

# Exploiting Modular Language Extensions in Legacy C Code: An Automotive Case Study

Andreas Grosche<sup>1</sup>, Burkhard Igel<sup>2</sup>, Olaf Spinczyk<sup>3</sup>

**Abstract:** Model-driven software development using language workbenches like JetBrains MPS provide many advantages compared to traditional software development. Base languages can be incrementally extended to increase the abstractness up to domain-specific languages (DSLs). Changes can be performed more efficiently in problem-oriented language extensions or DSLs, than in a base language. In addition, formal analysis can be performed on abstract models. To benefit from the model-driven approach, non-model-based legacy code has to be reusable and transformable to language extensions and DSLs. For the development of embedded systems, mbeddr provides a C99-like base language and extensions for MPS, such as mathematical symbols and state machines. This paper presents a case study that shows how many legacy C code fragments of three automotive series projects could be replaced by mbeddr language extensions. Furthermore, a proof of concept shows the feasibility of fraction and foreach loop refactorings. This work is a first approach for future language extension refactorings.

**Keywords:** Case Study, MPS, mbeddr, Automotive, Embedded Systems, Model-Driven Software Development, Legacy C Code, Refactoring, Restructuring, Reverse Engineering, Reengineering

## 1 Introduction

While the software architecture of automotive embedded systems is commonly modeled using UML, different implementation techniques exist. Code generation from UML models and/or graphical modeling tools, such as Matlab/Simulink, hides implementation details from the developer, and thus raises the level of abstraction. To gain transparency and control [Gr05] over registers, memory, runtime and synchronization, especially time-critical software and basic software modules are typically implemented using a low-level programming language, such as C. However, the implementation of architectural and domain-specific concepts is time-consuming and error-prone in such languages. Therefore, during the past few years there have been efforts to bridge the gap between abstract modeling and low-level programming using extensible languages that provide multiple levels of abstraction within

---

<sup>1</sup> Behr-Hella Thermocontrol GmbH, Hansastraße 40, 59557 Lippstadt, andreas.grosche@bhtc.com

<sup>2</sup> FH Dortmund, Sonnenstraße 96, 44139 Dortmund, igel@fh-dortmund.de

<sup>3</sup> TU Dortmund, Otto-Hahn-Str. 16, 44221 Dortmund, olaf.spinczyk@tu-dortmund.de

the same program. An important step in this direction has been taken with the JetBrains Meta Programming System (MPS)<sup>4</sup> and mbeddr<sup>5</sup>, which will be described in the following.

Traditional integrated development environments (IDEs) provide an editor to modify the source code of a program as plain text as shown in Fig. 1a [Ca16]. To support advanced IDE features, such as refactoring and navigation, the source code is divided into tokens that are further parsed to build up abstract syntax trees (ASTs) in a similar way as compilers work. A build system invokes a compiler to build the executable output.

In contrast to traditional IDEs, JetBrains MPS enables model-driven software development (MDSD). Instead of modifying text files, the user directly edits the ASTs of the program using a projectional editor (see Fig. 1b). An editor definition for each AST node defines, how the node is presented using a concrete syntax. Different editor definitions for the same AST nodes can provide textual or graphical views and reveal different information of the same model without the need of a tokenizer or parser.

MPS is a language workbench [Vo14] that provides a general purpose *base language*. This language can be incrementally extended by user-defined language extensions that abstract common and domain-specific code fragments. Different abstraction levels right up to domain-specific languages (DSLs) can be realized within the same program. During the build process, a generator transforms the DSLs and language extensions into the base language that is further transformed to text, such as Java code or XML. A build system can invoke a compiler to build executable code.

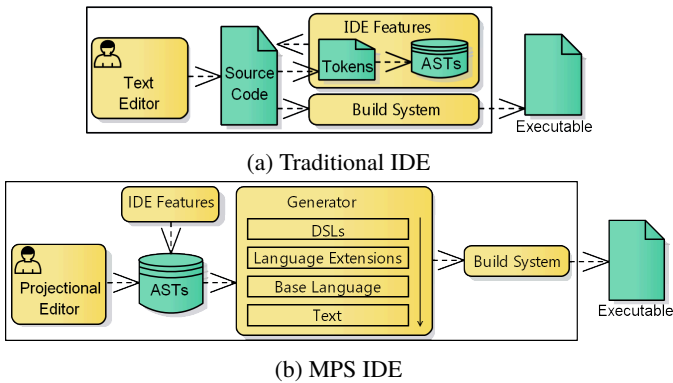


Fig. 1: Workflows of traditional IDEs and the MPS IDE.

