Towards an Extensible C for Embedded Programming

Markus Voelter[1], Bernhard Schaetz[2], Daniel Ratiu[2], and Bernd Kolb[3]

[1] independent/itemis
[2] Fortiss GmbH
[3] itemis AG

Abstract. Embedded software development suffers from inadequate languages and tools. The C programming language does not provide means for defining adequate abstractions. Furthermore, modeling tools and their languages are typically closed and cannot be adapted to particular domains. In this paper we present the current state of an extensible language based on C that combines the best of both worlds. The mbeddr C language is developed as part of the LWES KMU Innovativ research project run by fortiss, itemis, Lear and Sick.

1 Introduction

In the LWES KMU Innovativ project, the project partners explore how domain-specific extensions to C can increase the productivity and analyzability of C programming. We have implemented C in the MPS language workbench, which, as a consequence of MPS's architecture, makes the language and IDE extensible in meaningful ways. We are now in the process of developing actual extensions to C. In this paper, we describe some of the challenges in embedded development, as well as some of the extensions we built to address them.

The LWES project is still work in progress. Details can be found at `http://mbeddr.com`, and the project partners are itemis, fortiss, Lear Automotive and SICK Sensortechnik.

2 Challenges in Embedded Development

This section provides an overview over some of the typical challenges encountered in the development of embedded systems.

Abstraction without Runtime Cost Abstractions that fit the problem domain are important for modular and maintainable software. In embedded software, these abstractions should come with minimal runtime cost, because the target environments for embedded software are often severely resource constrained. This means that many of the abstractions have to be resolved statically during translation, resulting in an efficient C implementation.

Static Checks and Verification To build safe, secure and real-time systems, various forms of static analysis are used, from the compiler's type checks to sophisticated model checking approaches. To make verification feasible, those parts of a system that have to be verified should be isolated from the rest, and expressed in a formalism that makes the relevant verification feasible. An example formalism is state machines. However, these parts of the system still have to be integrated with the rest of the system, usually written in C. Extracting these parts into a completely separate environment, such as a state chart modeling tool, results in integration problems.

C considered Harmful While being efficient and flexible, especially for low-level, machine-dependent code, some of C's features are often considered harmful. Unconstrained casting via `void` pointers, using `int`s as Booleans or data structures like `union`s can result in hard-to-detect runtime errors. Consequently, these features of C, as well as others, are prohibited in many organizations. Standards such as MISRA-C limit the language subset to what is considered *safe*. However, most C IDEs are not out of the box MISRA-C aware, so separate checkers have to be used and they may or may not be integrated with developers' tools. This makes it hard for developers to use the safe language subset of C efficiently.

Inclusion of Meta Data Non-trivial embedded systems often associate various kinds of meta data with program elements. Examples include physical units, data encodings and value ranges for types, access restrictions, memory mappings and access frequency restrictions for (global) variables as well as trace information from code to other artifacts, typically requirements. These meta data often form the basis for checking and analysis tools. Since C programs cannot express these meta data directly (except as comments or `pragma`s, with the obvious drawbacks), they are stored separately, often in XML files, leading to unnecessary complexity and maintainability problems.

Product Line Support Most embedded systems are developed as part of product lines. This leads to two problems. First, each product line, or domain, comes with its own set of abstractions. If those can be made available to the developer directly, writing code for that domain becomes much simpler, and checks and verifications become more meaningful. We have already discuss these limitations in building custom abstractions above. A second challenge is the support for product line variability where certain (parts of) artifacts are only included in some of the products in the product line. This variability typically cross-cuts all the artifacts and tools, yet still has to be managed efficiently. Today this variability on code level is often implemented with the preprocessor leading to the problems discussed above and preventing static analysis of variant consistency.

3 mbeddr C Solution Approach

mbeddr C addresses these challenges by providing an extensible version of C. Instead of artificially separating "programming" and "modeling" into different tools, with the resulting integration challenges, concepts at every abstraction

level can be used in one program. Extensions to C a modular and incremental; new abstractions can be added at any time. Full IDE support, including syntax coloring, code completion, go-to-definition as well as refactoring are provided for the base language and all extensions.

