Preliminary Experience of using mbeddr for Developing Embedded Software

Markus Voelter

independent/itemis
Oetztaler Strasse 38
70327 Stuttgart
voelter@acm.org

Abstract: Over the last years, as part of the LW-ES KMU Innovativ research project, a team of developers at itemis and fortiss have developed the mbeddr system, which relies on language engineering to build a new class of environment for embedded software development. In essence, mbeddr consists of a set of extensions to C (such as state machines, units, interfaces and components) as well as a few additional languages for requirements engineering, documentation and product line engineering. mbeddr is still new, but a number of systems have been built with mbeddr. In this paper I summarize some preliminary experience with using mbeddr's default extensions to build embedded systems based on a set of case studies. The ability for mbeddr to be extended is *not* discussed in this paper, even though this has proven very useful as well.

1 About mbeddr

mbeddr[1] is an open source project supporting embedded software development based on incremental, modular domain-specific extension of C. It also supports other languages, for example, for capturing requirements, writing documentation that is closely integrated with code, and for specifying product line variability. Figure 1 shows an overview, details are discussed in [VRKS13] and [VRSK12]. mbeddr builds on the JetBrains MPS language workbench[2], a tool that supports the definition, composition and integrated use of general purpose or domain-specific languages. MPS uses a projectional editor, which means that, although a syntax may look textual, it is not represented as a sequence of characters which are transformed into an abstract syntax tree (AST) by a parser. Instead, a user's editing actions lead *directly* to changes in the AST. Projection rules render a concrete syntax *from* the AST. Consequently, MPS supports non-textual notations such as tables or mathematical symbols, and it also supports wide-ranging language composition and extension [Voe11] – no parser ambiguities can ever occur when combining languages.

mbeddr comes with an extensible implementation of the C99 programming language. On top of that, mbeddr ships with a library of reusable extensions relevant to embedded software. As a user writes a program, he can import language extensions from the library and use them in his program. The main extensions include test cases, interfaces and com-

[1] http://mbeddr.com
[2] http://jetbrains.com/mps

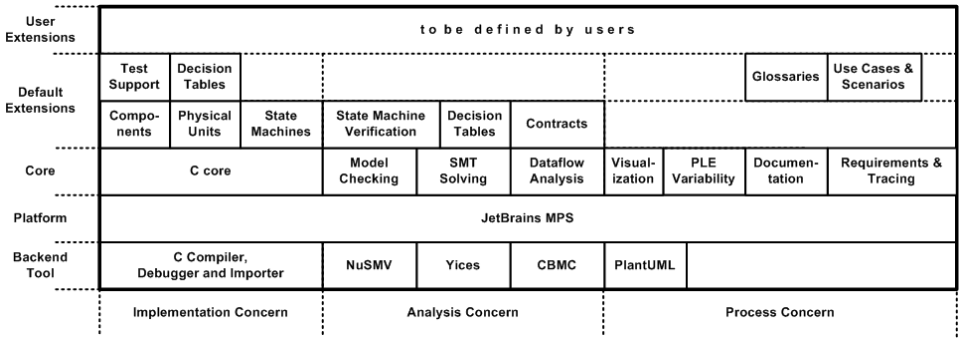

Figure 1: The mbeddr stack rests on the MPS language workbench. The first language layer contains an extensible version of C plus special support for logging/error reporting and build system integration. On top of that, mbeddr introduces default C extensions.

ponents, state machines, decision tables and data types with physical units. For many of these extensions, mbeddr provides an integration with static verification tools [RVSK12]. mbeddr also supports several important aspects of the software engineering process: documentation, requirements and product line variability. These are implemented in a generic way to make them reusable with any mbeddr-based language (we discuss aspects of the requirements support in detail in the remainder of this paper). Finally, users can build extensions to any of the existing languages or integrate additional DSLs.

2 Challenges

The goals of the mbeddr project are twofold: first and foremost, mbeddr is an industry-strength tool for developing embedded software. However, mbeddr is also a research vehicle for language engineering and language workbenches. In this paper we evaluate whether the language-oriented approach works for embedded software engineering[3].