The open source project mbeddr [Vo12] (primarily developed by itemis and fortiss) is based on MPS and provides a base language, which is similar to C99, for the development of embedded systems. The language extensions supplied with mbeddr, such as mathematical symbols and physical units, can be further extended by user-defined languages and DSLs

<sup>4</sup> <https://www.jetbrains.com/mps/>

<sup>5</sup> <http://mbeddr.com/>

to close the gap between high-level modeling and low-level programming languages. For instance, the language extension for state machines of mbeddr uses the full power of projectional editing. It now serves as an introductory example:

State machines can be modeled graphically, textually or as a table. The graphical projection of a state machine is shown in Fig. 2a. On reception of the *evtWrite* event, a transition to the *Writing* state is performed. On entry of that state, a function is called that writes the passed byte 12u via a universal asynchronous receiver/transmitter (UART). The state machine rests in the *Writing* state until the event *evtTxISR* is received. Fig. 2b shows the textual projection of this state machine.

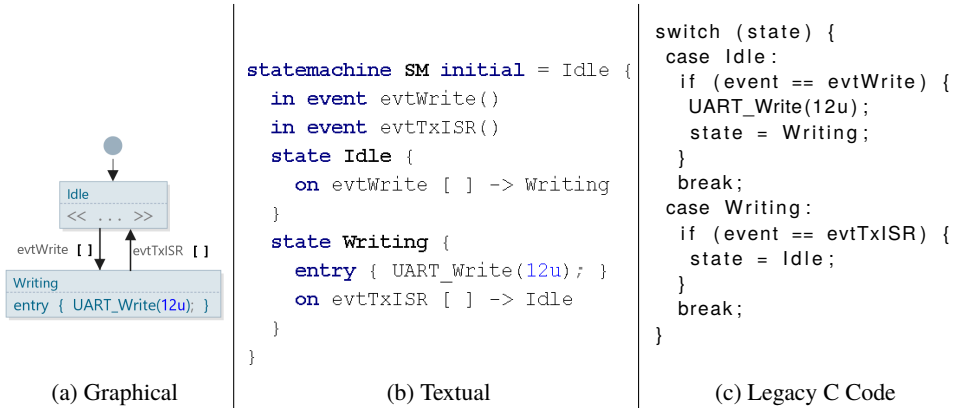


Fig. 2: The graphical and textual projections of an mbeddr state machine focus on the problem to be solved and hide implementation details.

Introducing mbeddr into the embedded software domain offers several advantages compared to traditional software development processes [Vo13b]:

- Projectional editors allow syntax that cannot easily be parsed by a traditional text-based IDE, such as mathematical symbols, tables and diagrams.
- The language extensions supplied with mbeddr can be further extended. The modular approach allows the combination of languages from different abstraction levels within the same program and even translation unit. For example, the C-like base language can be used for the definition of state machine actions (see Fig. 2a).
- Changing the structure of a software is simpler using higher-level abstractions than changing low-level C code, because implementation details are hidden.
- Automated domain-specific C verification [MVR14] closes the gap between C verification tools, such as CBMC [CKL04], and domain-specific language extensions, such as state machines and decision tables.

- Specifying requirements and unit tests using mbeddr allows the creation of links to achieve traceability between requirements, code fragments and verifications.

For an automotive company like Behr-Hella Thermocontrol GmbH (BHTC), it is common practice to reuse legacy C code in new projects, because *redevelopment* would be too expensive and time-consuming. The company itemis<sup>6</sup> developed a commercial importer based on TypeChef [Käl11] that allows the import of legacy C code into mbeddr. However, the imported code makes primarily use of the C99-like base language. To benefit from the advantages of the model-driven approach, legacy C code fragments have to be replaced by corresponding language extensions after the import. For example, the legacy C code shown in Fig. 2c should be replaced by an mbeddr state machine (see Fig. 2a). This would improve maintainability because the states, transitions and actions for different events could be modified without thinking about how state transitions and event handling are realized. In addition, automated verifications could be applied to the state machine.

To avoid time-consuming and error-prone manual replacements of base language code with language extensions, refactorings have to be developed. The complexity of such refactorings depends on the gap between the base language and the corresponding language extension in terms of abstraction levels. Regarding state machines, several implementation techniques exist, such as using `switch` (see Fig. 2c) statements or complex frameworks [Sa09]. The refactoring of language extensions with a small abstraction gap may be fully automated, while approaches with user interaction or machine learning may be required for extensions with larger abstraction gaps, such as state machines.

Language extension refactorings could also help developers to learn how and when to use language extensions. For example, when a developer writes a code fragment and the IDE would propose to replace that fragment with more abstract code using a language extension, the developer would learn when to use that language extension without the need of reading the language manual.

As languages evolve over time, it is likely that new language extensions will be added to mbeddr in the future. In addition, new language extensions can be developed by the user. Semi-automatic language extension refactorings could help to modify existing mbeddr code to use the new language extensions.

This paper presents a case study that shows the amount of code fragments in automotive legacy C code that could be replaced by language extensions supplied with mbeddr to assess the importance of language extension refactorings. We also provide a proof of concept to show that, at least in many cases, fully automated refactorings are possible.

The case study and the examined language extensions are introduced in Sect. 2. The proof of concept is shown in Sect. 3. After the presentation of related work in Sect. 4, we conclude and show future work in Sect. 5.

