2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops

Designing a graphical domain-specific modelling language targeting a filter-based data analysis framework

C. Köllner, G. Dummer, A. Rentschler and K.D. Müller-Glaser
FZI Research Center for Information Technology Karlsruhe
Dept. of Embedded Systems and Sensors Engineering (ESS)
Haid-und-Neu-Str. 10-14, D-76131 Karlsruhe, Germany
Email: {koellner|dummer|arentsch|kmg}@fzi.de

***Abstract*—We demonstrate the application of a Model-Driven Software Development (MDSD) methodology using the example of an analysis framework designed for a data logging device in the field of vehicle testing. This mobile device is capable of recording the data traffic of automotive-specific bus systems like Controller Area Network (CAN), Local Interconnect Network (LIN), FlexRay and Media Orientied Systems Transport (MOST) in real-time. In order to accelerate the subsequent analysis of the tremendous amount of data, it is advisable to pre-filter the recorded log data on device, during the test-drive. To enable the test engineer of creating data analyses we built a component-based library on top of the languages SystemC/C++. Problematic with this approach is that still substantial programming knowledge is required for implementing filter algorithms, which is usually not the domain of a vehicle test engineer. In a next step we developed a graphical modelling language on top of our library and a graphical editor. The editor is able of verifying a model as well as of generating source code which eliminates the need of manually implementing a filter algorithm. In our contribution we show the design of the graphical language and the editor using the Eclipse platform and the Graphical Modelling Framework (GMF). We describe the automatic extraction of meta-information, such as available components, their interfaces and categorization annotations by parsing the library's C++ implementation with the help of Xtext. The editor will use that information to build a dedicated tool palette providing components that the designer can instantiate and interconnect using drag-and-drop.**

I. Introduction

Automotive manufacturers subject their pre-production cars to extensive tests before the production starts. As modern cars feature an ever-increasing number of electronic control units (ECUs), the verification of inter-ECU communication is a critical aspect. ECUs are connected by automotive bus systems [1], such as the Controller Area Network (CAN), Local Interconnect Network (LIN), FlexRay and Media Oriented Systems Transport (MOST). Bus sniffing devices are available on the market which record the data exchange between ECUs during the test drives and enable subsequent verification and error analysis in the laboratory. 10.000 bus message frames per second and 2GByte of log data per hour, respectively, are typical quantities recorded during a test drive. After an 8 hours test drive the test engineer has to transfer 16 GBytes of log data out of the logger, which is a time-consuming process. In cooperation with a manufacturer of automotive data loggers we researched on methodologies to pre-filter the log data on-device in real-time. The intent was to provide the test engineer with some compact summary of the test drive, such as message sequences that might be of interest and that could be downloaded much quicker than the full data record. Our challenge was to choose a formalism that enables the intuitive description of even complex filter scenarios and allows for efficient implementation on the logging device, which has clear resource limitations. We designed a component-based library consisting of several SystemC modules that implement typical building blocks usually used in filter applications. Indeed, the library simplifies the development of filter algorithms. However, some C++ code is necessary to instantiate and to interconnect the library components. As we did not want to require the test engineer to have programming knowledge, we designed a simple modelling language that allows for graphical modelling of a filter implementation. We developed an appropriate graphical editor and a code generator that transforms a graphical model representation into a piece of C++ code that interfaces with the SystemC library and implements the modelled filter algorithm.

This paper is organised as follows: In section II, we outline SystemC, GMF and oAW which are the foundation of our contribution. Section III will point out related work that deals with the visualisation and editing of SystemC modules and models. The actual realisation of the graphical language and the editor build the main part of the paper and will be discussed in section IV. A summary and conclusion will be given in section V.