To evaluate mbeddr's usefulness, this section introduces challenges in embedded software, derived from industry experience of the mbeddr team; however, the challenges are in line with those reported by other authors from different communities (representative examples are [SK01, Lee00, Lee08, Bro06, KMT12]). The remainder of this paper evaluates the experience of using mbeddr against these challenges.

Abstraction without Runtime Cost Domain-specific abstractions provide more concise descriptions of the system under development. Examples in embedded software include data flow blocks, state machines, or interfaces and components. For embedded software, where runtime footprint and efficiency is a prime concern, abstraction mechanisms are needed that can be resolved before or during compilation, and not at runtime.

C considered Unsafe C is efficient and flexible, but some features are considered unsafe. For example, unconstrained casting via `void` pointers, using `int`s as Booleans, the weak typing of `union`s or excessive use of macros can result in errors that are hard to track down. These unsafe features of C must be prohibited in safety-critical domains.

[3] As a consequence of limited space, this evaluation cannot go into too much detail. We refer the reader to the papers mentioned in the introduction for further details.

Program Annotations For reasons such as safety or efficiency, embedded systems often require additional data to be associated with program elements. Examples include physical units, coordinate systems, data encodings or value ranges for variables.

Static Checks and Verification Embedded systems often have to fulfil safety requirements. Standards such as ISO-26262, DO-178B or IEC-61508 require that for high safety certification levels various forms of static analyses are performed on the software. More and more embedded software systems have to fulfil strict safety requirements [LT09].

Process Support There are at least three process-related concerns relevant to embedded software development. First, many certification standards (such as those mentioned above) require that code be explicitly linked to requirements such that full traceability is available. Second, many embedded systems are developed as part of product-lines, where each variant consists of a subset of the (parts of) artifacts that comprise the product-line. This variability is usually expressed using the C preprocessor via `#ifdef`s. As a consequence, variant management is a huge source of accidental complexity. The third process-related concern is documentation. In most projects, various software architecture and design documents have to be created and kept in sync with the code. If they are created using Word or LaTeX, no actual connection exists between the documentation and the code. This is tedious and error prone. Better tool support is urgently required.

3 Example Systems and their Use of mbeddr

To validate the usefulness of mbeddr based on the challenges above, this paper relies on the experiences from a number of development projects run with mbeddr[4]. The projects range from demo applications to real-world development projects:

Smartmeter: The Smartmeter project is the first commercial use of mbeddr and targets the development of the software for a 3-phase smartmeter. The software comprises ca. 40,000 lines of mbeddr code, has several time-sensitive parts that require a low-overhead implementation and will have to be certified by the future operator. This leads to an emphasis on performance, testing, formal analyses and requirements tracing. The software exploits existing code supplied by the hardware vendor in the form of header files, libraries and code snippets, even though most of the system has been rewritten by now. Smartmeter is developed by itemis France.

Park-o-Matic: As part of the LW-ES project, BMW Car IT[5] has developed an AUTOSAR component based on mbeddr. This component, called Park-o-Matic[6], coordinates various sensors when assisting the driver to park the car. It is fundamentally a state-based system. As part of this project, AUTOSAR-specific generators had to be built for the mbeddr components language. The current mbeddr generators map the components to plain C. In case of AUTOSAR, components have to integrate with the runtime environment (RTE), which means, for example, that calls on required port have to be translated to AUTOSAR-specific macro calls. In addition, an XML file

[4] Separate documents for some of these can be found at `http://mbeddr.com/learn.html`
[5] `http://www.bmw-carit.com/`
[6] This is not the actual name; I was not allowed to use the real name in the thesis.

has to be generated that describes the software component so that it can be integrated with others by an integration tool.

Lego Mindstorms: This example looks at the first significant demonstration project built with mbeddr: a set of C extensions for programming Lego Mindstorms[7] robots. These robots can be programmed with various approaches and languages, among them C. In particular, there is an implementation of the OSEK[8] operating system called Lejos OSEK[9]. We have developed several robots (and the respective software) based on a common set of C extensions on top of Lejos OSEK. Since OSEK is also used outside of Lego Mindstorms for real-world embedded applications, this system has relevance beyond Lego. The system was developed by the mbeddr team.