---

<sup>6</sup> <https://www.itemis.com/>

## 2 Case Study

The goal of this case study is to investigate the actual number of code fragments where our automotive series projects could benefit from the language extensions provided by mbeddr. Only handwritten C code in .c and .h files is part of this study. Generated code, such as from Matlab/Simulink models, is out of the scope of this case study, because refactoring such code should be done in Matlab/Simulink instead of the generated code. We introduce the analyzed projects in Sect. 2.1 before we present the applied method in Sect. 2.2. The examined language extensions are introduced in Sect. 2.3 and the results are discussed in Sect. 2.4.

### 2.1 Context

Our case study comprises three automotive projects with different characteristics as shown in Tab. 1. Project *A* contains the software for the main controller of a human-machine interface (HMI) control panel featuring a touch screen, capacitive buttons and acoustic feedback. Project *B* contains the software for the main controller of a center infotainment display (CID) featuring a touch screen, force sense and acoustic feedback. Project *C* contains the software for a CID slave controller that processes a tactile feedback.

Tab. 1: Automotive series projects analyzed in this case study.

Project	Status	Microcontroller	Standard	Files	Analyzed Files	Analyzed SLOC
A	Series	32 bit, 80 MHz	AUTOSAR	1,189	707 (289 .c and 418 .h)	238,651
B	Pre-Series	32 bit, 80 MHz	AUTOSAR	1,194	734 (283 .c and 451 .h)	222,203
C	Development	16 bit, 32 MHz	non-AUTOSAR	287	253 (101 .c and 152 .h)	15,390

Projects *A* and *B* are developed according to AUTOSAR [AU14]. The development phase of project *A* is finished, whereas project *B* is in the pre-series development phase. Both target a 32 bit microcontroller running at 80 MHz. For project *A*, we analyzed 707 handwritten .c and .h files containing 238,651 source lines of code<sup>7</sup> (SLOC). For project *B*, we analyzed 734 handwritten .c and .h files containing 222,203 SLOC. The software for project *C* is not developed according to AUTOSAR. The target is a 16 bit microcontroller running at 32 MHz. This project is under rapid development and currently comprises 253 handwritten .c and .h files containing 15,390 SLOC that we analyzed.

### 2.2 Method

To find code fragments that could be replaced by mbeddr language extensions in the source code of the analyzed projects, a semi-automatic approach has been chosen. In the first step, we had to select all handwritten .c and .h files from the projects to be analyzed. We did

<sup>7</sup> According to Count Lines of Code (CLOC), <http://cloc.sourceforge.net/>

this by scanning the files for specific keywords in the comment header of each file that are typical for handwritten and generated files. Only a few files had to be classified manually. The second step was to define code patterns for each mbeddr language extension. A code fragment is considered to be replaceable with a language extension, if the corresponding pattern matches. We used Coccinelle [Pa08] to match the patterns against the examined legacy C code. We could not specify exact patterns for all language extensions because of their complexity. That is why we had to verify and post-filter some of the results manually to avoid false positives.

List. 1 shows an excerpt of a Coccinelle pattern that finds all `for` loops in the analyzed code that fulfill the following requirements: In the initialization, an identifier has to be initialized with an arbitrary expression. This identifier has to be less than an arbitrary expression to enter the loop body. The same identifier has to be incremented on each iteration. The loop body may contain arbitrary statements. As `i` is a meta variable of Coccinelle the identifier can have an arbitrary name.

```
@@ identifier i; @@
* for (i = ...; i < ...; i++)
{
    ...
}
```

List. 1: Excerpt of the Coccinelle pattern that we used to find code fragments that could be replaced by `for` range loops.

## 2.3 Language Extensions

An overview of the examined language extensions is presented in Tab. 2. We give only a small introduction while Voelter et al. [Vo13a] present more details. The second column shows the language extensions as they can be edited in MPS. The third column shows examples of legacy C code in MPS that could be replaced by the corresponding extension. Except for error handling, which is explained later, the examples shown in the legacy C code column are conceptually similar to the code generated by MPS. We have subdivided the extensions into three categories:

**Syntactic Sugar:** The projectional editor allows mbeddr to provide language extensions for graphical mathematical symbols. We grouped them to fractions, mathematical functions as well as products (of sequences) and sums. **Fractions** can be used as an alternative syntax for divisions. To find code fragments replaceable by fractions, we used a Coccinelle pattern that looks for expressions that are divided by another expression.

Symbols for **mathematical functions** encapsulate calls to functions such as `abs`, `log`, `pow` and `sqrt` for calculating absolute values, logarithms, powers and square roots. To find code

Tab. 2: Screenshots of mbeddr language extensions and corresponding legacy C code in the MPS IDE.

