# *Modern model-based development approach for embedded systems*

## Practical Experience

Sergey Vinogradov
Artem Ozhigin
Corporate Technology, Research and Technology Center
Siemens
Russian Federation
{Sergey.Vinogradov; Artem.Ozhigin}@siemens.com

Daniel Ratiu
Corporate Technology, Research and Technology Center
Siemens
Germany
Daniel.Ratiu@siemens.com

***Abstract*— Control functionality of modern rail vehicles is getting more and more complex. It contains several modules such as the traction control unit or the central control unit, as well as input and output stations, such as driver's cab terminals and process I/Os. A plethora of devices are connected to the vehicle and train bus and are able to communicate. The functions of the vehicle control and traction systems are configured by using function blocks from which loadable programs are generated. The languages used to program the control units are well established in the field. However, one-size-fits-all approach cannot adequately address the increased complexity of the software in modern trains. In this paper we describe our preliminary experience with using the multi-paradigm modeling tool "mbeddr" in the railway domain. The following aspects have been in focus during the work: a) matching the application requirements and domain specific language used for implementation; b) integration of model-based approach into traditional product lifecycle; c) reengineering existing functionality using modeling and code generation capabilities of mbeddr. The system example we chose was the application logic of automated train driving system implemented in development environment of Siemens process automation framework.**

***Keywords—model based development; language engineering***

## I. INTRODUCTION

Historically, many legacy train control systems had been developed using engineering frameworks which are dedicated for usage in electrical engineering domain. The logic is implemented by specifying and linking physical signals as they are present on circuit boards: input circuit blocks, routing of signals into parallel/subsequent "OR" and "AND" blocks to define the signals logic, combining common functional blocks into functional packages, etc. These functional packages are used then to generate loadable programs and execute them on target hardware units. Such an approach works well for signal processing logic where functions executions are not repeated often and need to guarantee certain time bounds. Moreover, the resultant "pictures of code" are comprehensible for electrical engineers. However, the increase in complexity of the control systems in trains puts certain challenges on such engineering environments; especially a part of software development requires serious adaptation. The functionality to be implemented in modern trains does not belong only to the traditional controls domain and has for example also data-processing intensive components.

On the one side, innovation cycle for train control networks and corresponding software in industry takes usually from 8 to 10 years. It affects the migration plans of industrial companies regarding bringing new engineering methods and technologies into practice. In these cases, systems engineers have to carefully think about providing the projects with well defined approach to specification, implementation and validation of new system functions as well as integration of the new functionality into the legacy system.

On the other side, models are not new in software development anymore. Over the past few decades, the software industry has seen numerous analysis and design methods, each with its own modeling approaches and notations. Better integration of such models and code should significantly increase the opportunity to perform changes via models, rather than simply modifying the code directly [1]. Furthermore, using appropriate modeling abstractions is an enabler for higher reuse and more manageable complexity. The use of adequate modeling approaches prevent common errors and ensure a high quality.

Our work is part of the continuous efforts of the Corporate Technology division of Siemens to evaluate modern engineering approaches which are able to cope with increasing complexity of our products and to allow their development in an efficient manner. In this paper we report on our experience with using mbeddr (www.mbeddr.com) open-source model-based development workbench for the railway domain applications. We focused on the following practical goals: effective capturing and modeling of domain-specific requirements, introducing state-of-the-art software engineering practices to cope with the complexity and using advanced quality assurance to ensure the integrity of software-relevant functions.

In Section II we present a brief overview of the mbeddr modeling stack. In Section III we illustrate the key practical experience on using mbeddr to program train control units and on integration of the new tooling with the existing engineering framework. Section IV provides the major lessons learned and Section V the conclusions.

