( cstring msg );
  exported void ecrobot_set_light_sensor_active( SENSOR_PORT_T port_id );
  exported int ecrobot_get_light_sensor( SENSOR_PORT_T port_id );
  exported void ecrobot_set_light_sensor_inactive( SENSOR_PORT_T port_id );
  exported void display_goto_xy( int x, int y );
  exported void display_string( cstring str );
  exported void display_int( int val, int anotherArgumentIDontKnowWhatItMeans );
  exported void nxt_motor_set_speed( MOTOR_PORT_T motor_port, int speed, int direction );
  exported void display_update(   );

}
```

Figure 4: An example external module

The generator is implemented as a model-to-model transformation that maps these construct down to plain C (as defined in MPS). 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.

Calling external APIs. OSEK provides a set of APIs. Since they are not implemented with the MEL in MPS, it must be possible to call low-level APIs. External modules contain the signature of the functions that need to be called from within MPS. An external module also associates a set of linked resources. The code generated from modules that use an external module will *#include* the header represented by the external module and the linked resources will be added to the generated makefile to make sure they are linked into the resulting image.

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 to provide averaging variables is included. Figure 5 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. From this declaration the generator produces two more variables: a buffer that remembers the last 10 measurements and an index into that buffer to determine where the next value is stored. 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;
}
```

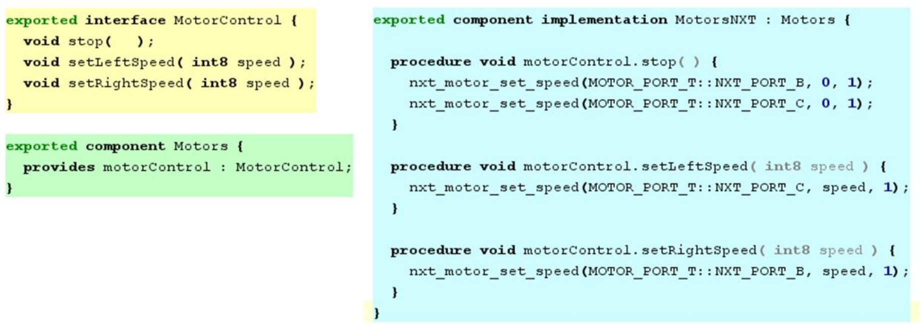
Figure 5: Averaging Variables

The system also includes physical quantities. They are different from *int*s in that they have an associated physical unit as well as a range (an example is *speed*, unit: *kph*, range: *0..300*). Based on the range, the resolution to the base type (*int8*, *int16*, *int32* and *int64*) is calculated. The values are stored with as high a precision as possible with the selected base type. The quantities are separate types and cannot be assigned to basic *int*s. A cast is required to make such a conversion explicit.

Safety. One aspect of safety is making sure that values are not assigned to variables that cannot hold these values. For example, if a programmer assigns a value to an *uint8*, the value must not be larger than 255. To enforce this, coding guidelines of many organizations require the use of safe functions (such as *mul32(...)*) instead of the built-in operators. The MEL supports such an approach transparently. A module can simply be 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*.

Components. Component-based development is widespread in software engineering in general, and in embedded systems in particular. Components are the "smallest architecturally relevant building block". They encapsulate behaviour and expose all system-relevant characteristics declaratively. In the proof-of-concept, a language module is available to express interfaces, components and component implementations, as shown in Figure 6.

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.

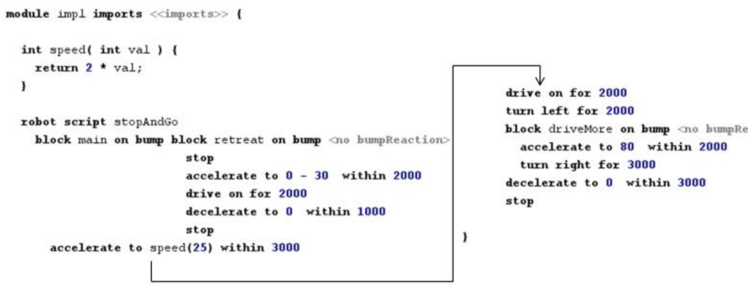


```
exported interface MotorControl {
  void stop(   );
  void setLeftSpeed( int8 speed );
  void setRightSpeed( int8 speed );
}

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 6 An interface, a component and a component implementation

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.