Lang. Ext.	mbeddr Extension Code	Legacy C Code
Fraction	$z = \frac{x}{y + 1};$	<code>z = x / (y + 1);</code>
Math. Function	$z =  x ; z = \log_y x;$ $z = x^y; z = \sqrt{x + y};$	<code>z = abs(x); z = log(x) / log(y);</code> <code>z = pow(x, y); z = sqrt(x + y);</code>
For Range	<code>for (i ++ in [5..20]) { array[i] = i; }</code>	<code>for (uint8 i = 5; i &lt; 20; i++) { array[i] = i; }</code>
Foreach	<code>foreach (frame sized FRAME_SIZE as it) { checksum += it; }</code>	<code>for (uint8 i = 0; i &lt; FRAME_SIZE; i++) { checksum += frame[i]; }</code>
Product and Sum	$z = \prod_{i=1}^{10} x + 1; z = \sum_{i=1}^{10} x + 1;$	<code>z = 1;</code> <code>for (int32 i = 1; i &lt;= 10; i++) { z *= x + 1; }</code>
Phys. Unit	<code>uint8/N/ f = 10 kg * <math>\frac{10 \text{ m}}{5 \text{ s}^2};</math></code> <small>Error: type (uint8    int8 )/kg/ is not a subtype of uint8 /N/</small> <code>f = 5 kg;</code>	<code>uint8 fInNewton = 10 * (10 / 5);</code> <code>fInNewton = 5;</code>
Error Handling	<pre>@errors E_UNINITIALIZED uint8 GetBrightness2() {     if (_state == Uninitialized) {         error E_UNINITIALIZED;     }     return _brightness; }  try {     uint8 b = GetBrightness2();     // do something with b } when E_UNINITIALIZED {     // error handling }</pre>	<pre>Std_ReturnType GetBrightness1(uint8* B) {     if (_state == Uninitialized) {         return E_NOT_OK;     } else if (B == NULL) {         return E_NOT_OK;     }     *B = _brightness;     return E_OK; }  uint8 b; if (GetBrightness1(&amp;b) == E_OK) {     // do something with b } else {     // error handling }</pre>
State Machine	See Fig. 2a	See Fig. 2c

fragments that could be replaced by such symbols, we looked for calls to functions with identifiers that contain `abs`, `log`, `pow` and `sqrt`. This also includes variants such as `labs` and user-defined implementations with a similar naming.

**Enriched Syntax:** To simplify `for` loops that are incremented or decremented by  $I$  on each iteration, `mbeddr` provides **for range** loops as shown in Tab. 2. Only the counter variable name, the minimum and the maximum (exclusive) have to be specified. The type of the counter variable, the compare operator in the condition as well as the increment of the counter variable are omitted. The `++` can be replaced by `--` to iterate backwards over the specified range. To find code fragments replaceable by `for range` loops, we looked for `for` loops that assign an arbitrary expression to a counter variable. This variable has to be used in the condition with an appropriate greater-than or less-than comparison. In addition, it has to be incremented or decremented in the iteration by  $I$ . Since we analyzed projects using C90 that requires the definition of the counter variable at the start of a block (e.g., a function body) instead of in the `for` loop itself, we further had to analyze the usage of the variable before and after the `for` loop.

The **foreach** language extension of `mbeddr` can be used to iterate through arrays. The `it` expression can be used to read or write the current array element. Tab. 2 shows an example of a `foreach` loop that iterates over the `frame` array with an array length of `FRAME_SIZE`. In the body, the current element `it` of the array is added to the `checksum` variable. To find code fragments that could be replaced by `foreach` loops, we looked for `for` loops where a counter variable is set to 0, compared within an appropriate condition and incremented by  $I$ . In addition, the counter variable has to be used in the body as an index to an array. It can be used multiple times, but only for the same array. Furthermore, calculations such as adding an offset to the counter variable must not be performed. As for `for range` loops, we had to analyze the context of the `for` loop, because the analyzed projects use C90.

**Products and sums** are expanded to `for` loops during the code generation. In the analyzed code, we looked for `for` and `while` loops that add or multiply an expression to an identifier using statements, such as `x = x + ...;` or the short form `x += ...;`.

**Enriched Semantic:** The **physical units** extension allows the annotation of types and literals with unit information. The force  $F$  in newton ( $N$ ) is defined as  $F = m \cdot a$ . Annotating the type of the variable `f` with the unit  $N$  requires assigned expressions to evaluate to the corresponding unit. Trying to assign an expression that evaluates to another unit, e.g.,  $kg$ , leads to an error message in the IDE as shown in Tab. 2. Units are evaluated in the model and do not add any overhead to the generated C code. As shown in the legacy C code column, developers sometimes append a suffix containing the unit to the identifiers. To find code fragments that could be annotated with physical units, we looked for macro and variable definitions containing identifiers that contain `us`, `ms`, `clk`, `Hz`, `freq`, etc. and further analyzed the context.



The common way in AUTOSAR to inform the caller of a function about an error is to return a value of the type `Std_ReturnType` [AU15] as shown in the simplified example in the legacy C code column of Tab. 2. The standard values are `E_OK` for success and `E_NOT_OK` for errors. These can be extended by user-defined values. The caller has to evaluate the return value and perform an error handling. As a consequence, call-by-reference has to be used to get values from functions like getter functions.