## II. MULTIPARADIGM PROGRAMMING WITH MBEDDR

mbeddr (www.mbeddr.com) is an open source model-based development workbench for embedded systems based on language engineering technologies. Mbeddr provides an implementation of the C language and a set of language extensions specific to embedded systems like state machines, components, decision tables, mathematical notations. These extensions can be seamlessly combined with C code whenever needed [2]. Finally, regular C-code is generated out of the models. When used in combination, these extensions allow multi-paradigm programming: parts of the developed system can be written procedurally in C, other parts using state-machines and others use advanced modules like components. In addition to code development, the tool provides other domain specific languages like for example for the definition of requirements, product-lines or writing code documentation. By using appropriate constructs whenever needed, mbeddr favors writing of correct-by-construction software. The programs contain less boilerplate code and the implemented architecture and logic is explicitly visible in the code. Code parts can be easily traced to requirements models which they implement.

Besides the new language constructs, mbeddr features also extended type-systems like for example to specify and check physical units. By deeply integrating many consistency checks directly in the IDE, mbeddr empowers its users to write robust code. For performing functional verification, mbeddr offers first class support for writing test cases and user friendly formal verification at the abstraction level of the DSLs. For formal verification, mbeddr integrates the CBMC model checker (http://www.cprover.org/cbmc/). mbeddr has language support for the definition of the verification environment and for specification of verification conditions. The verification environment is translated as macros supported by CBMC and the verification conditions as C-level asserts or error labels which can be natively checked by CBMC. The counterexamples provided by CBMC in case of verification failure are subsequently lifted by mbeddr at the abstraction level of the DSLs.

In Figure 1 we present an overview of the mbeddr technology stack. “mbeddr” itself is built on top of the Meta Programming System (MPS) from JetBrains. The open architecture of MPS allows the set of languages which come with mbeddr to be further extended according to the domain specific needs of its users.

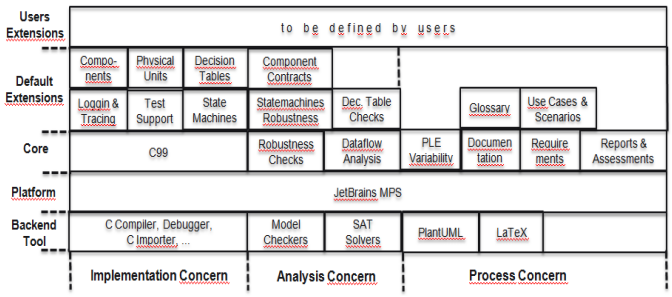



Figure 1. The “mbeddr” technology stack

## III. USING MBEDDR TO PROGRAM TRAIN CONTROL UNITS

Siemens process automation framework used to develop the train applications already incorporates a modeling approach using the CFC (Continuous Function Charts) diagram language. From CFC models is generated C-code. The framework itself allows for designing and implementing safety-critical software systems that are ensured by the following incorporated mechanisms: pre-certified library modules implementing basic functions and a proven-in-use C-compiler.

In our case study the non safety-relevant class software has been used. The algorithm of calculation of the target point of an automated train driving system has been implemented using mbeddr and seamlessly integrated into overall train control software system.

The work on the case study comprised of the following major steps shown on Figure 2.

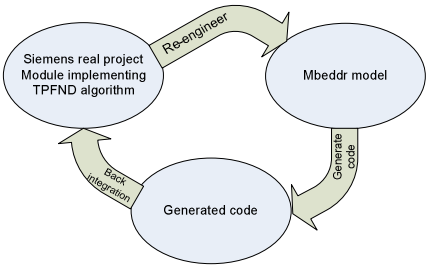



Figure 2. Case study workflow.

- Take a module from a Siemens real project “Automated train driving system” (see Figure 3). The algorithm of calculation of the target point has been implemented in Siemens process automation framework using CFC-diagram.

