Interface Definition and Code Generation in heterogeneous Development Environments from a Single-Source

A Domain Specific Language Approach for Automotive Predevelopment Environments

Dominik Bauch, David Lenz, Pascal Minnerup and Andrei Avram

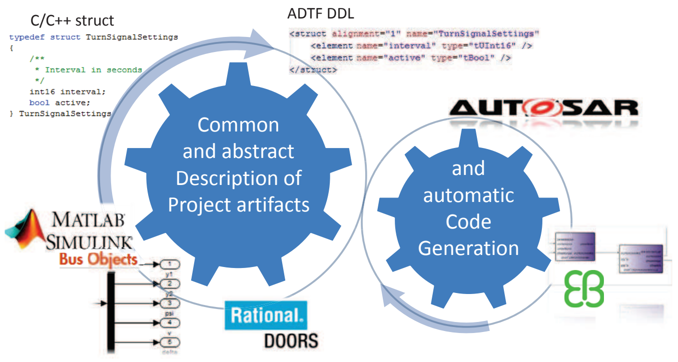
fortiss GmbH
An-Institut Technische Universität München
Guerickestr. 25, 80805 Munich, Germany
{bauch,dlenz,minnerup}@fortiss.org, andrei.avram@tum.de

Abstract: Typical automotive software development involves a broad spectrum of development environments and this leads to friction loss, unnecessary efforts and inconsistencies discovered later in the development process. In this paper we advocate that language engineering technologies can be used for the interface definition and code generation in order to glue artifacts and to support the work within such heterogeneous development environments. For our implementation we used the domain specific language stack *mbeddr* based on the *Meta Programming System* (MPS) from JetBrains. We show our preliminary experience with our method gained during the work on a component-based, distributed advanced driver assistant system. The system has to be realized for both the automotive specific prototyping framework ADTF in combination with MATLAB/Simulink and the AUTOSAR RTE middleware. With the help of the proposed approach, we were able to generate large redundant parts of the software system including different interface representations with only small modifications of the existing architecture.

1 Introduction

For the efficient and fast specification as well as implementation of novel software-intensive and highly integrated advanced driver assistance systems the utilization of a broad spectrum of specialized development tools is essential. Virtually all today's advanced driver assistance systems are built on top of a component-based software architecture and hence well-defined interfaces describing the individual's component communication flow are vital. But as an inherent consequence of the in general heterogeneous development environment, most of the interfaces have to be specified in various manifestations to fulfill the specific syntactic and semantic requirements of the respective tools. For example, MATLAB/Simulink[1] represents its interface with so-called bus objects whereas in a C programming environment typically simple records are used (cf. Figure 1). Although, these interfaces will contain semantically identical data they have to be maintained simultaneously

[1]`www.mathworks.com`


C/C++ struct
ADTF DDL
Common and abstract Description of Project artifacts
and automatic Code Generation
AUTOSAR
MATLAB SIMULINK Bus Objects
Rational. DOORS

```
typedef struct TurnSignalSettings
{
    /**
     * Interval in seconds
     */
    int16 interval;
    bool active;
} TurnSignalSettings
```

```
<struct alignment="1" name="TurnSignalSettings"
    <element name="interval" type="tUInt16" />
    <element name="active" type="tBool" />
</struct>
```

Figure 1: Centralized interface description in affiliation with a code generation framework for ADTF, MATLAB and AUTOSAR

with different tools. With the project's progress the individual interface representations will drift apart and an inconsistent specification is the inevitable consequence.

In addition, typically there are also two parallel and just loosely coupled lines of development which will intensify the above mentioned situation. There is on the one hand a lightweight and agile software engineering process with very short release cycles for the development of the actual functional code of the new advanced driver assistance system. This development branch uses an automotive-specific rapid prototyping environment that allows for an integration of various tools like the *Automotive Data and Time-Triggered Framework* (ADTF). On the other hand there is a more heavy weight systems engineering process fitted for the development of the software's target architecture and ECU itself. Nowadays, the target architecture is typically based on a specific implementation of the well-known *Automotive Open System Architecture* (AUTOSAR) standard. At certain points in the project's timeline, both development branches with their independently evolved interface specifications will be merged together. The better the functional code follows the interface specifications of the target architecture at this moment the less effort is expected.

