Improving Maintenance by Creating a DSL for Configuring a Fieldbus

Mathijs Schuts
Philips
Best, The Netherlands
mathijs.schuts@philips.com

Jozef Hooman
TNO-ESI, Eindhoven & Radboud University,
Nijmegen
The Netherlands
jozef.hooman@tno.nl

Abstract

The high-tech industry produces complex devices in which software plays an important role. Since these devices have been developed for many decades, an increasing part of the software can be classified as legacy which is difficult to maintain and to extend. To improve the maintainability of legacy components, domain specific languages (DSLs) provide promising perspectives. We present a DSL for creating configuration files that describe the topology of a fieldbus. This DSL improves the maintainability and extensibility of a legacy component. Compared to the current way-of-working, the configuration files generated by the DSL are of higher quality due to the concise representation of DSL instances and additional validation checks. To raise the level of abstraction even more, we have created a second DSL which allows a concise description of system configurations and the generation of topologies.

Categories and Subject Descriptors D.2.7 [*Distribution, Maintenance, and Enhancement*]: Restructuring, reverse engineering, and reengineering

Keywords Legacy Software, Software Maintenance, Domain Specific Languages, Industrial Application

1. Introduction

The software life cycle consists of a number of distinct phases (Vyatkin 2013). The first phase of an software system is its construction. During construction the system is specified, designed, build and tested. After acceptance by the customer it is taken into use and needs to be maintained by the constructor of the system. The system is taken out of use when it has reached end-of-life and it is replaced by its successor.

(Bennett et al. 2000) propose a staged model in which the maintenance phase is split up into an evolution stage and a servicing stage. In the evolution stage, changes are made to cope with changes in functional and non-functional requirements. If evolutionary changes are no longer possible, the system moves to the servicing stage. In the servicing stage only service patches are applied to keep the system alive.

In the high-tech industry, complex software intensive systems have been produced for many decades. While the systems for the customer might be new, they are constructed by a collection of many components from which some can be characterized as legacy components (Breivold et al. 2011). If one projects the software life cycle on the individual components of a system then legacy components are components that are in the maintenance phase of the software life cycle (Froschauer et al. 2008).

According to (Vliet 2008) 50-75 % of all effort on a software system is spent in the maintenance phase. Hence, for improving the software life cycle most can be gained by improving the effort that needs to be spent in the maintenance phase.

Legacy components are often the result of decades of development with dozens of man-year invested. Creating new implementations would require a similar investment and typical more resources than keeping the legacy implementation alive; scarce resources that could also be used to implement product innovations.

Most research on Domain Specific Language (DSL) (Van Deursen et al. 1998) focuses on the development of new components (Cao et al. 2009). The aim of our work is to investigate whether a DSL could improve the maintainability and extensibility of a legacy component in the context of high-tech systems. The aim is to keep components longer in the evolution state. We would like to get an answer to the following questions:

- Is it financially feasible to extend the life of a legacy component using a DSL?

DSM'16, October 30, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4894-2/16/10...$15.00
http://dx.doi.org/10.1145/3023147.3023152

- What are the pros and cons of using a DSL compared to the current way of working?

In this paper we try to answer these research questions based on experiences at Philips. We have created a DSL to generate topology files for a fieldbus used in systems for image guide therapy. These systems are used during minimally invasive medical treatments, such as the treatment of cardiology and vascular diseases. An example is the interventional X-ray system shown in Figure 1, where X-ray images support minimally-invasive medical procedures such as placing a stent via a catheter.

Figure 1. Interventional X-ray System

Given the long history of these systems and the frequent need for changes to support new medial procedures, it is important to keep the software architecture and its components flexible and extensible. Hence, ways need to be found to manage legacy implementations.

An example of such a legacy implementation is the component which uses a fieldbus. A fieldbus is an industrial network for real-time distributed control. It is used to manage and control the position of the X-ray beam with respect to the patient. This can be done by moving the table or the stand which holds the X-ray generator and detector. The table and stands come in many configurations. For all combinations, a separate fieldbus configuration needs to be created. In the foreseeable future about 2000 different topology configuration files are needed. For this reason, we investigated the use of a DSL to generate the configuration files.

This paper is organized as follows. We relate our work to the literature in Section 2. Section 3 describes our industrial case. The developed DSL is presented in Section 4. Section 5 describes validation checks that have been added to the language. To raise the abstraction level, a second language is created, as described in Section 6. Section 7 contains an overview of the results and an answer to our research questions.

2. Related Work

In the literature there are three main directions for improving maintainability and extensibility challenges: model-based reverse engineering, object orientation and domain specific languages.

An application of model-based reverse engineering is described in (Bergmayr et al. 2016). The fREX tool reverse engineers the executable behaviour of a software system. The result is an fUML model that can be used to generate a new implementation while removing obsolete technologies from the code base. An alternative way of improving the maintainability of legacy software is to transform the legacy implementation. Rascal (Basten et al. 2015) is a tool created for meta-programming, e.g., reverse engineering and re-engineering of legacy software.