4 Example Extensions to C

In this section we show how to use the extensible C language and some example extensions to address the challenges discussed above (Section 2).

```
int decide(int x, int y) {
  return int, -1
```

	x == 0	x == 1	x > 1
y == 0	0	1	2
y == 1	1	3	2
y > 1	2	2	4

```
;
} decide (function)
```

Fig. 1. A decision table evaluates to the value in the cell for which the row and column headers are true, the default value `-1` otherwise.

Decision Tables are a new expression. An example is shown in Fig. 1. Since expressions need a type and a value, decision tables specify the expected type of the result expressions (`int` in the example) and a default value (`-1`). A decision table is basically a two-level `if` statement. It evaluates to the value in the cell whose column and row headers evaluate to `true`.

Unit Tests are additional a new kind of top level constructs (Fig. 2). They are introduced in a separate *unittest* language that extends the C core. Unit tests are like void functions without arguments. The *unittest* language also introduces the `assert` and `fail` statements which can only be used inside of a test case.

```
module UnitTestDemo imports multiplier {
  exported test case testMultiply {
    assert(0) times2(21) == 42;
    if ( 1 > 2 ) { fail(1); }
  }
}
```

Fig. 2. Test cases are new top level constructs. The *unittest* language also introduces the `assert` and `fail` statements which can only be used inside of a test case. The arguments to `assert` and `fail` denote the index of the statement. In error messages this number is output as a way of finding the cause in the code. These indexes are automatically projected and cannot be changed in the editor.

State Machines provide a new top level construct as well as new statements and expressions to interact with state machines from regular C code. Entry, exit and transition actions may only access variables defined locally in state machines and set output events. This way, state machines are semantically isolated from the remaining system and can be model checked using the integrated NuSMV model checker. State machines can be connected to the surrounding C program by mapping output events to function calls and by regular C code triggering input events in the state machines. Both these aspects are not relevant to verification.

```
module Statemachine from cdesignpaper.statemachine {
  statemachine Counter {
    in start()
       step(int[0..10] size)
    out started()
        resetted()
        incremented(int[0..10] newVal)
    vars int[0..10] currentVal = 0
         int[0..10] LIMIT = 10
    states (initial = start)
     state start {
       on start [] -> countState { send started(); }
     }
     state countState {
       on step [currentVal + size > LIMIT] -> start { send resetted(); }
       on step [currentVal + size <= LIMIT] -> countState {
         currentVal = currentVal + size;
         send incremented(currentVal);
       }
       on start [ ] -> start { send resetted(); }
     }
  }
}
```

Fig. 3. A state machine embedded into a C module. It declares in and out events as well as local variables, states and transitions. Transitions react to in events and out events can be created in actions. State machines can be verified with the NuSMV model checker, a topic not discussed further in this paper. Through bindings (not shown) they can interact with surrounding C code, e.g. by calling functions when an out event is created.

Components are new top level constructs used for modularization, encapsulation and the separation between specification and implementation (Fig. 4). Interfaces declare operation signatures that can be implemented by components. Provided ports specify the interfaces offered by a component, required ports specify the interfaces a component expects to use. Different components can implement the same interface operations differently. Components can be instantiated, and each instance's required ports have to be connected to compatible provided ports provided by other component instances.

Physical Units are new types that, in addition to defining their actual data type, also specify the physical unit (such as m/s or N). They also specify a resolution. New literals are defined to support specifying values for these types that include the physical unit. The type checker takes into account unit compatibility. Scaling (e.g. if working with N and kN in one expression) is taken into account automatically.

Requirements Traces Requirements traces are meta data annotations that point to requirements, essentially elements in other models imported from tools such as DOORS. Requirements traces can be applied to any program element without the definition of that element being aware of this (see Fig. 5).