In our understanding, to reduce costs and risks for maintenance and integration it is indispensable to introduce a centralized and common specification method, which is able to ensure high interface integrity even in such complex, distributed and particularly heterogeneous development environments. We believe also that it is advantageous for the quality of the software system if its specification is decoupled from the specific capabilities of the individual development tools. Within an automotive predevelopment project, there are also some recurring, error-prone and especially time consuming tasks which should be automated as far as practicable. Especially with the help of a suitable code generation framework the mostly trivial source code for the wrappers of the functional software, needed for its embedding within the target architecture as well as within the rapid prototyping environment, can be derived from an interface specification[2]. Figure 1 outlines

[2]Of course, assuming an implementation architecture for the functional code is given

our notion that abstract interface descriptions should serve as a unified representation of concrete interface definitions as well as a source for platform-dependent wrappers.

Contribution of this paper In the following, we will present an approach based on our domain specific language *Rapid Automotive Predevelopment Interface Definition* (RAPID). It is a possible solution to the problem of inconsistencies with interface specifications within heterogeneous and distributed development environments along with component based architectures. Using this domain specific language (DSL) as a kind of glue-code, we are able to establish reliable links between different development and documentation tools whereupon only one common and centralized description of the relevant project artifacts exists. Furthermore, we leverage the information content of RAPID, in conjunction with the knowledge about a project-wide established implementation architecture for the functional code, to generate ready-to-use software component wrappers. This code generation allows the seamless integration of our functional code in the automotive-specific rapid prototyping environment ADTF as well as in an AUTOSAR target architecture.

In the following sections of the paper we introduce the sketched approach in detail and discuss its integration into an automotive predevelopment process. Within Section 2, related work is discussed. Section 3 gives a brief overview of the tools and techniques used for the implementation of our DSL approach. Subsequently, Section 4 describes our domain specific language and illustrates its usage with an example. Furthermore, we present our preliminary experiences and outline the integration into an automotive development process. Finally, the last section summarizes our work and highlights ideas for future tasks.

2 Related Work

Amongst others, we see our observations and hypotheses stated in Section 1 as well as our drawn conclusions in general confirmed by the author of [1]. In this paper, Broy identified and described the most important key challenges future automotive software engineering projects will have to cope with. Some of the fields which are addressed by our paper in a pragmatic way, namely an improved development tool integration and a sophisticated interface specification method together with a fitting requirements engineering, are explicitly mentioned by Broy.

The authors of [2] point out that domain specific abstractions in requirement models can help by the integration of continuous and discrete systems. Whereas models of continuous systems can be used for code generation, the models of discrete systems are very helpful for an unambiguous interface specification and the description of relationships between components (especially when the models are enriched by a behavior specification) but with the main drawback that two artifacts, model and code, have to be maintained and synchronized. The approach proposed in our paper addresses exactly this issue and ensures at all times a coherent and consistent view on the interfaces. Hence, all information about the interfaces' specifications is deposited in one centralized source.

In [3], the authors present a vision on how an integrated model-based tool environment should look like in order to ensure a seamless development process. Especially in the automotive domain, the process is mostly imposed by the tools available and thus many proprietary software and file formats dominate the domain. As each tool is specialized for one development step, models have to be repeatedly (mostly manually) transformed or rebuilt throughout the process. This leads to information-loss, redundancy, inconsistency and thus to inefficiency. Although the authors show how tools have to be designed to overcome these drawbacks, the description is on an abstract level and more a guideline and roadmap towards an improved model-based process. In contrast, our work aims at focusing on a concrete aspect of this process, i.e. the interface definitions, and shows how model-based design can overcome inconsistencies between different branches of development.

An overview about Domain Specific Languages is presented in [4]. As in our approach, the paper identifies maintainability and portability as some of the main advantages of domain specific languages. Additionally they point out that DSLs allow optimization improve the testability. The main challenges of DSLs are seen in implementing a DSL and the availability of DSLs. Our research adds some practical experience about designing and using DSLs.

Such practical experience has also been produced by other research groups. In [5] a DSL is presented that provides real time and validation properties that are particularly useful for embedded computing. The research focuses on designing a language that allows high confidence in the developed software. The approach described in [6] includes generating middleware specific code out of a domain specific XML document. The goal of that paper is to manage the complexity and heterogeneity of distributes online games.