Object-oriented analysis and design (Sarnath et al. 2010) has been advocated as a way to create maintainable systems using variability and commonality analysis (Coplien et al. 1998) and design patterns (Wolfgang 1994). (Deursen 1997) describes the differences between object-oriented frameworks and DSLs and provides criteria to chose between the two approaches. The paper also contains a DSL for the financial domain to hide legacy libraries written in COBOL.

According to (Ward 1994), DSLs improve the maintainability of new software due to code size reduction and improved readability. In (Batory et al. 2002) a DSL for a command-and-control simulator for Army fire support has been defined. They report on improvements of the maintainability and extensibility by raising the abstraction using domain concepts.

In contrast to the work mentioned above, our work concentrates on improving the maintainability of legacy software. Related to our work, (Fehrenbach et al. 2013; Erdweg et al. 2014) try to improve the maintainability of large software systems by an evolutionary process that can be used to incrementally refactor an implementation and raise abstraction using an extensible programming language called SugarJ. SugarJ enables the use of multiple small embedded DSLs, e.g. they created DSLs for XML and SQL. Major benefit of these embedded DSLs is that instances can be statically validated, e.g., to check whether an XML file has the right closure. The embedded DSLs are placed in libraries, to enable incremental introduction of generated code in a code base. By importing the DSLs in a source file, the DSL can be used only when required.

At a large Austrian electricity company with more than 140 power plants, a DSL has been developed for a legacy software system. The system describes schedules that are used for trading electricity between companies (Sobernig et al. 2014). They conclude the following: “This project report shows that a DSL-based system refactoring can provide benefits in terms of reduced code redundancy for an improved maintainability of a code base.”

Different from the related work above, we used DSLs to improve the use of configuration files instead of source code. Tolvanen et al. (Tolvanen et al. 2005) describe over 20 industrial applications of DSLs, including the generation of configuration files. They observed that DSLs are beneficial for design guidance and early error prevention or detection. In addition, they report that DSLs increase productivity due to the raised level of abstraction.

Software Product Line Engineering (SPLE) addresses modelling and analysis of commonality and variability. System configurations can be generated from the resulting models. Berger et al. (Berger et al. 2013) conducted a survey on the industrial usage of variability modelling. They conclude that the SPLE community focuses on creating methods and tools for new systems and a shift might be needed towards support for legacy software.

3. Configuring the X-ray System

The interventional X-ray system depicted in Figure 1 consists of a number of building blocks such as the patient table, one or two stands which hold an X-ray generator and a detector, and a stand mover that can position the stands away from the table. A building block has a number of axes that are used to position the X-ray beam with respect to the patient. The axes are controlled by motion drives which are connected by means of a fieldbus.

For each building block there are a number of variations, e.g., they may have a different number of axes or might come from different third-party vendors. For example, the table can have one, two, three, four, five or six moveable axes. Moreover, there are many possible combinations of these building blocks, leading to many systems configurations depending on the wishes of the customer. The fieldbus needs a topology description for all these combinations. In addition, past and future configurations need to be supported.

Figure 2 depicts an example network topology. A number of components can be distinguished: a computer, a Hub, a stand, a table, and a stand mover. The computer runs the fieldbus master and also hosts the motion application. The fieldbus Hub is an embedded device that supports the use of tree topologies. The Hub is optional; for instance, a configuration with only a table does not need a Hub. Because of the Hub, different cable sets toward the table and stand(s) can be bundled. A stand consists of two motion drives, each controlling a number of axes, and a node that is used to prevent collisions between the stand and other objects in the room. The table has a motion drive for every axis. So the table has up to six motion drives. The stand mover has a motion drive that controls a number of axes to move the stand away from the table.

Every node or motion drive in the fieldbus has a certain type. A type is a combination of a vendor and model. The concept is that every motion drive in the system can be replaced by a compatible type from another vendor. The Hub has two nodes of TYPE_A1 and TYPE_A2. A stand uses devices of TYPE_D and TYPE_E. The table uses motion drives from TYPE_B and the stand mover from TYPE_C.

The nodes of the Hub have four ports. The node of TYPE_A1 is connected to a master via port A. The nodes TYPE_A1 and TYPE_A2 are internally connected via port D of TYPE_A1 and port A of TYPE_A2. The motion drives have two ports: A and B. At the end of a branch it is possible that a port is not connected to another device.

The arrows in Figure 2 describe the flow of messages over the fieldbus. The master sends a packet to the node it is connected to. A packet consists of different fields. Every node in the network has its own field. Every node reads and writes its field of the message. At the end, the master receives a message with all updates of the nodes. The master sends a message with a time interval of 2 milliseconds.

During the start-up of the network, the master reads a configuration file that describes the physical network. The master then starts the network by programming the nodes. The nodes need to be programmed such that they know their field, the elements of this field, and the address of the elements.