As shown in Tab. 2, `mbeddr` provides sophisticated **error handling** in the style of Java or C++ exceptions. The example shows a getter function that is annotated with the errors that are thrown in the function body using the `error` statement. The getter function is called in a `try` block. The execution of the statements in the `try` block is aborted as soon as a function throws an error, which is caught in the corresponding `when` block. During the code generation, this error handling is expanded to `goto` statements and an error function argument (call by reference). Therefore, the overhead is comparable to the AUTOSAR approach. To identify code fragments replaceable by `mbeddr` error handling, we looked for functions that return a value of type `Std_ReturnType`.

State machines are also examined but not shown in the table, because they have already been introduced in Sect. 1. To find state machines in the analyzed C code, we focused on `switch` and `if` statements. We further examined these constructs manually with typical implementations of state machines [Sa09] in mind, such as using a variable that holds the current state (see Fig. 2c).

The presented language extensions have been ordered by their level of abstraction. *Syntactic sugar* language extensions are simple alternative syntactic representations compared to the respective legacy C code. They do not provide notable abstractions but still improve readability. *Enriched syntax* language extensions provide in-place substitutions to simplify compound expressions or statements. They provide simple abstractions that hide implementation details and possibly provide declarative flavor. *Enriched semantic* language extensions extend the type system, provide meta information, abstract data or control flow or have a cross-cutting character. Members of all categories can make use of graphical representations to further improve readability and maintainability.

## 2.4 Results

The results of this case study are presented in Tab. 3. Each row contains the number of code fragments that could be replaced by the different `mbeddr` language extensions of one analyzed project. Except for the mathematical functions, products and sums, we got two-digit and three-digit numbers of possible replacements. The larger projects *A* and *B* generally contain more replaceable code fragments than the smaller project *C*. The details for the different language extensions are discussed in the following.

Tab. 3: Number of possible replacements of C code fragments with mbeddr language extensions in three automotive series projects.

Project	Fraction	Math. Func.	For Range	ForEach	Product	Sum	Physical Unit	Error Handling	State Machine
A	150	0	483	112	6	2	54	323	58
B	243	0	462	86	6	2	45	191	75
C	16	0	40	24	0	6	69	87	21

The MISRA C:2012<sup>8</sup> rule 12.1 advises to use parentheses to make the operator precedences of C expressions explicit [MI13]. In practice, this leads to extensive use of functionally unnecessary parentheses to reduce possible human mistakes with the precedence rules of C. An example of the analyzed code is shown in Fig. 3. The first line shows a simplified version of a statement of the analyzed code with parentheses around the divisions to meet the MISRA C:2012 rule 12.1. The second line shows the same statement using **fractions**. This statement is much more readable and, in our opinion, the parentheses around the fractions can be omitted, because the precedence is obvious due to the graphical notation. In the generated C code, parentheses are inserted to meet the MISRA rule.

```
max = (1732u / fCLK) + 36u + (((4351u / fCLK) + 7324u) * blk) + (((184u / fCLK) + 44u) * n);

max =  $\frac{1732u}{fCLK} + 36u + ((\frac{4351u}{fCLK} + 7324u) * blk) + ((\frac{184u}{fCLK} + 44u) * n);$ 
```

Fig. 3: A statement without and with using fractions in MPS.

The small amount of code fragments replaceable by symbols for **mathematical functions**, **products** and **sums** can be explained by the systematic use of Matlab/Simulink for signal processing in the analyzed projects. The code generated by Matlab/Simulink is not part of the analysis. The found code fragments replaceable by sums are used to calculate simple checksums for inter-processor-communications. The calculations of products are used to convert raw bus signals to SI values and vice versa.

For projects A and B, the **for range** loop is the most usable language extension. The abstraction gap of for range loops is pretty small and the use of a for loop with a loop variable that is incremented or decremented by 1 is very common.

Some of the found for loops that could be replaced by **foreach** loops could also be replaced by sums. An example is the primitive checksum calculation over an array (e.g., bytes received via a serial communication) as shown in Tab. 2. As we further analyzed non-matching for loops, we found loops that iterate over multiple arrays at the same time as shown in a simplified version in Fig. 4a. These arrays store different information for the same entity. In this example, one array stores the information whether each UART peripheral is enabled and another array stores the current state of each UART peripheral. The counter variable of

<sup>8</sup> Guideline for the use of the C language in critical systems published by the Motor Industry Software Reliability Association (MISRA).

the `for` loop is used as an index for both arrays. In addition, the counter variable is passed to the `SendByte` function that uses the argument to access other arrays that store information about the UART peripherals.