There are several approaches of creating interface definition languages. Well known examples are the AUTOSAR Textual Language Framework [7] or the OMG interface specification language [8]. These interface specification languages can express very complex interfaces, but are either not able to store all information necessary for the code generation presented in this paper or require a large overhead to express it. Furthermore we want to exploit the advantages of language engineering for making a compact and extendible language that can evolve with the predevelopment needs.

3 Language Engineering Tools

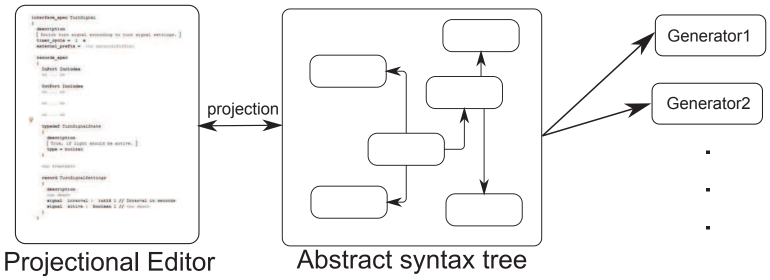
For realizing the domain specific language RAPID defined in Section 4 we rely on the existing *Meta-Programming-System* (MPS) from JetBrains [9] and on the language stack *mbeddr* [10]. In this section we briefly introduce these tools and the concepts behind them.

3.1 Meta Programming System

MPS is a language workbench [11], i.e. a framework that allows the definition and extension of domain specific languages and provides tools to work with them.

After defining the syntax and semantics of a language in form of a meta model in MPS, one can also use MPS to write models in this language. All the information of a model is stored in a so-called abstract syntax tree (AST) representing the elements of the syntax of the defined DSL. In contrast to plain text editors in traditional IDEs for programming languages, MPS implements a projectional editor for the language syntax. That means that one modifies the AST directly instead of relying on parsers to generate this tree from plain text. Parsers can be very complicated to build and to maintain. Projectional editing has the advantage that the written code is correct by construction. In other words, only syntactically and semantically correct input is allowed. This is ensured by the type system derived from the meta model. The prevention of incorrect input makes the use of a DSL straight forward and makes it less error-prone and thus usable for domain experts that are not familiar with general purpose programming languages like C.

Within the MPS framework, it is possible to define generators that transform the specified model into arbitrary representations. One of those representations can be conventional type definitions, e.g. C-structs or Matlab bus objects, but also more sophisticated functional (wrapper) code. This approach allows generating different views of the same information for different development environments. This ensures that changes have to be made only for the model and all representations can be regenerated in a coherent manner. This concept of MPS with projectional editing and generation of artifacts can be seen in Figure 2.



```
interface_spec TurnSignal
{
  description
  [ Switch turn signal according to turn signal settings. ]
  timer_cycle =  1  a
  external_prefix =  <no externalPrefix>

  records_spec
  {
    InPort Includes
    << ... >>

    OutPort Includes
    << ... >>

    << ... >>

    << ... >>

    typedef TurnSignalState
    {
      description
      [ True, if light should be active. ]
      type = boolean
    }

    <no freetext>

    record TurnSignalSettings
    {
      description
      <no desc>
      signal  interval :  int16 1 // Interval in seconds
      signal  active :  boolean 1 // <no desc>
    }
  }
```

Figure 2: MPS projectional editing concept. The editor changes the AST directly. Generators build different representations of the model.

3.2 mbeddr

mbeddr provides an environment within MPS for embedded software development in C. It offers higher-level extensions for the C-language like state machines, unit tests, etc. as well as static checks and verification mechanisms. It aims at making C extendable without adding additional runtime-costs. For our approach, we use the ability of MPS to extend a DSL in order to use the already defined semantics of datatypes and structures of mbeddr. Also parts of the existing code generation for C can be re-used and extended for our purposes.

4 RAPID — The proposed DSL

The tools described in Section 3 are the basis for defining our DSL RAPID. This section presents the meta model of the interface description. Next, it describes the generation of wrapper code and other development artifacts based on an instance of this meta model. An example will illustrate the whole process.