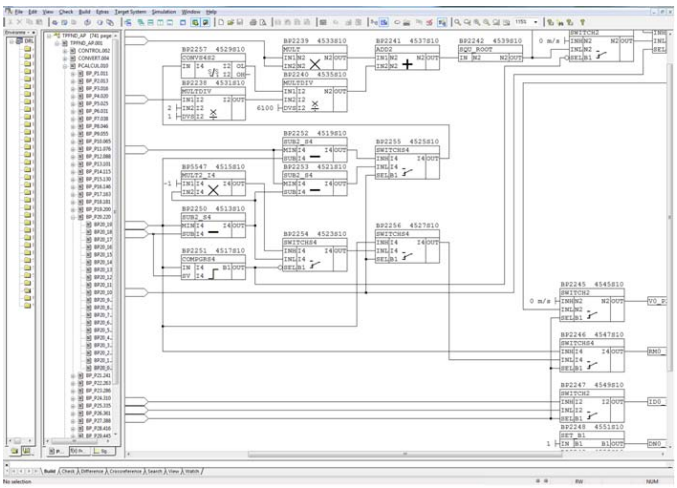


Figure 3. Functional block of Siemens project.

- Re-implement the function using mbeddr using the high level abstractions which fit at best for our problem domain. It included modeling the requirements; code generation and refinement (see Figure 4).

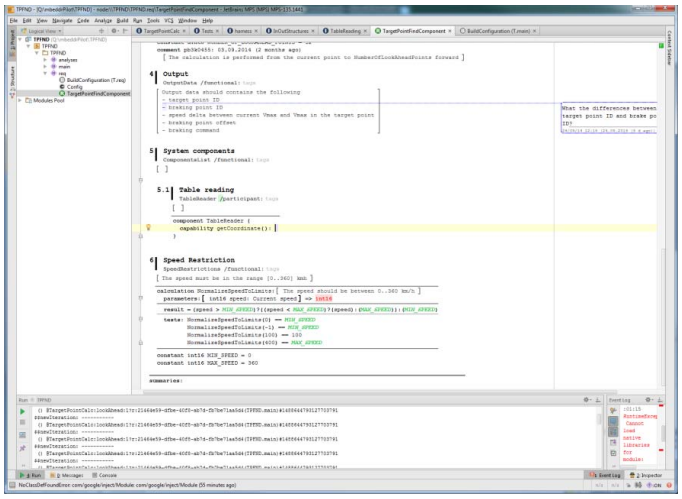
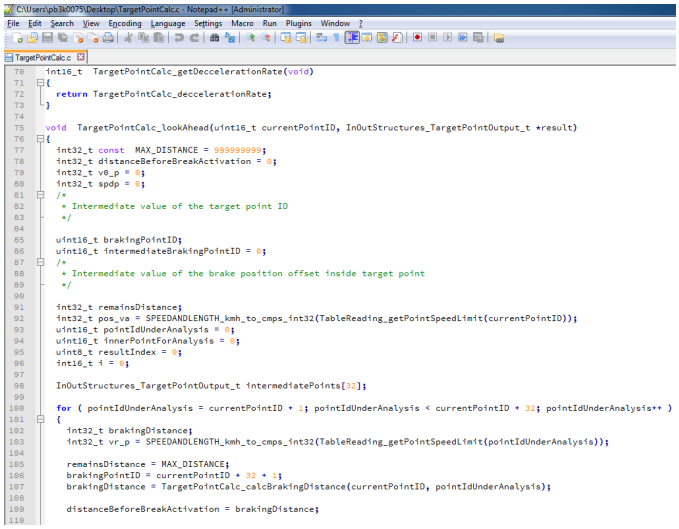


Figure 4. mbeddr model.