Pacemaker: This system, developed by students at fortiss, addresses mbeddr's contribution to the Pacemaker Challenge[10], an international, academic challenge on the development and verification of safety-critical software, exemplified by a pacemaker. This system emphasizes code quality, verification techniques and systematic management of requirements. Performance is also important, since the software must run on the very limited resources provided by the microcontroller in a pacemaker.

4 Addressing the Challenges

As a means of evaluating mbeddr, this section revisits the challenges introduced in Section 2 and shows how mbeddr's features address these challenges.

4.1 Abstraction without Runtime Cost

This section investigates whether and how mbeddr's extensions are used in the example systems, and whether their overhead is acceptable.

Smartmeter: Smartmeter uses mbeddr's components to encapsulate the hardware-dependent parts of the system. By exchanging the hardware-dependent components with mocks, integration tests can be run and debugged on a PC without using the actual target device. While this does not cover all potential test and debugging scenarios, a significant share of the application logic can be handled this way. In particular, interfaces and components are used heavily to modularize the system and make it testable. 54 test cases and 1,415 assertions are used. Physical units are used heavily as well, with 102 unit declarations and 155 conversion rules. The smartmeter communicates with its environment via several different protocols. So far, one of these protocols has been refactored to use a state machine. This has proven to be much more readable than the original C code. The Smartmeter team reports significant benefits in terms of code quality and robustness. The developers involved in the project had been thinking in terms of interfaces and components before; mbeddr allows them to express these notions directly in code.

[7] http://mindstorms.lego.com/
[8] http://en.wikipedia.org/wiki/OSEK
[9] http://lejos-osek.sourceforge.net/
[10] http://sqrl.mcmaster.ca/pacemaker.htm

Park-o-Matic: The core of Park-o-Matic is a big state machine which coordinates various sensors and actuators used during the parking process. The interfaces to the sensors and actuators are implemented as components, and the state machine lives in yet another component. By stubbing and mocking the sensor and actuator components, testing of the overall system was simplified.

Lego Mindstorms: mbeddr's components have been used to wrap low-level Lego APIs into higher-level units that reflect the structure of the underlying robot, and hence makes implementing the application logic that controls the robot much simpler. For example, an interface `DriveTrain` supports a high-level API for driving the robots. We use pre- and post-conditions as well as a protocol state machine to define the semantics of the interface. As a consequence of the separation between specification (interface) and implementation (component), testing of line-following algorithms was simplified. For example, the motors are encapsulated into interfaces/-components as well. This way, mock implementations can be provided to simulate the robot without using the Mindstorms hardware and API. The top-level behavior of a line-follower robot was implemented as a state machine. The state machine calls out to the components to effect the necessary changes in direction or speed.

Pacemaker: The default extensions have proven useful in the pacemaker. Pacemaker uses mbeddr's components to encapsulate the hardware dependent parts. Furthermore, the pulse generator system is divided into subsystems according to the disease these subsystems cure. The pacemaker logic for treating diseases is implemented as a state machine. This makes the implementation easier to validate and verify (discussed in Section 4.4). Requirements tracing simplifies the validation activities.

Generating code from higher-level abstractions may introduce performance and resource overhead. In embedded software, this overhead must not be significant. It is not clearly defined what "significant" means; however, a threshold is clearly reached when a new target platform is required to run the software, "just because" better abstractions have been used to develop it, because this will increase unit cost. As part of mbeddr development, we have not performed a systematic study of the overhead incurred by the mbeddr extensions, but preliminary conclusions can be drawn from the existing systems:

Smartmeter: The Smartmeter code runs on the intended target device. This means the overall size of the system (in terms of program size and RAM use) is low enough to work on the hardware that had been planned for use with the native C version.