II. Applied Tools and Technologies

A. SystemC

SystemC[1] [2] is technically a class library written in the programming language C++, but can also be thought as of an embedded domain-specific language for modelling event-driven systems. It enhances C++ with the concepts of event-

[1]SystemC, `www.SystemC.org`

978-0-7695-4038-2/10 $26.00 © 2010 IEEE
DOI 10.1109/ISORCW.2010.33

IEEE computer society

triggered processes, signal-based data processing and structural decomposition of complex systems. Its primary intent is modelling of hardware/software co-design systems in the domain of electronic design automation (EDA). Each system component is represented by a module instance which itself may symbolise a complex subsystem, consisting of many inner module instances. A module instance is described by a module definition. The definition comprises a behavioural description (an implementation) and an interface, which is a contract on how the module interacts with the outside world. The interface consists of a list of typed ports that serve as communication endpoints when module instances are inter-connected. Any communication between modules must must involve one of their ports[2]. A port models the generic concept of a communication entity, but does not prescribe what information is transferred or how this should be accomplished. Amongst others, the SystemC standard library provides specialised ports that represent a signal flowing into or out of a module. Signals in SystemC are quantities of some typically primitive type (such as *int*) that may change their values at discrete[3] time instants. Depending on its causality (input, output or input/output), the signal port provides accessor methods to read and/or update the current signal value. Other kinds of ports model the concept of a software interface. An interface can either be provided (implemented), which is then called an *export*, or be required by a certain port (in which case it does not have any special name).

Obviously, there is an intuitive way of representing the structure of a SystemC model graphically, which can be thought as of an electrical circuit. Many graphical modelling tools from different domains, such as circuit simulators, Matlab/Simulink, ASCET or Dymola employ the simple paradigm of providing the user with some block library whose elements the user can instantiate simply by drag-and-drop. Each block provides some "pins" that serve as wiring end-points. As our data filtering framework is based on SystemC, our goal was to apply the block paradigm and design a custom graphical editor without abusing some existing commercial tool. The Graphical Modelling Framework (GMF) and openArchitectureWare (oAW) turned out to be suitable foundations for that intent and will be outlined in the next sections.

B. GMF

The Graphical Modelling Framework (GMF) [3] is a subproject of the Eclipse project and aims to support the development of graphical editors. It provides a generative component and a runtime infrastructure. The latter combines the Eclipse Modelling Framework (EMF) for the representation of models and the Graphical Editing Framework (GEF) [4] for the user interface. The generative component allows for the automatic generation of graphical editors out of a domain model, a graphical definition model, a tooling definition model and a mapping model. The generated code is based on the capabilities of the GMF runtime and additionally uses the GEF directly in some cases. The EMF is built upon Ecore, which is based on Essential Meta-Modelling Framework (EMOF), a subset of UML's standardised meta-modelling architecture MOF.

C. oAW

openArchitectureWare (oAW) [5] [6] is a tool-chain on top of the Eclipse platform which has been incorporated into the EMF by now. oAW aims to support the realisation of projects according to Model-Driven Software Development (MDSD) principles. A family of languages supports the description of

- model-to-model transformations: Xtend,
- model-to-text transformations (e. g. code generation): Xpand,
- model validation: Check,
- grammars of textual domain-specific languages: Xtext. Xtext also allows for parser generation and the generation of Eclipse editor components, e.g. with language-specific syntax highlighting.

III. Related Work

Similar work affects the visualisation and graphical editing of SystemC components and models. Amongst commercial tools, such as ModelSim[4] (MentorGraphics) or Visual SC Designer (Oustech)[5], miscellaneous academic projects focus on different aspects of processing SystemC-based models. The SystemC Environment for Eclipse (SCE) [7] extends the Eclipse platform with an IDE for textual editing and visualisation of SystemC models. SystemCXML [8] is an open-source parser that extracts the structural information of SystemC models with the help of Doxygen and converts them into an XML description. An exemplary backend transforms the result into a visual graph structure. Work from Christian Genz, Rolf Drechsler et. al. [9] [10] ranks among the most elaborate approaches concerning the visual representation of SystemC code. The syntax tree gained from parsing is analysed regarding model structure and behaviour. An algorithm walks through the hierarchy of inheritance, looking for SystemC types. The results may be graphically visualised. The intent is to cope with the complexity of large projects by leaving out non-relevant information and focusing on the overall dependency structure. General considerations on the design of modelling languages for existing libraries and frameworks can be found in [11].