On the one hand, the meta model of the interface description has to support all information that is necessary for generating the wrapper code and other needed development artifacts of the automotive project. On the other hand, it should remain simple in order to allow fast adaptation to new requirements of the specific project and simple creation of projectional editors. In contrast to general purpose interface definition languages like OMG's IDL [8], the meta model presented in this section is optimized to fulfill these requirements.

Figure 3 shows the meta model of the interface description. The root class is the system specification (*system_spec*). It is the starting point from which a user can reach all other artifacts and it is, in principle, a collection of all component interfaces of the whole software system. The system specification can contain an arbitrary number of these interface specifications (*interface_spec* in Figure 3). Each instance of an *interface_spec* includes the data types used (*records_spec*), the incoming connection ports (*inputports_spec*) and the outgoing connection ports (*outputport_spec*). Finally it contains a mapping specification (*mappings_spec*) that the generator code can use to map the results of triggered input port functions to output ports, handle timer events and store status information.

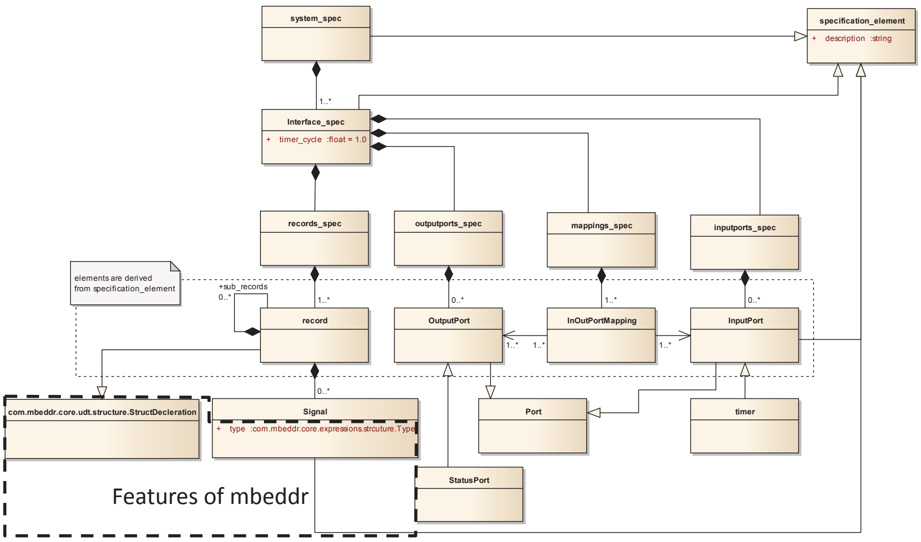


Figure 3: RAPID meta model

The members of the *records_spec*, *outputports_spec*, *mappings_spec* and *inputports_spec* are derived from a common specification element that offers documentation capabilities. A record, which is based on the mbeddr's declaration of structs, contains *Signal*s and nested records (*sub_records*). Signals refer to some of the built-in data types defined in mbeddr. For the ports used in the *InOutPortMapping*, there are two kinds of special values. The *StatusPort* is a virtual outport that defines the wrapper code should not forward the mapped input directly to the functional code, but store it to be available at any place in the component. The timer is a special inport that is not triggered on receiving data from another component, but triggered periodically.

```
/* Switch turnsignal */
interface_spec TurnSignal
  timer_cycle = 0.1 s
  records_spec
    record TurnSignalSettings
      signal interval : uint16 /* Interval in seconds */
      signal active : bool /* Activate the turnsignal */
  inputports_spec
    port turnSignalSettingsPort: TurnSignalSettings
  outputports_spec
    port turnSignalStatePort: TurnSignalState
  mappings_spec
    mapping timer => turnSignalStatePort
```

Listing 1: Example of an interface specification for a turn signal controller. But in principle due to the concepts of projectional editing the formatting of the DSL can be arbitrary.