Pacemaker: The Pacemaker challenge requires the code to run on a quite limited target platform, the PIC18[11]). The C code is compiled with a proprietary C compiler. The overhead of the implementation code generated from the mbeddr abstractions is small enough so that the code can be run on this platform in terms of performance, program size and RAM use.

mbeddr's extensions can be partitioned into three groups. The first group has no consequences for the generated C code at all, the extensions are related to meta data (require-

[11] `http://en.wikipedia.org/wiki/PIC_microcontroller`

ments tracing) or type checks (units). During generation, the extension code is removed from the program.

The second group are extensions that are trivially generated to C, and use at most function calls as indirections. The resulting code is similar in size and performance to reasonably well-structured manually written code. State machines (generated to functions with `switch` statements), unit value conversions (which inline the conversion expression) or unit tests (which become `void` functions) are an example of this group.

The third group of extensions incurs additional overhead, even though mbeddr is designed to keep it minimal. Here are some examples. The runtime checking of contracts is performed with an `if` statements that check the pre- and post-conditions, as well as assignments to and checks of variables that keep track of the protocol state. Another example is polymorphism for component interfaces, which use an indirection through a function pointer when an operation is called on a required port.

In this third group of extensions there is no way of implementing the feature in C without overhead. The user guide points this out to the users, and they have to make a conscious decision whether the overhead is worth the benefits in flexibility or maintainability. However, in some cases mbeddr provides different transformation options that make different trade-offs with regards to runtime overhead. For example, if in a given executable, an interface is only provided by one component and hence no runtime polymorphism is required, the components can be connected statically, and the indirection through function pointers is not necessary. This leads to better performance, but also limits flexibility.

We conclude that mbeddr generates reasonably efficient code, both in terms of overhead and performance. It can certainly be used for soft realtime applications on reasonably small processors. We are still unsure about hard realtime applications. Even though Smartmeter is promising, more experience is needed in this area. In addition, additional abstractions to describe worst-case execution time and to support static scheduling are required. However, these can be added to mbeddr easily (the whole point of mbeddr is its extensibility), so in the long term, we are convinced that mbeddr is a very capable platform for hard realtime applications.

Summing up, the mbeddr default extensions have proven extremely useful in the development of the various systems. Their tight integration is useful, since it avoids the mismatch between various different abstractions encountered when using different tools for each abstraction. This is confirmed by the developers of the Pacemaker, who report that *the fact that the extensions are directly integrated into C as opposed to the classical approach of using external DSLs or separate modeling tools, reduces the hurdle of using higher-level extensions and removes any potential mismatch between DSL code and C code.*

4.2 C considered Unsafe

The mbeddr C implementation already makes some changes to C that improve safety. For example, the preprocessor is not exposed to the developer; its use cases (constants, macros, `#ifdef`-based variability, `pragma`s) have first-class alternatives in mbeddr that are more robust and typesafe. Size-independent integer types (such as `int` or `short`) can only be

used for legacy code integration; regular code has to use the size-specific types (`int8`, `uint16`, etc.). Arithmetic operations on pointers or `enum`s are only supported after a cast; and mbeddr C has direct support `boolean` types instead of treating integers as Booleans.

> **Smartmeter:** Smartmeter is partially based on code received from the hardware vendor. This code has been refactored into mbeddr components; in the process, it has also been thoroughly cleaned up. Several problems with pointer arithmetics and integer overflow have been discovered as a consequence of mbeddr's stricter type system.

More sophisticated checks, such as those necessary for MISRA-compliance can be integrated as modular language extensions. The necessary building blocks for such an extension are annotations (to mark a module as MISRA-compliant), constraints (to perform the required checks on modules marked as MISRA-compliant) as well as the existing AST, type information and data flow graph (to be able to implement these additional checks).

Finally, the existing extensions, plus those potentially created by application developers, let developers write code at an appropriate abstraction level, and the unsafe lower-level code is generated, reducing the probability of mistakes.

> **Smartmeter:** Smartmeter combines components and state machines which supports decoupling message parsing from the application logic in the server component. Parsing messages according to their definition is notoriously finicky and involves direct memory access and pointer arithmetics. This must be integrated with state-based behavior to keep track of the protocol state. State machines, as well as declarative descriptions of the message structure[12] make this code much more robust.