IV. Implementation

The standard GMF workflow requires four different input documents which provide an adequate description to generate a fully-functional graphical editor:

[2]Indeed, implementing some different communication mechanism (e.g. directly calling a public method of some module) is possible in the C++ language, however this would be a violation of SystemC design rules.

[3]Continuous-time signals are in the scope of SystemC-AMS, which is not considered in this contribution.

[4]`http://www.model.com`

[5]`http://www.oustech.com`

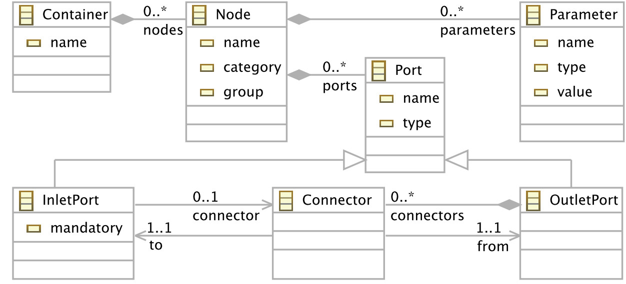

Figure 1. Domain model of the data analysis library.

- The *Domain Model* specifies the underlying metamodel. Each model that can be created with the editor is a specific instance of that metamodel.
- The *Graphical Definition Model* specifies the geometric elements that make up the graphical representation of a model, such as rectangles or text labels.
- The *Tooling Definition Model* specifies the items inside the editor's toolbar and menus.
- The *Mapping Model* links the first three models together. It specifies which geometric elements to use for a specific component in metamodel as well as what metamodel component should be instantiated when the user selects a specific item in the toolbar.

A. Fully Generative vs. Hybrid Approach

Following a fully generative approach, each of the above mentioned models is completely specified. The graphical editor can then be generated in a fully-automatic manner, with almost no code that has to be written manually. This approach implies that all library modules are reflected in the domain model, with one metaclass for each module and appropriate metaclasses for each port and each parameter of each module. However, this is not tractable due to two reasons:

- Persistency of the domain model: More than 300 modules are contained in the library, which itself is subject to further development. With every new release of the library, the domain model would have to be changed and the editor would have to be re-generated.
- Code-size of the generated implementation: If we assume on average 5 artefacts (ports and parameters) per module class, the domain model would contain more than $300 + 5 * 300 = 1800$ metaclasses. As the GMF generates multiple Java classes per metaclass, more than 9000 Java classes would be generated in total (a detailed derivation of that estimation can be found in [12]).

Consequently, we chose a hybrid approach: the domain model captures only the generic aspects, and the editor configures itself at runtime. This is possible because the GMF-generated editor code provides hook-functions that allow arbitrary custom enhancements.

B. The Domain Model

Figure 1 shows the domain model that represents the components of the data analysis library. The *Container* metaclass describes the filter application which contains several module instances, each modelled as *Node*. Each module requires a set of arguments that are supplied during its constructor call. This fact is modelled by the metaclass *Parameter*. Three further attributes are needed to describe a module:

- *id* is the class name of the module that should be instantiated from the analysis library.
- *instance* is a user-defined instance name that can be used to generate a variable name for the module instance.
- *category* is an annotation tags that is supplied by the developer for each module in the analysis library. The intent is to group modules with similar functionality, in order to provide the user with a well-arranged toolbar.

The ports of each module fall into the categories of inputs (*InletPort*) and outputs (*OutletPort*). This distinction is well-defined for signal-type ports where there is a clear indication whether the signal "flows" into or out of the module. For non-signal ports that provide or require some abstract software interface, it is not well-defined in which category they should fall. In that case it is up to the library developer to assign a certain port either to the input or the output port group. His choice should depend on some "intuitive flow" of information. The attribute *mandatory* applies to input ports and indicates whether an incoming connection is necessary or optional. Each port has a certain type that determines whether the port represents a signal, provides an interface or requires a certain interface. The data type of the carried signal or accordingly the interface name complete the type information, which is encoded into the string attribute *type* using a special naming convention.