Listing 1 shows an example of an interface specification for a turn signal controller. The turn signal controller is a simple component that expects a turn signal setting as an input and switches the turn signal on and off according to this setting. For example the signal setting can specify that the controller should switch the turn signal on and off every half second. The corresponding software component includes an input port for setting the turn signal behavior and an output port stating, whether the light is currently active. Listing 1 defines the types *TurnSignalState* and *TurnSignalSettings* for these ports. Additionally there is a mapping (*turnSignalStateMapping*) stating that at every timer event, the wrapper code should write a new value to the *turnSignalStatePort*. Finally, the interface definition specifies the cycle time of this timer event as 0.1 seconds. Hence, the actual user can command the turn signal controller to switch the turn signal on and off at any interval that is a multiple of 0.1 seconds.

This short and simple RAPID specification suffices for generating the ADTF and potentially AUTOSAR wrappers.[3] In ADTF the software engineer can configure the software system by connecting software components in a filter graph. Each software component is represented as an ADTF filter. As described in the introduction we use this filter as a wrapper for the actual functional code. Figure 4 shows an excerpt of our architecture allowing such a separation of functional code and wrapper code. The *MessagePassing-Interface* contains a sending method for each *OutPort* defined in RAPID. Plus, for each *StatusPort*, it offers a getter method for reading the stored value. The ADTF filter as well as the AUTOSAR wrapper has to implement these methods. Additionally, the filters con-

[3]with wrapper we refer to code that embeds the functional code to an AUTOSAR conform environment

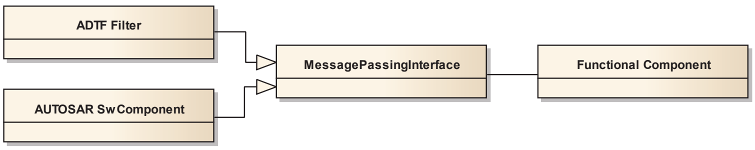


Figure 4: Excerpt of our implementation architecture for functional code and corresponding wrappers

tain code for initialization, creation of the pins, processing incoming data and converting it to the target C structs. Code generators can generate all this code using the information of the RAPID files.

The remaining part is the *Functional Component* (compare Figure 4). For this component, a framework is generated including initialization and deinitialization functions plus one receiving function for each input port defined in RAPID. These port functions are also defined in an interface of the functional component. This way, any change of the generated interface is immediately recognized by the compiler and the software developer is forced to update the functional code, too.

Furthermore and as depicted in Figure 4, the *Functional Component* has only a reference to the abstract *MessagePassingInterface*. Hence, it does not know, whether it is using ADTF or AUTOSAR for communicating with other components. Thus, the developer does not have to cope with the implementation details of the specific middleware and they can reuse the same functional code for ADTF and AUTOSAR. Only for the final serial deployment the software engineers should remove the abstraction layers introduced for platform independence and optimize the functional code. In the example used for this paper, the output port *turnSignalStatePort* is mapped to a timer. Therefore, the generated interface for the functional code also includes a method with *TurnSignalState* as return type that the wrapper code will trigger periodically.

All in all, the code generation replaces the error-prone activity of writing more than 400 lines of repeating code by hand. These 400 lines of code do not include the completely identical functions that a software engineer can implement in a common base class for all ADTF filters.

Besides the in detail explained generation of platform dependent wrapper code, there exist also several artifacts in a heterogeneous development environment describing the same interface:

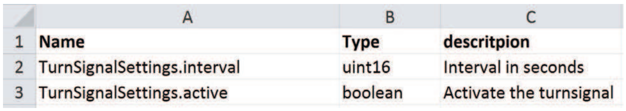
- Software systems implemented in C/C++ need C interfaces,
- components implemented in MATLAB/Simulink need Simulink Bus Objects,
- debugging and connection tools in ADTF need ADTF data definition language (DDL) files and
- the project management needs to include the interfaces in the requirements specification, amongst others, to make contracts with suppliers for a later series development.

```
<adtf:ddl xmlns:adtf=”adtf”>
  <header><description>Switch turnsignal</description></header>
  <structs><struct name=”TurnSignalSettings”>
      <element ... name=”interval” type=”tUInt16”/>
      <element ... name=”active” type=”tBool”/>
  </struct></structs>
</adtf:ddl>
```

(a) ADTF description of the record TurnSignalSettings

```
typedef struct TurnSignalSettings
{
  int16 interval; // Interval in seconds
  boolean active; // Activate the turnsignal
}TurnSignalSettings;
```