<pre>for (uint8 i = 0; i &lt; UARTS; i++) {     if (uartEnabled[i] &amp;&amp; uartState[i] == 1u) {         SendByte(i);     } }</pre>	<pre>foreach (UARTS sized UARTS as it) {     if (it.Enabled &amp;&amp; it.State == 1u) {         SendByte(&amp;it);     } }</pre>
(a) Without OO	(b) With OO

Fig. 4: A `for` loop without and a `foreach` loop with object-oriented flavor in MPS.

The attributes of all UARTs could be restructured from multiple arrays of primitive data types to one array of structures. Each structure could hold all attributes of a UART in an object-oriented way. The result would be a `for` loop that iterates over one array. This loop could be replaced by a `foreach` loop as shown in Fig. 4b. The attributes of the iterated UARTs could then be accessed using the `it` expression and a reference to `it` could be passed to a function that could access the attributes of the UART in a convenient way.

We did not include this kind of `for` loops in the results, because more sophisticated analyses would be necessary. Further replacements by `foreach` loops would be possible looking for `while` and `do...while` loops with appropriate data flow analysis to match the preconditions for the counter variable incrementation.

More code fragments could be replaced by **physical units**, if signal processing would not be done in Matlab/Simulink. However, physical units do not only make sense in signal processing but also in basic software, e.g., for calculations of times and frequencies. It is notable that more physical units could be used in the small non-AUTOSAR project *C* than in the larger AUTOSAR projects *A* and *B*. This can be explained by the workflow of code generators that directly calculate register values to configure the hardware in the AUTOSAR projects. To allow a convenient hardware configuration in project *C* despite the lack of code generators, calculations of register values are done in C code to allow the specification of the hardware configuration parameters in physical units, such as Hz and ms.

A variety of approaches exist to implement **state machines** [Sa09]. In the analyzed code, we found primarily simple approaches, which use `switch` or `if` statements to determine the current state using a state variable. Depending on the state, corresponding actions are implemented and the state variable is reassigned to switch to the next state. We also found more sophisticated approaches using tables as well as code fragments that were not designed with a state machine in mind that could be restructured to get a well-designed state machine.

Although we used complex patterns to compensate implementation variations, such as using `switch` or `if` for state machines, the results shown in Tab. 3 are pessimistic. More relaxed patterns in combination with program and data flow analysis would reveal more refactoring possibilities. For example, we assumed that the power is calculated using the `pow` function.

However, programmers also use multiplications, such as  $x = y * y$ , that may even be split into multiple statements.

Tab. 4 shows the minimum and maximum SLOC of C code that we found in the examined projects that could be replaced with language extensions. To replace divisions by **fractions** and for loops by **for range** loops, only one line of code has to be modified. Especially for fractions, multiple divisions may occur in one line. The replacement of for loops with **foreach** loops requires the modification of the loop header and each array access using the loop variable in the body. The code fragments of the examined projects that could be replaced by **products** or **sums** are similar to the legacy C code outlined in Tab. 2.

The number of affected SLOC for replacements with physical units, error handling and state machines have been determined statistically using three exemplary translation units. For **physical units**, we considered the definition (e.g., of a variable) and the usages. For the context of each usage, we recursively analyzed which identifiers and literals are involved and how appropriate units can be applied to them. The SLOC for **error handling** is the sum of lines to be modified in the function that emits an error and the error handling of all calling functions. For **state machines**, we counted the lines of code required for the definition and implementation of the states and transitions.

Tab. 4: Minimum and maximum SLOC of C code replaceable with mbeddr language extensions in three automotive series projects.

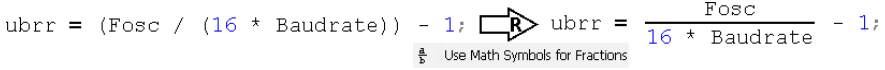
Project	Fraction	Math. Func.	For Range	For-each	Product	Sum	Physical Unit	Error Handling	State Machine
A	1	0	1	2 ... 10	4	4	1 ... 16	2 ... 9	36 ... 215
B	1	0	1	2 ... 10	4	4	1 ... 22	2 ... 30	18 ... 187
C	1	0	1	2 ... 4	0	4	1 ... 28	2 ... 9	67 ... 161

In general, larger abstraction gaps between language extensions and the base language require more complex patterns to find possible replacements. For example, the most complex Coccinelle pattern for the examined *syntactic sugar* language extensions took seven lines of code and no manual post-filtering was needed. In contrast to that, the *enriched semantic* language extensions required up to 234 lines of Coccinelle pattern and manual post-filtering.

### 3 Proof of Concept

As a proof of concept, we implemented refactorings that transform divisions to fractions and for loops to foreach loops in MPS. We used a simple approach that relies on the MPS base language with predefined extensions for model queries and transformations. The applicability of a refactoring is determined recursively using a set of preconditions. If all preconditions match, a model transformation is performed to replace divisions with fractions or for with foreach loops keeping the required properties and child elements.

Fig. 5 shows a statement for the calculation of the USART baudrate register of an Atmel ATmega8 microcontroller. Performing the refactoring for fractions on this statement replaces the division with a fraction bar. In addition, the parentheses around the fraction and around the denominator are removed because they are not required anymore.