Connections between module-ports are modelled by the *Connector* class. A connection goes always from one *OutletPort* to one *InletPort*. To allow for floating connections (when the user starts dragging a connection), each *Connector* is associated with *at most* one *OutletPort* and *at most* one *InletPort*, instead of *exactly one*. Two further modelling rules are apparent in the domain model diagram, namely:

- An *InletPort* may have at most one incoming connection.
- An *OutletPort* may have an arbitrary number of outgoing connections.

Obviously, there should be some more rules that disallow incompatible connections between ports with different types. That kind of rule belongs to the part of model verification and is not made explicit here, as it would increase the complexity of the domain model by orders of magnitude. We will go into detail in section IV-F.

C. The Graphical Definition Model

The graphical definition model comprises an assembly of drawing primitives, such as *Rectangle*, *Polyline* or *Label*. The visual representation of a domain model instance is completely composed of those elements in the graphical definition. A

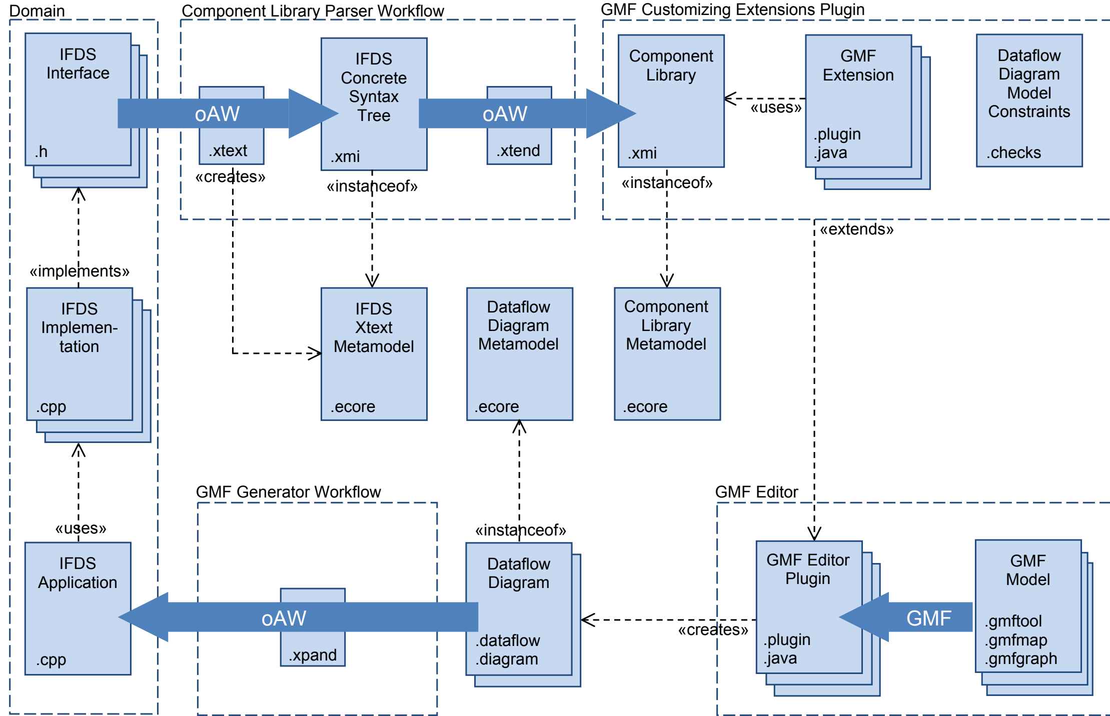

Figure 2. Relationships between all involved artefacts.

presentation of the graphical definition model that we use in the developed editor would hardly provide some interesting insights. Instead, we refer the interested reader to one of the numerous GMF tutorials [13] [14] [15].