- Replace the selected module from the original project with generated code from mbeddr (see Figure 5).



```
int16_t  TargetPointCalc_getDecceleration Rate(void)
{
  return TargetPointCalc_decceleration Rate;
}

void  TargetPointCalc_lookAhead(uint16_t currentPointID, InOutStructures_TargetPointOutput_t *result)
{
  int32_t const  MAX_DISTANCE = 999999999;
  int32_t distanceBeforeBreakActivation = 0;
  int32_t v0_p = 0;
  int32_t spdp = 0;
  /*
   * Intermediate value of the target point ID
   */
  uint16_t brakingPointID;
  uint16_t intermediateBrakingPointID = 0;
  /*
   * Intermediate value of the brake position offset inside target point
   */
  int32_t remainsDistance;
  int32_t pos_va = SPEEDANDLENGTH_kmh_to_cmps_int32(TableReading_getPointSpeedLimit(currentPointID));
  uint16_t pointIdUnderAnalysis = 0;
  uint16_t innerPointForAnalysis = 0;
  uint8_t resultIndex = 0;
  int16_t i = 0;

  InOutStructures_TargetPointOutput_t intermediatePoints[32];

  for ( pointIdUnderAnalysis = currentPointID + 1; pointIdUnderAnalysis < currentPointID + 32; pointIdUnderAnalysis++ )
  {
    int32_t brakingDistance;
    int32_t vr_p = SPEEDANDLENGTH_kmh_to_cmps_int32(TableReading_getPointSpeedLimit(pointIdUnderAnalysis));

    remainsDistance = MAX_DISTANCE;
    brakingPointID = currentPointID + 32 + 1;
    brakingDistance = TargetPointCalc_calcBrakingDistance(currentPointID, pointIdUnderAnalysis);

    distanceBeforeBreakActivation = brakingDistance;
```

Figure 5. Generated code.

- Equivalence check using a set of test cases and review by domain-matter experts.

Each of the following subsections presents the steps we performed during our case-study.

### A. *Implementing the application using higher level abstractions and domain specific languages*

As mentioned above, mbeddr provides many first-class concepts on top of traditional c-language. While we were interested in trying as many of them as possible to evaluate their usefulness and fitness for practical development of the function we selected, it was obvious that we must find proper balance and match the set of used languages and constructs to the real needs of the application.

In order to identify those needs we started our mbeddr project from requirement modeling stage. Requirement modeling language of mbeddr allows for structured collection of requirements not only in textual natural language form, but also provides capability to define constants and portion of logic which can further be used in other models. This prevents data duplication and ensures consistency of requirements and further implementation.

Since the function under consideration contained mostly kinematic calculation of current and future train motion parameters quite natural was the application of physical unit modeling. We defined the model of physical units (meter, second, meter per second, and others) and the rules of their conversion.

In the implementation model we utilized reference to logical part of requirements, tracing of code and requirements, physical units and mathematical notation. Combined utilization of mathematical notation and physical units added design-time checking of calculation consistency and visual expressiveness of the model. Consistency was assured by automatic unit conversion and checking of assignment operations which provided by mbeddr as an extra layer of type checks (e.g. attempt to assign result of an expression giving meter per seconds to a variable holding meters will be marked as a type error).

We also tried to introduce state machine notation into the implementation model, but it was considered redundant, as the function itself was not tending to have portions which could be interpreted in terms of states and transitions. Moreover, the study showed that state machine model yields the c-code with use of C99-specifix syntax features, while target compiler was only supporting C89 standard.

Finally a set of test cases was developed. They are also traced back to the requirements they cover.

### B. *Integration of model-based approach into existing product*

Target development environment based on CFC-diagram language was in fact using C-based representation of function blocks. That was not pure C-code but specifically formatted files with specific header containing meta-data of the functional block and its signal-oriented interfaces and code sections supplemented with primitive template substitution language used for the target code generation.

C-code generated from mbeddr shall be inserted into one of those code sections in functional block file. Before this could be done, the source file shall be made as much as possible independent form any external header files dependency. This was achieved by means of C-preprocessor which was applied to mbeddr-generated code.

Meta-data header and interface code sections were prepared as a separate template file, and again the C-preprocessor was used to insert the mbeddr-code into the template avoiding manual copy and paste operations.

In this way using a script with appropriate calls to the C-preprocessor and a template file, the integration of new code into legacy tool chain was automated to a large extent. The import of new functional block into the functional block library still required manual work in library management tool.

### C. *Modeling and code generation capabilities of model-based development workbench*

Mbeddr modeling capabilities encompass wide range of development cycle stages: starting from requirements and ending with code verification, testing and documentation. Ability to combine several modeling languages not only in the same project, but also in the same module supports achievement of high degree of logical consistency and avoidance of data duplication.