4.3 Program annotations

Program annotations are data that improves the type checking or other constraints in the IDE, but has no effect on the binary. Physical units are an example of program annotations.

> **Smartmeter:** Extensive use is made of physical units; there are 102 unit declarations in the Smartmeter project. Smartmeters deal with various currents and voltages, and distinguishing and converting between these using physical units has helped uncover several bugs. For example, one code snippet squared a temperature value and assigned it back to the original variable (`T = T * T;`). After adding the unit `K` to the temperature variable, the type checks of the units extension discovered this bug immediately; it was fixed easily. Units also help a lot with the readability of the code.

As part of mbeddr's tutorials, an example extension has been built that annotates data structures with information about which layer of the system is allowed to write and read these values. By annotation program modules with layer information, the IDE can now check basic architectural constraints, such as whether a data element is allowed to be written from a given program location.

In discussions with a prospective mbeddr user other use cases for annotations were discovered. Instead of physical units, types and literals could be annotated with coordinate systems. The type checker would then make sure that values that are relative to a local

[12] This is an extension that is being built as this thesis is written.

coordinate system and values that are relative to a global coordinate systems are not mixed up. In the second use case, program annotations would have been used to represent secure and insecure parts of a crypto system, making sure that no data ever flows from the secure part to the insecure part. Both customer projects did not materialize, though.

4.4 Static Checks and Verification

Forcing the user to use size-specific integer types, providing a `boolean` type instead of interpreting integers as Boolean, and prohibiting the preprocessor are all steps that make the program more easily analyzable by the built-in type checker. The physical units serve a similar purpose. In addition, the integrated verification tools provide an additional level of analysis. By integrating these tools directly with the language (they rely on domain-specific language extensions) and the IDE, it is much easier for users to adopt them.

Smartmeter: Decision tables are used to replace nested `if` statements and the completeness and determinism analyses have been used to uncover bugs. The protocol state machines are model-checked. This uncovered bugs introduced when refactoring the protocol implementation from the original C code to mbeddr state machines.

Pacemaker: The core behavior of the pacemaker is specified as a state machine. To verify this state machine and to prove correctness of the code, two additional C extensions have been developed. One supports the specification of nondeterministic environments for the state machine (simulating the human heart), and other one allows the specification of temporal properties (expressing the correctness conditions in the face of its nondeterministic environment). All three (state machine, environment and properties) are transparently translated to C code and verified with CBMC[13].

Park-o-Matic: It was attempted to verify various aspects of the state machine. However, this failed because the analyses were only attempted *after* the state machine was fully developed, at which point it was tightly connected to complex data structures via complex guard conditions. This complexity thwarted the model checker.

The overall experience with the formal analyses is varied. Based on the (negative) experience with Park-o-Matic and the (positive) experience with Smartmeter and Pacemaker, we conclude that a system has to be designed for analyzability to avoid running into scalability issues. In Park-o-Matic, analysis was attempted for an almost finished system, in which the modularizations necessary to keep the complexity at bay were not made.

4.5 Process Support

mbeddr directly supports requirements and requirements tracing, product-line variability and prose documentation that is tightly integrated with code. This directly addresses the three process-related challenges identified before. All of them are directly integrated with the IDE, work with any language extension and are often themselves extensible. For example, new attributes for requirements or new kinds of paragraphs for documents can be defined using the means of the MPS language workbench on which mbeddr relies.

[13] `http://www.cprover.org/cbmc/`

Smartmeter: Smartmeter uses requirements traces: during the upcoming certification process, these will be useful for tracking if and how the customer requirements have been implemented. Because of their orthogonal nature, the traces can be attached also to the new language concepts specifically developed for Smartmeter[14].

Pacemaker: Certification of safety-critical software systems requires requirements tracing, mbeddr's ubiquitous support makes it painless to use. Even though this is only a demo system for the Pacemaker Challenge, it is nonetheless an interesting demonstration how domain-specific abstractions, verification, requirements and requirements tracing fit together.

Lego Mindstorms: Lego being what it is, it is easy to develop hardware variants. We have used mbeddr's support for product-line variability to reflect the modular hardware in the software: sensor components have been statically exchanged based on feature models.