D. Tooling Definition and Mapping Model

The tooling definition model lets the user define information like palettes and menus for the graphical editor, whereas the mapping model links the information of all previous models together. The editor's tool's should list the module classes that are defined in the data analysis library. As we pursue the hybrid approach, only a partial specification of tooling definition model and mapping model is possible at this point. The actual behaviour, such as configuring the toolbar, is implemented using manually-written code. To accomplish this task, we make use of GMF extension points that provide a structured means for injecting code into the generated framework.

E. The Overall Approach

Figure 2 subsumes the overall approach. The domain model is shown as *Dataflow Diagram Metamodel*. The lower right part symbolises the different models that enable code generation for the graphical editor. We also mentioned that we had to extend our editor with custom code in order to construct the tooling definition dynamically. This is shown in the upper right part of the diagram. To the left, the application domain is shown[6]. On the one hand, it consists of the data analysis library which can be separated into the parts *interface* (a C++ header file containing all module declarations) and *implementation*. On the other hand, a specific data analysis application makes use of that library.

The following sections will focus on two further interesting aspects, namely the model verification (section IV-F) and the extraction of a components library which is the basis for the tooling definition (section IV-G). We will leave out the aspect of code generation, which is depicted as "GMF Generator Workflow" in figure 2. Contrary to parsing, the conversion of a model into runnable C++ code is a rather trivial task. It was implemented using the Xpand language, which is part of oAW.

F. Model verification

With model verification, we mean the determination if some given instance of the domain model is valid. We define the validity of a model to be fulfilled if the represented SystemC model passes successfully through the elaboration phase[7]. There are basically two reasons that might cause the elaboration to fail:

- *Invalid arguments*: Each module class needs a certain set of parameters that will be supplied to its constructor dur-

[6]IFDS is an abbreviation for "Intelligentes Fahrzeugdatenanalysesystem", which is the name of the project in whose context this work was done. An english translation could be: Intelligent vehicle data analysis system.

[7]During elaboration, all module instances are constructed and interlinked.

ing elaboration. Given some invalid value, the constructor may abort the elaboration phase.
- *Invalid or missing connections*: Only compatible ports may be connected. Signal outputs may only be linked with signal inputs and the data types of the signals have to be the same. “Required interface” ports may only be connected to “Provided interface” ports and both interfaces have to be equal. Some ports require a connection, some do not, and some allow for multiple connections.

Such constraints can hardly be expressed in the domain model. A more convenient way is to use an appropriate language. We formulated a set of accordant rules in the Check validation language which is part of oAW. Check is similar to the *Object Constraint Language (OCL)* which belongs to the UML standard.

G. Parsing the Data Analysis Library

The goal was to transform the analysis library into some suitably structured description of each module. As the library is implemented in C++, the extraction of all required information to feed the domain model is not a trivial task. Manual extraction is not tractable, as the library contains more than 300 different modules. We chose to parse the library in order to obtain a descriptor for each module that comprised the following information:

- class name,
- constructor arguments and their respective types,
- a category annotation (if present),
- the set of input and output ports and their respective types.

To simplify the parsing task, the library developer is required to keep on a certain coding style. Figure 3 shows an exemplary module declaration in the data analysis library. As defining multiple constructors is perfectly legal in C++, the developer must use special macros to indicate which constructor should be used for the extraction of arguments. With further macros, the developer can assign a category and a group attribute to each module class. These are intended to provide the designer with a well-arranged hierarchical module list in the editor. The set of input ports and output ports have to be declared in separate data structures which are supplied as template arguments[8] to *OperationModule*. In the given example, *BusFilterInput* defines the set of input ports, whereas *CANFilterOutput* defines the set of output ports. Both are simple *structs* which are not listed here for the sake of brevity. The class *OperationModule* is a descendant of *sc_module*, the SystemC base class for modules.

```
class CANFilter :
    public BusFilter,
    public OperationModule<BusFilterInput,
                           CANFilterOutput>
{
public:
        CANFilter( string Alias );
        [...]
    BEGIN_AUTOMATION( CANFilter )
        ARGUMENT( Alias, string )
    END_AUTOMATION

    CATEGORY("Import")
};
```

Figure 3. Exemplary module declaration in the data analysis library.