Modeling and generation approach used in mbeddr allows for easy control over the target code. This is evidenced by the fact that despite the code generator is based on C99 standard; we were able to compile the code with old C89 compiler.

Below is a summary of mbeddr capabilities which were used during our pilot project:

- Useful from requirements to acceptance testing phases
- Supports physical units, state machines, decision tables, math equations
- Generates well-formatted and human-readable C99 code
- Testing with asserts and mocks
- Debugging on models level
- Supports formal verification
- Supports requirements, tracing, documentation generation.

## IV. Lessons Learnt

We applied a modern model-based development approach (mbeddr) to reengineer a subsystem written in a legacy engineering framework for railways applications. Below are the most important lessons that we learnt from our case-study.

*Using inappropriate languages is painful on the long run.* The application was written originally in a legacy language widespread in the rail domain. That language needed to be used due to pragmatic reasons even if it was not adequate to capture the business logic of our application. Due to the conceptual gap between the business domain of the application and the language mechanisms, the code was originally unnecessarily big and complex. In order to implement change requests and to maintain the code over several years, a substantial effort is needed and this results in unnecessarily big costs.

*Easing the implementation of complex logic* has been achieved by use of mathematical notations and procedural programming features of mbeddr. Thereby we reached a significant reduction of code size and complexity. The code from mbeddr is much easier to review and changes are much easier to implement.

*Linking requirements to the code and tests eases the quality assurance and functional verification.* By linking requirements to the code and to the corresponding tests, we made quality assurance more systematic. Furthermore, code reviews are eased and the coverage of the verification with respect to the functional requirements more transparent.

*Introducing state-of-the-art software engineering practices* significantly improve the software development and would allow for involving software-domain experts into design of complex train control logic. Among such practices are like requirements traceability, support of unit test, support of formal verification, improvement of code robustness by using advanced type-checks.

*Using projectional editors.* Changing mind from the traditional way of coding using plain text to the projectional editing approach of mbeddr took around two weeks. After that it became easier to use this new way of editing. One found drawback was the restrictions imposed on the copy-paste of code. Using the automation provided by mbeddr, the coding speed and cleanness of the code were increased.

Adaptation and introduction period will be required both by control engineers familiar with CFC diagrams and traditional software developers as mbeddr offer the editing approach differing from both of those. It is also restrictive in terms of used IDE – since everything in mbeddr is considered as a model (even though it looks like c-code or plain text) and stored in specific format it is not possible to use your favorite text editor, you must stick to projectional editor in mbeddr environment.

The s*upport of safety-relevant certification* of the artifacts generated from mbeddr workbench will require further investigation. According to safety standards, the domain specific languages used need to be thoroughly specified and the correctness of the mbeddr code generator shall be proved. On other side, the verification capabilities of the CBMC tool (or other formal verification tools which can be integrated into mbeddr) could become a strong argument in the safety case since the coverage that these tools can achieve is much higher than what can be achieved by traditional testing.

## V. Conclusions and Future Work

As a result, the identified advantages of the undertaken approach are a) modeling spans from requirements down to code generation and validation; b) available rich set of expressive domain-specific languages; c) seamless integration with versioning control system.

At the same time, we found several drawbacks pointing to significant efforts from practitioners to apply the described approach: a) new languages (totally different from CFC-diagrams) to learn; b) more efforts to support separate development cycle and integration.

As further work we will be extending mbeddr to the level of tool for data analysis or other complex components for Siemens business units and the railway industry in particular. I will extend mbeddr code generator for automatic creation of legacy engineering framework's compatible code and will extend mbeddr languages to support domain-specific abstractions like "normalized values".

## References


[1] M. Voelter and T. Stahl. "Model-Driven Software Development: Technology, Engineering, Managment", Wiley, 2006.

[2] M. Voelter, D. Ratiu, B. Kolb, B. Schaetz , "mbeddr: Instantiating a Language Workbench in the Embedded Software Domain," Journal of Automated Software Engineering, Vol. 20, Nr. 3, pp. 339-390, 2013.