(b) C description of the struct TurnSignalSettings



	A	B	C
1	**Name**	**Type**	**descritpion**
2	TurnSignalSettings.interval	uint16	Interval in seconds
3	TurnSignalSettings.active	boolean	Activate the turnsignal

(c) Tabular representation of TurnSignalSettings

Figure 5: Different representations of the TurnSignal interface

Without code generation, developers have to maintain all these semantically equal descriptions manually and inconsistencies are unavoidable. The interface description language presented in this paper solves this problem by creating a single source of information as we are able to generate all the above listed artifacts with the help of RAPID. Listing 5a shows how the struct *TurnSignalSettings* is represented in ADTF with its DDL format, the C/C++ version is depicted in Listing 5b and Figure 5c shows a tabular representation that a project leader can use for importing it to a requirements management tool. The implementation of the corresponding generation files is straight forward, as each of these representations mainly contains the same information. If a new representation is necessary that contains additional information, developers can add this information to the RAPID meta model. This way, RAPID offers the tools necessary for keeping many development artifacts consistent.

4.1 Experience with the code generation

We tested the code generation on our sub-project, a part of a distributed advanced driver assistant system. The sub-project consists of a total of 17 components with 119 ports. For developing the code generation we found several places in the source code that we had to change, because we had implemented slightly different architectures for similar wrapper code. The code generation enforced to implement a common architecture. Some of these architectural inconsistencies were even undetected defects in the source code. Further-

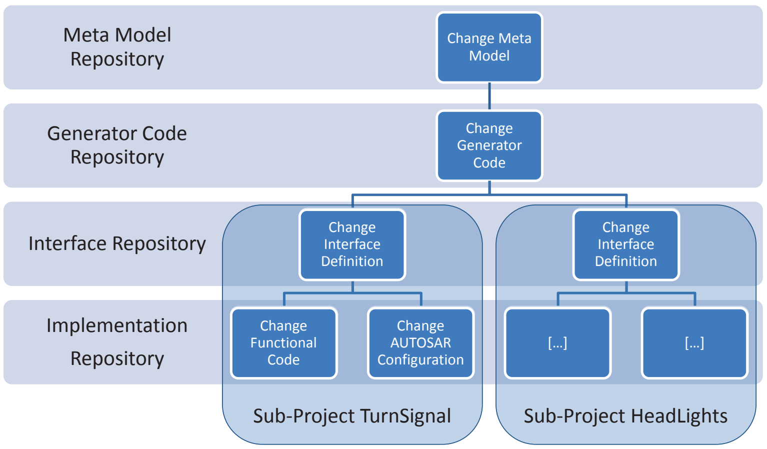


Figure 6: Hierarchy of RAPID artifacts. The higher a modified artifact is located in the hierarchy, the higher the impact on other teams.

more, we had to rename variables and some namespaces to a common naming pattern that code generators can describe and update our C include hierarchy to a systematical pattern. As we had already strictly separated functional code and wrapper code no other changes were necessary.

4.2 Integration into the Development Process

As described in the introduction, there are typically two or more independent development lines and many independent teams using different development tools like MATLAB/Simulink, ADTF or an AUTOSAR tool chain. The previous section described how RAPID code generation can generate the different artifacts needed by these teams. The process engineers should integrate this code generation into the development process. The following section describes this integration.

Figure 6 visualizes the general principle of integrating the interface definition and code generation framework into the development process. The engineers working on a specific development line (i.e. the functional prototype of the sub-project *TurnSignal* depicted in Figure 6) should not directly change the generated interfaces or other generated code. If possible, developers should make the changes in the non-generated functional code: the implementation repository level shown in Figure 6. If the change requires modifying the interfaces, they should change the RAPID files and regenerate the language specific generated interface code. Automatic build scripts should include the regeneration and the developers should check in RAPID files to a common version control repository. As soon as the other teams update and rebuild their projects, the automatic build script will generate their language and domain specific interfaces, too (compare common box ”‘Change

Interface Definition"' in Figure 6). If the interface change causes a problem for one of the projects, this is recognized at this early stage and not postponed to the integration process. Generating the code with mbeddr and build scripts is possible by using ANT scripts[4]. In order to ensure that no developer uses old generated code files, version control should exclude the generated code.