```
module impl imports <<imports>> {

  int speed( int val ) {
    return 2 * val;
  }

  robot script stopAndGo
    block main on bump block retreat on bump <no bumpReaction>
                        stop
                        accelerate to 0 - 30  within 2000
                        drive on for 2000
                        decelerate to 0  within 1000
                        stop
      accelerate to speed(25) within 3000

                                                     drive on for 2000
                                                     turn left for 2000
                                                     block driveMore on bump <no bumpRe
                                                       accelerate to 80  within 2000
                                                       turn right for 3000
                                                     decelerate to 0  within 3000
                                                     stop
                                                   }
```

Figure 7: A robot routing script embedded in a module

Consider now the role of a robot vendor who sells Lego robots with two wheels that can drive a predefined route. Each customer wants a different predefined route. The vendor has to develop a different route-driving program for each customer. Of course this can be achieved with tasks, state machines, variables and functions. But it would be better if a domain specific language for defining routes was available.

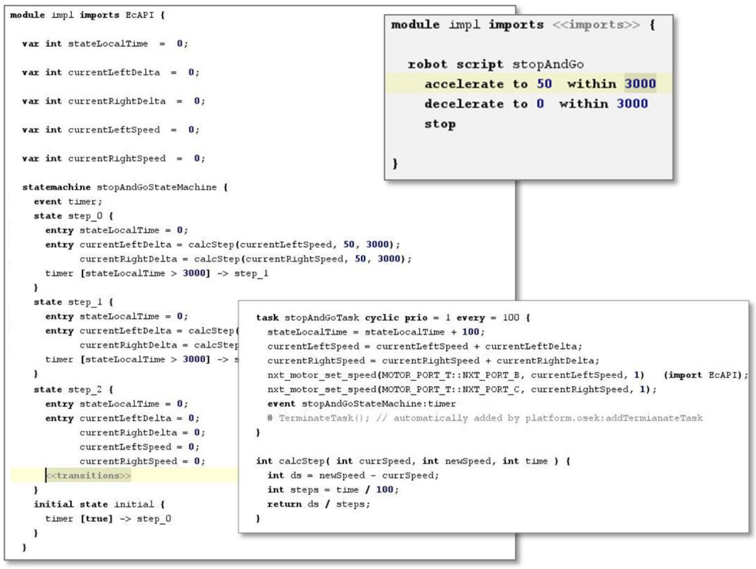
This is an example of cascading languages: on top of a (relatively) general purpose language (Embedded/C/OSEK/Mindstorms), a (relatively) domain specific language (robot routing) is cascaded. The domain specific constructs are translated down to the more general purpose constructs for execution. Figure 7 shows an example route expressed in the new *TwoWheels* DSL.

This program uses native route definition constructs. Since the robot routing language extends the core language, and the *robot script* construct implements the *IModuleContents* interface, it can be embedded in a module - like the lower-level constructs. The *robot script* can even call functions. The robot routing language is executed by translation into the following lower level constructs:

- a state machine that keeps track of the current command/step

- module variables that remember the current speeds for the two motors
- module variables that store the current deltas for the speeds of both motors to be able to "ramp up" the speeds in accelerate and decelerate commands
- a cyclic tasks that updates motor speeds based on the currently selected deltas.

Figure 8 shows a simple robot script (grey box) and its translation into states and tasks, which are in turn translated into "regular" C and the OIL file by the incremental, multi-stage translation engine.



```
module impl imports EcAPI {

  var int stateLocalTime  =  0;

  var int currentLeftDelta  =  0;

  var int currentRightDelta  =  0;

  var int currentLeftSpeed  =  0;

  var int currentRightSpeed  =  0;

  statemachine stopAndGoStateMachine {
    event timer;
    state step_0 {
      entry stateLocalTime = 0;
      entry currentLeftDelta = calcStep(currentLeftSpeed, 50, 3000);
            currentRightDelta = calcStep(currentRightSpeed, 50, 3000);
      timer [stateLocalTime > 3000] -> step_1
    }
    state step_1 {
      entry stateLocalTime = 0;
      entry currentLeftDelta = calcStep(
            currentRightDelta = calcStep
      timer [stateLocalTime > 3000] -> s
    }
    state step_2 {
      entry stateLocalTime = 0;
      entry currentLeftDelta = 0;
            currentRightDelta = 0;
            currentLeftSpeed = 0;
            currentRightSpeed = 0;
      <<transitions>>
    }
    initial state initial {
      timer [true] -> step_0
    }
  }
```

```
module impl imports <<imports>> {

  robot script stopAndGo
    accelerate to 50  within 3000
    decelerate to 0  within 3000
    stop