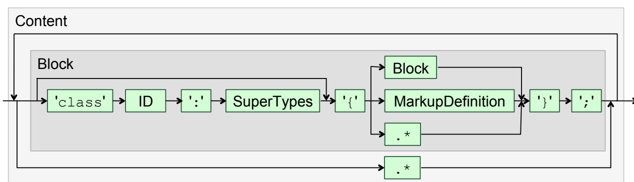

Figure 4. Syntax diagram illustrating a grammar with accept-anything option.

Unfortunately, the C++ grammar is still complex. On the other hand, it is not necessary to understand the full grammar when only some dedicated information is to be extracted. Using Xtext, we implemented a parser that basically recognises the hierarchy of inheritance when parsing C++ classes and includes some special handling for the mentioned macros. This was possible with a considerably small subset of the C++ grammar. Figure 4 shows the rule for parsing a class declaration. To skip structures that are not included in the grammar, we added an “accept anything” token (depicted as .* in the diagram) to each production. This will result in an ambiguous grammar, which the Xtext parser can handle with its backtracking capability.

V. Conclusion

Given a domain-specific data analysis library based on SystemC, we showed the design of a suitable modelling language and an appropriate graphical editor. We explained the underlying domain model and pointed out how to write a parser using Xtext that extracts all required information from the library to configure the editor at runtime. The implementation of such an editor from scratch is usually a sophisticated task, consuming man-years of implementation efforts. Using the Eclipse platform, GMF and oAW it was possible within less than half a year. Figure 5 shows a screenshot of the editor. Its implementation comprises about 5000 lines of Java code and 2000 lines of script codes (Xtend, Xpand, Check and Workflows). Approximately 3600 lines of Java code are generated, 1400 manually written lines were necessary for custom adaptations.

The learning curve for developing a custom graphical language with GMF is very stiff and contains many pitfalls. On the other hand, the technology behind is a very powerful and complex framework. Investing some time for diving into the depths of GMF will be worth the trouble on any account.

[8]This practice is actually not intended to ease the parsing but due to some implementation considerations.

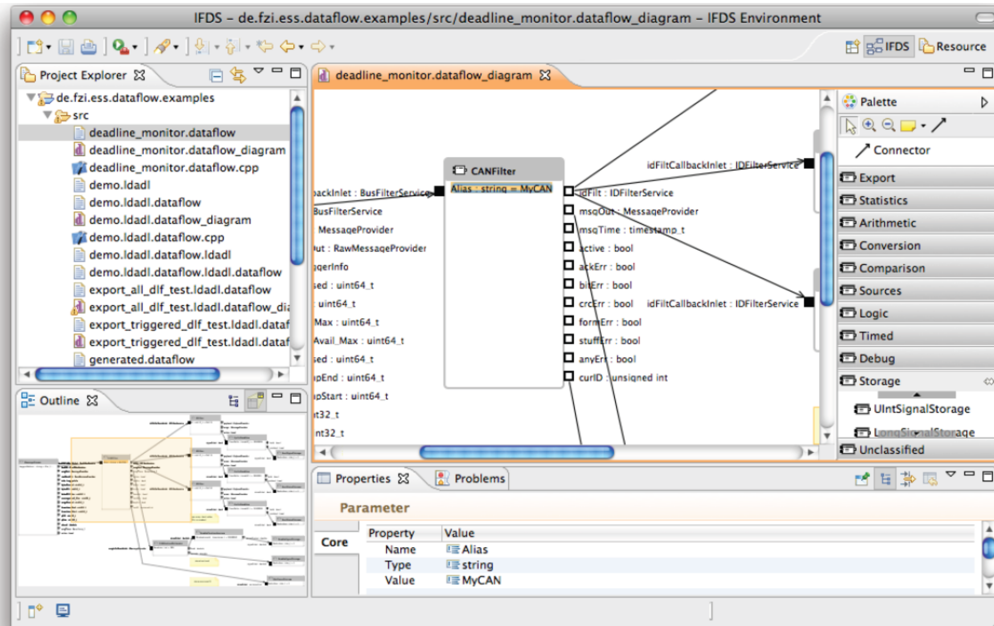

Figure 5. Screenshot of the graphical editor.