In some cases changing the RAPID description will not suffice, because the information that the developer needs to change is contained in the code generators. The code generation framework should be a separate version controlled project that is included into other projects using for example git sub modules[5]. As the code generation is included in an automatic build script, changes to the generators will have an effect on all other teams depending on it very quickly. As indicated in Figure 6, a large number of projects might depend on these generators.

The same applies to the project management information specified in a requirements management tool. If a bad specification is found in the specification file, requirements engineer should not make the change directly in the requirements management tool. Instead, they should make the change in the RAPID files and regenerate the requirements specification document. This ensures consistency between the functional software and the requirements and hence the project team can use the working prototype as a proof for a good requirements specification.

5 Conclusions and Future Work

In this paper we presented the domain specific language RAPID together with a powerful code generation framework that has the capability to better integrate tools of a heterogeneous development environment. We showed that RAPID can act as a single-source for required artifacts within a software-intensive industrial automotive pre-development setting. As a result, no more information about interface specifications is duplicated and hence modifications have only to be done at one dedicated place. Consistency in the resulting interface specification for the individual tools is built in. Furthermore, language engineering is able to embrace the concepts of the developer's domain. Therefore, in our context, using language engineering for the system specification is a powerful and at the same time very intuitively usable to instrument.

As shown in Section 4.2, such an approach can be seamlessly embedded in an already established distributed development process. One reason for this is the very compact and natural (in terms of our domain) design of RAPID that allows easy and pretty fast migration of existing code. Furthermore, the possibility of code generation out of the domain specific language is particularly easy with the leveraged framework because there is no need to parse complicated syntactical constructs thus the design of the language can be chosen arbitrarily.

Up to our measurements, we have reduced the amount hand-written code, which is nec-

[4]`http://confluence.jetbrains.com/display/MPSD30/HowTo+--+MPS+and+ant`
[5]`http://git-scm.com/book/en/Git-Tools-Submodules`

essary for the interface specification and the implementation for a wrapper (filter) for the ADTF environment in our setting, by a factor of eight. We assume that the factor in terms of working hours is even higher due to fact that developers can focus on the actual important things of their work.

We are currently extending our code generation framework so we will be able to generate AUTOSAR *RunnableEntity*s from a RAPID interface specification. This will be done in the same fashion as we did for the ADTF filters. With this extension we can embed our functional code within the rapid prototyping environment of the agile line of development as well as within the systems engineering line of development. This is achieved fully transparently and without any additional effort. Additionally, an implementation of the proposed connection to requirements documentation tools, e.g. DOORS from IBM Rational, as well as to the tools used for the ECU system specification is a valuable goal which will increase productivity and in particular quality of the overall process and, by implication, the product's quality more and more. Finally, we will assure our position about saving working hours with the help of a case study done within our project about a highly integrated advanced driver assistance system.

References

[1] M. Broy, "Challenges in automotive software engineering," in *Proceedings of the 28th international conference on Software engineering*, pp. 33–42, 2006.

[2] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner, "Software engineering for automotive systems: A roadmap," in *2007 Future of Software Engineering*, pp. 55–71, 2007.

[3] M. Broy, M. Feilkas, M. Herrmannsdoerfer, S. Merenda, and D. Ratiu, "Seamless model-based development: From isolated tools to integrated model engineering environments," *Proceedings of the IEEE*, vol. 98, no. 4, pp. 526–545, 2010.

[4] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography.," *Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[5] K. Hammond and G. Michaelson, "Hume: a domain-specific language for real-time embedded systems," in *Generative Programming and Component Engineering*, pp. 37–56, 2003.

[6] T.-Y. Hsiao and S.-M. Yuan, "Practical middleware for massively multiplayer online games," *Internet Computing, IEEE*, vol. 9, no. 5, pp. 47–54, 2005.

[7] Artop User Group, "ARText — An AUTOSAR Textual Language Framework." `https://www.artop.org/`.

[8] Object Management Group, "OMG formal specifications." `http://www.omg.org/spec/`.

[9] Jetbrains, "Meta programming system (MPS)." `http://www.jetbrains.com/mps/`.

[10] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb, "mbeddr: an extensible c-based programming language and IDE for embedded systems," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pp. 121–140, 2012.

[11] M. Fowler, "Language workbenches: The killer-app for domain specific languages," 2005.