}
```

```
task stopAndGoTask cyclic prio = 1 every = 100 {
  stateLocalTime = stateLocalTime + 100;
  currentLeftSpeed = currentLeftSpeed + currentLeftDelta;
  currentRightSpeed = currentRightSpeed + currentRightDelta;
  nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_B, currentLeftSpeed, 1)   (import EcAPI);
  nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_C, currentRightSpeed, 1);
  event stopAndGoStateMachine:timer
  # TerminateTask(); // automatically added by platform.osek:addTermianateTask
}

int calcStep( int currSpeed, int newSpeed, int time ) {
  int ds = newSpeed - currSpeed;
  int steps = time / 100;
  return ds / steps;
}
```

Figure 8: A simple robot script and the resulting lower level program

Requirements traceability and product line variability. The MEL also supports traceability to requirements as well as feature annotations to express product line variability. Since it is a complex and big topic on its own I have described this in a separate paper [15].

5 Discussion

The experience in building the MEL for Lego Mindstorms allows us to conclude a number of things which we describe in this section.

It is useful to extend plain C with concepts that are tailored for embedded programming. Programs become more readable and can be analyzed much more thoroughly.

The ability to extend languages quickly and easily is very powerful. One really starts thinking differently about programming and modeling if the language isn't a fixed quantity anymore. More specifically, being able to go bottom (i.e. starting with C code, not with high-level models) up has proven useful. Additional abstractions are added when the need for them becomes apparent at any time during the project.

The ability to mix what's traditionally considered programming and what's traditionally considered modeling is clearly an advantage. Being able to use C expressions in state machine guard conditions, or using C statements in the actions of state machines is very useful - especially since consistent IDE support is available and the symbols and type systems are semlessly integrated.

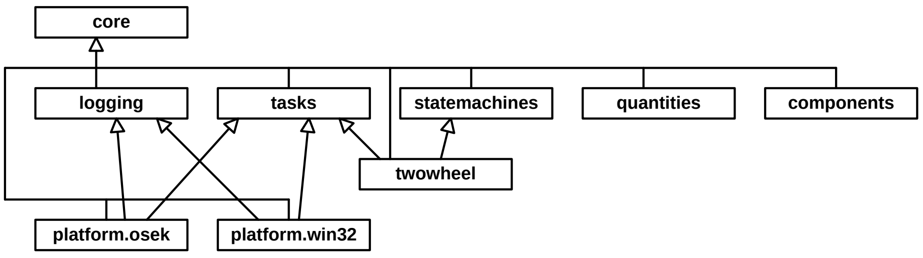


Figure 9: Dependencies between the languages mentioned in this paper

It is feasible to package the various aspects into separate language modules (see Figure 9). This makes sure languages do not become bloated - different users can extend the core with concepts suitable to what they want to do. Low level and high level constructs are all implemented in the same tooling makes cascading and incremental refinement feasible and mixing of abstraction levels possible.

The generated C code not different from what developers would have written by hand. Transformations or code generation is incremental (higher level constructs are transformed into instances of lower level constructs), making optimizations implemented in lower levels available to higher levels automatically.

The effort for building the languages, the IDE and the language extensions is very modest. For example, developing the basic C implementation has taken about three weeks. Adding the state machine facilities (state machines, event keyword, stateswitch, including the generator down to C) has been done in one afternoon. Creating the robot routing DSL on top was a matter of 4 hours, including the mapping down to tasks and statemachines. This proofs that adding language abstractions as they become necessary is feasible.

6 Future Work

In addition to continue working on our Lego proof-of-concept, we're planning to engage on a larger, real-world prototype embedded systems programmers. In this project we will

- identify and implement more abstractions and notations as well as the relationships between them to facilitate meaningful modularization
- understand and prototype the integration of the approach and the language workbenches with build, test, version control, requirements, and variant management tools to make sure the approach can be used in real development environments.
- integrate the languages with the debugger framework that will ship with the next version of MPS to be able to debug programs on MPS-source level.

- explore the use of other notations, mainly tabular and graphical
- target a different platform, for example a real AUTOSAR-based [5] vehicle ECU

7 Related Work

Language composition is researched by various groups, exemplified by tools such as SDF [16] and Monticore [17]. Projectional editing is also supported by Intentional Software's Domain Workbench [25]. However, since I describe language composition in detail in another paper [11]. I will not discuss this topic any further here.
The closest relative to the approach described in this paper is nesC [22], an extension to C designed to embody the concepts and execution model of TinyOS, an event-driven operating system designed for sensor network nodes that have very limited resources. Like the MEL proposed in this paper, nesC is an extension of C. However, it is one *specific* extension, whereas this paper produces modular language extension as a general paradigm for efficient embedded software development. However, the nesC extensions are a good example of how extensions to a general purpose language are useful; nesC could be easily built with the approach described in this paper.