The requirements language has been proven very useful. In fact, it has been used as a standalone system for collecting requirements. Tracing has also proven to be useful, in particular, since it works out of the box with any language.

The documentation language has not been used much in the six example systems, since it is relatively new. However, we are currently in the process of porting the complete mbeddr user guide to the documentation language. The tight integration with code will make it very easy to keep the documentation in sync with an evolving mbeddr.

The experience with the product-line support is more varied. The definition of feature models and configuration works well (which is not surprising, since it is an established approach for modeling variability). The experience with mapping the variability onto programs using presence condition is mixed. It works well if the presence condition is used to remove parts of programs that are not needed in particular variants. However, once references to variable program parts get involved, the current, simple approach starts to break down. The same is true if the variability affects the type of program nodes. A variability-aware type system would be required. At this point it is not clear whether this is feasible algorithmically and in terms of performance. Also, without enhancements of MPS itself it is likely not possible to build such a variability-aware type system generically, i.e. without explicit awareness of the underlying language. This would be unfortunate, since extensions would have to built in a variability-aware way specifically.

5 Conclusion

In this paper I have reported on some preliminary experience with using mbeddr for developing embedded software. The results so far are promising, even though more research is required. Two perspectives are obvious. First, the impact of the ability to build domain-specific extensions "on the fly" needs to be evaluated, since in this scenario, the effort and complexity of building mbeddr extensions with MPS becomes important. So far, most

[14] An important aspect of mbeddr is that project-specific extensions can be developed easily. However, this aspect of mbeddr is not considered in this paper.

of the extensions (even those specific to Smartmeter) have been built by (or with support from) the mbeddr team. Second, the benefits of MPS for building flexible and extensible systems like mbeddr don't come quite for free: the MPS takes some time to get used to. We are currently running a survey among MPS and mbeddr users to find out more about this aspect. mbeddr has also been selected as the basis for a new embedded engineering tool by a major international tool vendor. While this is not a scientifically relevant result, it is certainly very encouraging to the mbeddr team.

Acknowledgements: Even though I am the author of this paper, mbeddr is a team effort with my colleagues Bernd Kolb, Dan Ratiu, Kolja Dumman and Domenik Pavletic. I also want to thank Stephan Eberle, Stefan Schmierer and Birgit Engelmann (for trying out mbeddr when it was not very mature yet) as well as the MPS team at JetBrains led by Alexander Shatalin (for tirelessly helping us with MPS issues and questions).

References

[Bro06] Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06. ACM, 2006.

[KMT12] Adrian Kuhn, GailC. Murphy, and C.Albert Thompson. An Exploratory Study of Forces and Frictions Affecting Large-Scale Model-Driven Development. In RobertB. France, Juergen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *LNCS*. Springer, 2012.

[Lee00] E.A. Lee. What's ahead for embedded software? *Computer*, 33(9):18–26, 2000.

[Lee08] E.A. Lee. Cyber Physical Systems: Design Challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369, 2008.

[LT09] Peter Liggesmeyer and Mario Trapp. Trends in Embedded Software Engineering. *IEEE Softw.*, 26, May 2009.

[RVSK12] Daniel Ratiu, Markus Voelter, Bernhard Schaetz, and Bernd Kolb. Language Engineering as Enabler for Incrementally Defined Formal Analyses. In *FORMSERA'12*, 2012.

[SK01] Janos Sztipanovits and Gabor Karsai. Embedded Software: Challenges and Opportunities. In ThomasA. Henzinger and ChristophM. Kirsch, editors, *Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*. Springer, 2001.

[Voe11] Markus Voelter. Language and IDE Development, Modularization and Composition with MPS. In *GTTSE 2011*, LNCS. Springer, 2011.

[VRKS13] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. mbeddr: Instantiating a Language Workbench in the Embedded Software Domain. *Journal of Automated Software Engineering*, 2013.

[VRSK12] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems. In *Proc. of the 3rd conf. on Systems, programming, and applications: software for humanity*, SPLASH '12, 2012.