Acknowledgment

The authors would like to thank Dr. Weiß, Mr. Blaum and all other staffs of the log device manufacturer X2E[9] for the good cooperation and for organising insightful discussions with test engineers.

References

[1] R. Schmidgall, *Bussysteme in der Fahrzeugtechnik*. Vieweg+Teubner Verlag, 1 2008. [Online]. Available: http://amazon.com/o/ASIN/3834804479/

[2] *Open SystemC Language Reference Manual*, IEEE, December 2005. [Online]. Available: http://standards.ieee.org/getieee/1666/download/1666-2005.pdf

[3] R. C. Gronback, *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*, 1st ed. Addison-Wesley Professional, 3 2009. [Online]. Available: http://amazon.com/o/ASIN/0321534077/

[4] *Eclipse Development Using the Graphical Editing Framework And the Eclipse Modeling Framework (IBM Redbooks)*. IBM.Com/Redbooks, 2 2004. [Online]. Available: http://amazon.com/o/ASIN/0738453161/

[5] G. Pietrek, J. Trompeter, J. C. F. Beltran, B. Holzer, T. Kamann, M. Kloss, S. A. Mork, B. Niehues, and K. Thoms, *Modellgetriebene Softwareentwicklung - MDA und MDSD in der Praxis*, 1st ed. Entwickler.Press, 6 2007.

[6] S. Efftinge, P. Friese, A. Haase, D. Hübner, C. Kadura, B. Kolb, J. Köhnlein, D. Moroff, K. Thoms, M. Völter, P. Schönbach, M. Eysholdt, and S. Reinisch, "openarchitectureware user guide - version 4.3.1," December 2008. [Online]. Available: http://www.openarchitectureware.org

[7] S. Kramer, "Entwicklung einer systemC-umgebung für eclipse," 09 2005.

[8] D. Berner, J.-P. Talpin, H. D. Patel, D. Mathaikutty, and S. K. Shukla, "SystemCXML: An extensible systemC front end using XML," in *FDL*. ECSI, 2005, pp. 405–409. [Online]. Available: http://www.ecsi-association.org/

[9] C. Genz and R. Drechsler, "System exploration of systemC designs," in *ISVLSI*. IEEE Computer Society, 2006, pp. 335–342. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ISVLSI.2006.87

[10] R. Drechsler, G. Fey, C. Genz, and D. Große, "Syce: An integrated environment for system design in systemc," *Rapid System Prototyping, IEEE International Workshop on*, vol. 1, pp. 258–260, 2005.

[11] J.-P. Tolvanen, "Creating a domain-specific modeling language for an existing framework," *Methods & Tools*, vol. 14, pp. 18–28, 2006.

[12] A. Rentschler, "Entwurf einer grafischen domänenspezifischen modellierungssprache für ein filterbasiertes datenanalyseframework," Diplomarbeit, Embedded Systems and Sensors Engineering (ESS), FZI Forschungszentrum Informatik Karlsruhe, January 2010.

[13] C. Aniszczyk, "Learn eclipse gmf in 15 minutes," 2006. [Online]. Available: http://www.ibm.com/developerworks/opensource/library/os-ecl-gmf/

[14] Eclipsepedia, "Eclipsepedia gmf tutorial," 2008. [Online]. Available: http://wiki.eclipse.org/index.php/GMF_Tutorial

[15] B. Kolb, S. Efftinge, M. Voelter, and A. Haase, "Graphical modeling framework." [Online]. Available: http://www.voelter.de/data/articles/ix-gmf2.pdf

[9]X2E: http://www.x2e.de