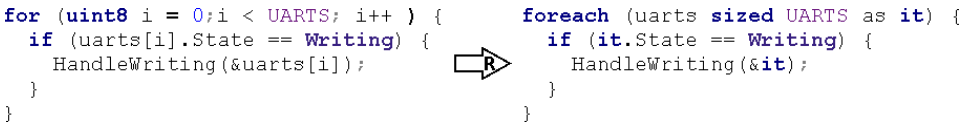


```
ubrr = (Fosc / (16 * Baudrate)) - 1; → ubrr =  $\frac{Fosc}{16 * Baudrate}$  - 1;
```

Use Math Symbols for Fractions

Fig. 5: Screenshots of the ATmega8 USART baudrate register calculation before and after the fractions refactoring in MPS.

An example for the `foreach` refactoring is shown in Fig. 6. The `for` loop is replaced with a `foreach` loop. The upper bound `UARTS` of the condition is reused in the new loop and all occurrences of an array indexed by the loop variable are replaced by the `it` expression. The number of preconditions to be checked for this refactoring is very large compared to the fractions refactoring, because many variations can occur. For example, one precondition ensures that the loop index is incremented by 1 on each iteration. To check this single precondition, multiple variations of the incrementation, such as `i++`, `++i` and `i = i + 1`, have to be considered. Using data and control flow analysis could additionally enable the replacement of `while` and `do..while` loops.



```
for (uint8 i = 0; i < UARTS; i++) {  
  if (uarts[i].State == Writing) {  
    HandleWriting(&uarts[i]);  
  }  
}  
→  
foreach (uarts sized UARTS as it) {  
  if (it.State == Writing) {  
    HandleWriting(&it);  
  }  
}
```

Fig. 6: Screenshots of a `for` loop before and after the refactoring to a `foreach` loop in MPS.

## 4 Related Work

Several case studies exist regarding the use of model-driven software development with `mbeddr` for embedded software. One case study deals with programming Lego Mindstorms robots based on an OSEK operating system [VKa] using appropriate language extensions with relevance beyond the Lego use case. Another case study approaches the first real-world project using `mbeddr` [VKb]. Further case studies show how language extensions affect the complexity, testability and runtime overhead of embedded software using `mbeddr`. They also show the effort for engineering a new project [Vo15, Vo17]. Vinogradov et al. [VOR15] describe the experience of using `mbeddr` in the railway domain including the integration of the model-driven approach into the traditional product lifecycle. All of these case studies conclude that language extensions and DSLs simplify reviews and the implementation of changes. Our case study is the first approach to evaluate the extent of applicability of `mbeddr` language extensions in existing automotive series projects.

Refactorings have been a research topic for several years [MT04] and most modern IDEs support basic refactorings. These traditional refactorings primarily focus on the structure of a software and improve it by restructuring, e.g., by moving statements into a new function

[FB13]. In contrast, language extension refactorings focus on the behavior and purpose of code fragments (considering different implementation techniques) and the replacement by more abstract language extensions.

Like the presented approach, reverse engineering aims at the automated comprehension of software to enhance development efficiency and maintainability. While classic reverse engineering creates representations at a higher level of abstraction without modifying the software [CC90], language extension refactoring *transforms* software fragments to a higher level of abstraction retaining the external behavior. Reverse engineering and design recovery techniques are a promising approach for the realization of language extension refactorings. However, language extension refactoring should not only recover design but also support the developer in improving the design and implementation which may require additional semantic analysis and input of domain-specific knowledge by the developer.

## 5 Conclusion and Future Work

Model-driven software development using mbeddr closes the gap between the programming language C, which is close to the hardware, and modeling languages, such as UML. Several case studies show the advantages of using mbeddr for the development of embedded software. Siemens PLM Software made use of these advantages and released the commercial LMS Imagine.Lab Embedded Software Designer (ESD) [Si] that is based on mbeddr.

The case study presented in this paper reflects the usefulness of the language extensions supplied with mbeddr in the automotive domain. It evaluates the number of possible replacements of legacy C code fragments by mbeddr language extensions in three automotive series projects. The extensions for products, sums and mathematical functions could rarely or not at all be applied. All other examined language extensions (e.g., fractions, for range loops and error handling) could be applied with a moderate to high degree. We could not specify sufficient preconditions to cover all the language extensions shipped with mbeddr, because of the high complexity of some of the language extensions, namely decision tables, interfaces and components. The missing extensions could be part of future case studies.

Since a refactoring must not alter the external behavior of a program while improving the internal structure [FB13], two major challenges have to be addressed for future language extension refactorings. The first challenge is to determine the applicability of a refactoring on a specific code fragment to ensure that the refactoring does not change the external behavior. The second challenge is the model-to-model transformation that replaces code fragments with more abstract language extensions. As a proof of concept, we implemented refactorings that transform divisions to fractions and for loops to foreach loops. Our implementation uses a simple approach that is sufficient for small abstraction gaps between the base language and the language extensions. However, further research is needed for refactorings of more abstract language extensions, such as state machines.