Presence Conditions Like requirements traces, presence conditions can be attached to any program element. They are Boolean expressions over features in a feature model (essentially configuration switches). Their semantics is that the

```
module Components {
  c/s interface Calculator {
    int multiply(int x, int y) ;
  }

  component Computer {
    provides Calculator calc
    requires LoggingService log
    int calc_multiply(int x, int y) <- op calc.multiply {
      log.info("called with " + x + " and " + y);
      return x + y;
    }
  }

  component PrimitiveComputer {
    provides Calculator calc
    int calc_multiply(int x, int y) <- op calc.multiply {
      int res = 0;
      for (int i = 0; i < y; i++) {
        res = res + x;
      }
      return res;
    }
  }
}
```

Fig. 4. Two components providing the same interface. The `<-` notation maps operations offered through provided ports to their implementation in components.

element they are attached to is only part of a program variant, if the Boolean expression is true for the feature combination defined by that variant's feature configuration. During code generation, the program is "cut down" by removing all those elements whose condition is false. This can also be done in the editor, supporting variant-specific editing of the program.

Safe Modules restrict the set of constructs that can be used inside them. For example, pointer arithmetics is disallowed. Errors are reported if this is attempted. In addition, runtime range checking is performed for arithmetic expressions and assignments. To enable this, arithmetic expressions are replaced by function calls that perform range checking and report errors if an overflow is detected.

5 Related Approaches

In addition to the general-purpose embedded software modeling tools such as Simulink and ASCET, much more specific languages have been developed. Examples include Feldspar [1], a DSL embedded in Haskell for digital signal processing or Hume [2], a DSL for real-time embedded systems. Our approach is different because our DSLs are directly integrated into C, whereas the examples mentioned in this paragraph are standalone DSLs that generate C code.

Static analysis of C programs is an active research area (as exemplified by [3–5]), and several commercial tools are available, such as the Escher C Verifier or Klocwork. We believe that we can simplify some of the analyses provided by these tools by providing suitable extensions to C which embody relevant

```
initialize {
  trace OptionalOutput
  { debugString(0, "state:", "initializing"); }
  ecrobot_set_light_sensor_active(SENSOR_PORT_T::NXT_PORT_S1);
  trace Calibration
  ecrobot_init_sonar_sensor(SENSOR_PORT_T::NXT_PORT_S2);
  trace OptionalOutput
  { debugString(0, "state:", "running"); }
  event linefollower:initialized
}
```

Fig. 5. Requirements traces can be attached to any program element of any language. An intention (pressing Alt-Enter and selecting *Add Trace*) is used to attach them.

semantics directly, avoiding the need to reverse engineer the semantics for static analysis. For example, by expressing state-based behavior directly using state machines instead of a low level C implementation, the state space relevant to a model checker can be reduced significantly, making model checking more feasible.

6 Summary

The implementation of the basic C language in MPS took ca. 3 person months. The implementation of extensions such as decision tables is a matter of hours. Components and state machines took a couple of days each. Extensions include syntax, type system, semantics and IDE.

The example extensions we have built so far is a strong indication that, from a technical perspective, the extension of C with domain specific concepts works. We are now in the process of building extensions requested by our project partners Sick and Lear to validate the approach in a real, industrial setting.

The implementation of C, as well as the MPS language workbench itself are Open Source and available from:

- MPS: `http://jetbrains.com/mps`
- mbeddr C: `http://mbeddr.com`

References

1. E. Axelsson, K. Claessen, G. Devai, Z. Horvath, K. Keijzer, B. Lyckegard, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *MEMOCODE 2010*.
2. K. Hammond and G. Michaelson. Hume: a domain-specific language for real-time embedded systems. In *GPCE 03*, GPCE '03. Springer-Verlag New York, Inc., 2003.
3. S. Karthik and H. G. Jayakumar. Static analysis: C code error checking for reliable and secure programming. In *International Enformatika Conference, IEC 05*, pages 434–439. Enformatika, Canakkale, Turkey, 2005.
4. A. Mine. Static analysis of run-time errors in embedded critical parallel c programs. In *ESOP 2011*, volume 6602 of *LNCS*, pages 398–418. Springer, 2011.
5. A. Puccetti. Static analysis of the xen kernel using frama-c. *J. UCS*, 16(4), 2010.