MPS

mbeddr

Short description:

A set of integrated and extensible languages for embedded software engineering.

Problem

Most embedded systems companies implement embedded software in C[1]. C is good at low-level algorithms and produces efficient binaries, but it provides only limited support for defining custom abstractions.

This can result in code that is hard to understand, maintain, and extend. On the other hand, high-level modeling tools make it hard to effectively address the low-level aspects that are important for embedded software.

Solution

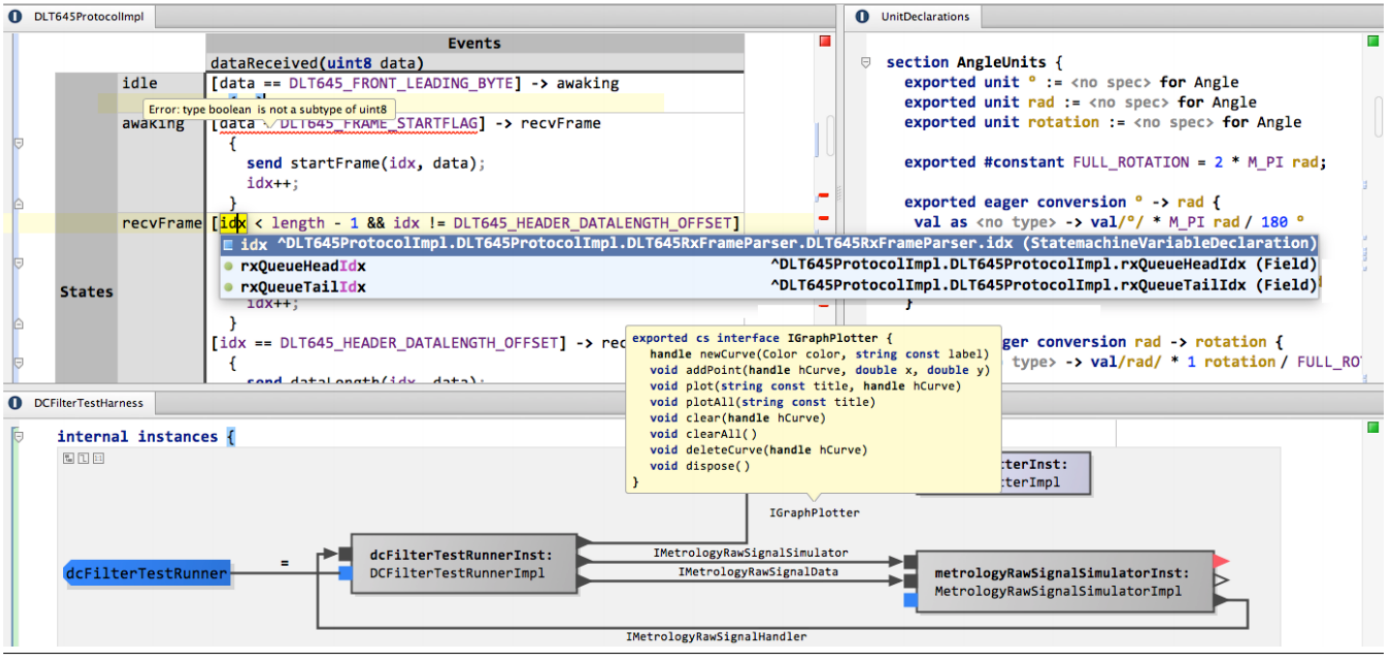
mbeddr is a platform built on top of JetBrains MPS that provides an extensible version of C plus a set of predefined extensions such as physical units, interfaces and components, state machines and unit testing. Since extensions are syntactically embedded in C programs, users can mix higher-level abstractions with low-level C code. Developers are not forced to use the extensions; they can choose to use them when they consider them appropriate.

The extensions facilitate strong static checking, improve readability, and help avoid low-level mistakes. Testing is also a challenge for embedded software because of hardware dependencies, restrictions on on-device debugging, and subtle timing and resource constraints. Embedded software often has only a few or no automated unit tests, which is a problem for quality, productivity, and its ability to evolve. mbeddr has first-class support for assertions, unit tests, and test suites.

The mbeddr languages use a wide variety of notations – textual, tabular, symbolic, and graphical – and the C extensions are modular; new extensions can be added without changing the existing implementation of C.

[1] C. Ebert and C. Jones.
Embedded software: facts, figures, and future.
Computer, 42(4), April 2009

What mbeddr looks like



The screenshot shows various parts of a Smart Meter implementation[2] using mbeddr: a part of the protocol parser state machine (top left), unit declarations (top right), and component wiring for a test case (bottom). It also illustrates how mbeddr provides IDE support for C and its extensions, including syntax highlighting, code completion, error markup, refactorings, quick fixes, and tooltips. The screenshot also showcases the support for mixed notations (text, tables, and diagrams).

Mbeddr is currently being used in a number of commercial products from companies.

Why MPS?

Itemis is the company behind the development of mbeddr.
The support of non-textual notations, language modularity, and language development are key elements to chose MPS to build mbeddr.

Language Development

Automated testing, continuous integration, and version control are crucial for any project to be integrated into an established development process.

mbeddr integrates a graphical user interface with a wide range of version control systems, git included, so users can pull, commit, or merge from within MPS. It also supports a headless mode to run tests or generate models. To initiate this, MPS provides ant[3] tasks. These tasks can be integrated with essentially all existing CI servers.

---

[2] M. Voelter, A. van Deursen, B. Kolb, and S. Eberle.
Using C language extensions for developing embedded software:
A case study. In OOPSLA 2015, 2015.

[3] S. Erdweg, T. Storm, M. Völter, et al.
The state of the art in language workbenches. In M. Erwig, R. Paige, and E. Wyk, editors,
Software Language Engineering, volume 8225 of LNCS. Springer, 2013.

Projectional Editor

The rich and composable ecosystem of C extension of mbeddr would not be possible using parser-based language engineering technology. The Projectional Editor is the distinguishing characteristic of MPS compared to most other industrial-strength language workbenches.

Language Modularity

An important part of MPS, and a cornerstone of modern language engineering in general, is language modularity and composition. It makes it possible to use a set of independently developed languages together, in one file or model. Language Modularity refers not just to syntax, but also to the type system, the semantics, and the IDE support.

Language modularization is essential for managing complexity in language development, just as it is essential for software engineering in general[4]: it helps break down a large language into several smaller ones that are easier to understand and maintain, it makes building and testing each language module faster, and it allows new languages (or language extensions) to be developed without changes to the other modular languages, and in particular, the base language (in case of extension).

Modularization is also important for language users, because they can use only those features (extensions) relevant to their particular task or skill level. This has turned out to be useful in mbeddr[5], because it allows mbeddr applications to be developed bottom-up: first implement C, and then, incrementally, add domain-specific abstractions as modular language extensions. Users can skip down to the next lower abstraction level if an abstraction is not suitable or if it introduced too much performance overhead (a very important concern in embedded software).

This document was almost entirely curated from the following sources:

- mbeddr website.
- Lessons Learned from Developing mbeddr
- Using C Language Extensions for Developing Embedded Software: A Case Study.

[4] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen.
The structure and value of modularity in software design.
In ACM SIGSOFT Software Engineering Notes, volume 26, pages 99–108. ACM, 2001.

[5] M. Voelter, A. van Deursen, B. Kolb, and S. Eberle.
Using C language extensions for developing embedded software:
A case study. In OOPSLA 2015, 2015.