## References

- [AU14] AUTOSAR: AUTOSAR 4.2.1 – 054 – Main Requirements. AUTOSAR, Munich, Germany, 2014.
- [AU15] AUTOSAR: AUTOSAR 4.2.2 – 043 – General Requirements on Basic Software Modules. AUTOSAR, Munich, Germany, 2015.
- [Ca16] Campagne, Fabien: The MPS Language Workbench Volume I. Campagne Laboratory and CreateSpace Independent Publishing, New York, NY and North Charleston, South Carolina, third edition, version 1.5.1, march 2016 edition, 2016.
- [CC90] Chikofsky, E. J.; Cross, J. H.: Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, 1990.
- [CKL04] Clarke, Edmund; Kroening, Daniel; Lerda, Flavio: A Tool for Checking ANSI-C Programs. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pp. 168–176. Springer, 2004.
- [FB13] Fowler, Martin; Beck, Kent: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 2013.
- [Gr05] Grossman, Dan; Hicks, Michael; Jim, Trevor; Morrisett, Greg: Cyclone: A Type-Safe Dialect of C. *C/C++ Users Journal*, 23(1), 2005.
- [Kä11] Kästner, Christian; Giarrusso, Paolo G.; Rendel, Tillmann; Erdweg, Sebastian; Ostermann, Klaus; Berger, Thorsten: Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In: *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA '11)*. ACM, New York, NY, USA, pp. 805–824, 2011.
- [MI13] MIRA Limited: MISRA C:2012. MIRA Limited, Nuneaton, UK, 2013.
- [MT04] Mens, Tom; Tourwe, Tom: A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.
- [MVR14] Molotnikov, Zaur; Voelter, Markus; Ratiu, Daniel: Automated Domain-Specific C Verification with mbeddr. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ACM Press, New York, NY, USA, pp. 539–550, 2014.
- [Pa08] Padioleau, Yoann; Lawall, Julia; Hansen, René Rydhof; Muller, Gilles: Documenting and Automating Collateral Evolutions in Linux Device Drivers. In: *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Eurosys '08)*. ACM Press, New York, NY, USA, pp. 247–260, 2008.
- [Sa09] Samek, Miro: *Practical UML Statecharts in C/C++*. CRC Press, Boca Raton, 2. ed. edition, 2009.
- [Si] Siemens PLM Software: Delivering model-based software engineering for software-intensive systems: with LMS Imagine.Lab Embedded Software Designer.
- [VKa] Voelter, Markus; Kolb, Bernd: Lego Mindstorms: an mbeddr Case Study. [http://mbeddr.com/files/mbeddr\\_casestudy\\_mindstorms.pdf](http://mbeddr.com/files/mbeddr_casestudy_mindstorms.pdf).

- [VKb] Voelter, Markus; Kolb, Bernd: Smart Meter: an mbeddr Case Study. [http://mbeddr.com/files/mbeddr\\_casestudy\\_smartmeter.pdf](http://mbeddr.com/files/mbeddr_casestudy_smartmeter.pdf).
- [Vo12] Voelter, Markus; Ratiu, Daniel; Schaetz, Bernhard; Kolb, Bernd: mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems. In: Proceedings of the 2012 ACM Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH'12). Association for Computing Machinery, New York, NY, USA, pp. 121–140, 2012. <https://doi.org/10.1145/2384716.2384767>.
- [Vo13a] Voelter, Markus: DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. [dslbook.org](http://dslbook.org), 2013.
- [Vo13b] Voelter, Markus; Ratiu, Daniel; Kolb, Bernd; Schaetz, Bernhard: mbeddr: Instantiating a Language Workbench in the Embedded Software Domain. *Automated Software Engineering*, 20(3):339–390, 2013.
- [Vo14] Voelter, Markus: Generic Tools, Specific Languages. Ph.D. dissertation, Delft University of Technology, 2014.
- [Vo15] Voelter, Markus; van Deursen, Arie; Kolb, Bernd; Eberle, Stephan: Using C Language Extensions for Developing Embedded Software: A Case Study. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'15). SIGPLAN notices, ACM Press, New York, NY, USA, pp. 655–674, 2015.
- [Vo17] Voelter, Markus; Kolb, Bernd; Szabó, Tamás; Ratiu, Daniel; van Deursen, Arie: Lessons Learned from Developing mbeddr: A Case Study in Language Engineering with MPS. *Software & Systems Modeling (SoSyM)*, pp. 1–46, 2017.
- [VOR15] Vinogradov, Sergey; Ozhigin, Artem; Ratiu, Daniel: Modern model-based development approach for embedded systems: Practical Experience. In: Proceedings of the 2015 IEEE International Symposium on Systems Engineering (ISSE). pp. 56–59, 2015.