SystemC [21] is a C++-based class library for systems engineering and embedded software development. It has some of the same goals as the MEL introduced here, but uses libraries, not language extension. Language extension is more powerful because the syntax can be adapted, IDE support can be provided and semantic validation can be more meaningful. Modelica [22] is an open standard for systems engineering and simulation. It provides many meaningful abstractions for systems engineering, among them the ability to define formulae and differential equations. However, Modelica does not allow language extension. I consider the ability to extend the language with domain-specific concepts the main contribution of the MEL. A similar statement can be made for Matlab [23]. It is very widely used in control and automation applications. Just like Modelica, Matlab comes with a simulation workbench. The language is textual and graphical (block diagrams). New blocks can be added through via libraries. Code generation is supported based on its Real-Time Workshop. However, real language extension is not possible, either. SCADE [21] is another language for embedded systems engineering with a strong focus on realtime and safety. Again, the core language is not extendible. It is not clear at this point whether extensible languages would make safety and certification issues simpler or more complicated. ECSL-DP [24] is a tool suite for distributed embedded control systems. It integrates a set of tools, among them, Matlab [20] and GME. It provides graphical notations, and, again, the idea of extending the languages in domain-specific ways is not part of the methodology. It presents *one* set of languages, not workbench for domain-specific extensions.

Acknowledgements

I would like to thank Alexander Bernauer, Christoph Elsner, Iris Groher, Michael Kircher, and Guy Philipp Bollbach for their valuable feedback on previous versions of the paper. I also want to thank Bernhard Merkle, without whom I would have had a much harder time getting started with Mindstorms and OSEK.

References

1. MISRA Group, Misra-C, http://www.misra-c2.com/
2. Wikipedia, State Machines, http://en.wikipedia.org/wiki/Finite-state_machine
3. National Instruments, Block Diagram Data Flow, http://zone.ni.com/reference/en-XX/help/371361B-01/lvconcepts/block_diagram_data_flow/
4. ATESST Consortium, EAST ADL Specification, http://www.atesst.org/home/liblocal/docs/EAST-ADL-2.0-Specification_2008-02-29.pdf
5. AUTOSAR Development Partnership, Automotive Open System Architecture, http://www.autosar.org/
6. Fowler, M., Language Workbenches: The Killer-App for Domain Specific Languages?, http://martinfowler.com/articles/languageWorkbench.html
7. Wikipedia, View Model, http://en.wikipedia.org/wiki/View_model
8. Wikipedia, Requirements Traceability, http://en.wikipedia.org/wiki/Requirements_traceability
9. Eisenecker, U., Czarnecki, K., Generative Programming, Addison-Wesley, 2000
10. JetBrains, Meta Programming System, http://jetbrains.com/mps
11. Voelter, M., Solomatov, K., Language Modularization and Composition with Projectional Language Workbenches illustrated w/ MPS, submitted to SLE 2010
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994
13. Lego, Mindstorms, http://mindstorms.lego.com
14. Sourceforge.net, nxtOSEK, http://lejos-OSEK.sourceforge.net/
15. Voelter, M., Product Line Engineering with Projectional Language Workbenches, submitted to GPCE 2010
16. Visser, E., Syntax Definition, http://swerl.tudelft.nl/bin/view/EelcoVisser/SyntaxDefinition
17. Software Engineering Group at RWTH Aache, Project MontiCore, http://www.monticore.de/
18. Open SystemC Initiative, SystemC, http://www.systemc.org/
19. Modelica Association, Modelica, http://modelica.org/
20. The MathWorks, Matlab - the language of technical computing, http://www.mathworks.com/products/matlab/
21. Esterel Technologies, SCADE, http://www.esterel-technologies.com/products/scade-suite/
22. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D., The nesC language: A holistic approach to networked embedded systems, ACM SIGPLAN Notices, Volume 38, Issue 5 (May 2003)
23. Fowler, M., Domain Specific Languages, http://www.martinfowler.com/bliki/DomainSpecificLanguage.html
24. Neema, S., Karsai, G., Vizhanyo, A., Embedded Control Systems, Language for Distributed Processing (ECSL-DP), http://www.omg.org/news/meetings/workshops/MIC_2004_Manual/06-1_Neema_etal.pdf
25. Intentional Software, Intentional Domain Workbench, http://intentsoft.com
26. Kästner, C., Thüm, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., Apel, S.. FeatureIDE: Tool Framework for Feature-Oriented Software Development. In Proceedings of ICSE 2009, Formal Demonstration Paper