To create the configuration files of the master, a commercial tool is used. A tutorial of 29 pages describes how a configuration file needs to be created. The first two pages explain how to install the tool and some basic explanation of the topologies used in the system. The remaining 27 pages describe what needs to be filled in when making network topologies for the system. Currently, the system needs about 40 different topology configuration files. All files are created with the commercial tool.

To test the configuration files, it is too expensive to physically build 40 different complete systems, in terms of effort, lead time and system cost. Hence, for testing a lab set-up is created. In the lab set-up the master is started using a stripped version of the motion application. The nodes, such as the motion drives, are placed on a board. Using the board it is possible to reroute the cables to test different configurations.

The configuration files are formatted using eXtensible Markup Language (XML). Today, the simplest configuration file consists of 2147 lines and the most extensive configuration contains 13128 lines.

4. DSL for Fieldbus Configurations

The number of configurations explodes when multiple suppliers and motion drive types have to be supported. The commercial tool that is used to create network topologies has many settings and options. However, for the system configurations we want to describe, the settings and options are always the same. The only variation between the different configurations is the number of nodes, their type and how they are connected to each other.

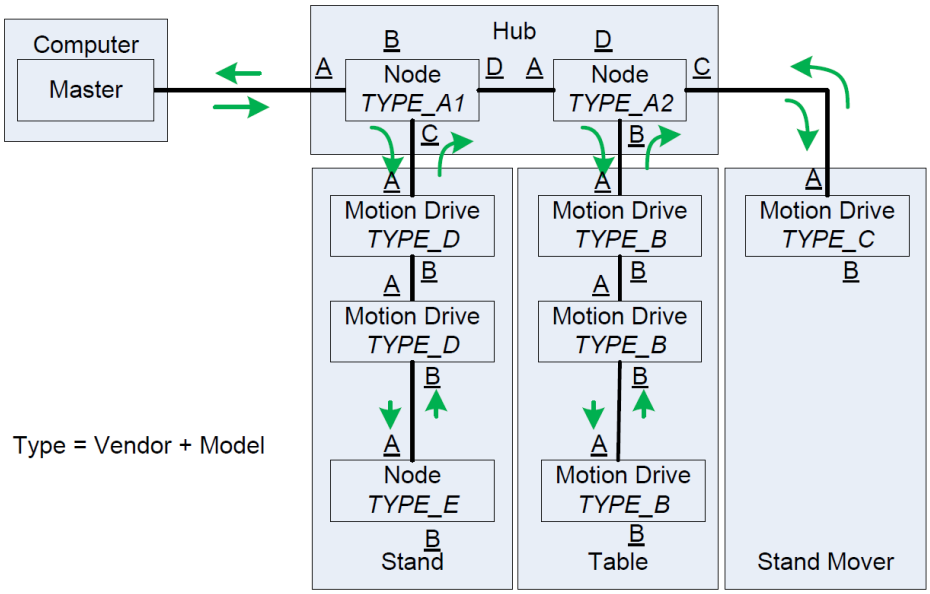

Figure 2. Fieldbus Topology

To handle these differences, a DSL is created to generate configuration files for the fieldbus. The DSL only describes the variations; the fixed options and settings are defined by the generator. For creating the language, Eclipse (Geer 2005) is used with the Xtext and Xtend plug-ins (Bettini 2013). The choice for this language workbench was pragmatic because the Philips engineer who created the language was familiar with these tools. Figure 3 presents an example instance of the language. Each topology needs to have a name which is used as the file name for the generated configuration. After the *network* keyword the ordering of the nodes is described. The language has predefined node types. In the language used at Philips the types have more meaningful names, but for confidentiality reasons we use abstract names in this paper.

```
topology
name          ExampleTopology
network       TYPE_A1 <-> TYPE_A2 <-> TYPE_B
                  <-> TYPE_B <-> TYPE_B
              prev TYPE_A2 port C TYPE_C
              prev TYPE_A1 port C TYPE_D
                  <-> TYPE_D <-> TYPE_E
```

Figure 3. Example Topology Description

Figure 3 presents the network topology of the system configuration depicted by Figure 2. The example presents a single network topology, but typically a DSL instance consists of multiple topology definitions. The master is always present in a topology and hence omitted in the DSL instances. The nodes TYPE_A1 and TYPE_A2 of the Hub are connected to each other. Because the connection between TYPE_A1 and TYPE_A2 is hardwired inside the Hub, this information does need to be provided when creating a DSL instance. The other devices (nodes and motion drives) are implicitly connected via port A to port B of the previous device. Hence, this information does not have to be described in a DSL instance. With "prev TYPE_A2 port C TYPE_C" a branch is created by connecting port C of TYPE_A2 to port A of TYPE_C. Similarly, a branch is created from TYPE_A1 to TYPE_D.

From every topology in a DSL instance, an XML configuration file is generated. The XML configuration file generator has been defined using multi-line template expressions (Bettini 2013). The settings that are always the same for network topologies are part of the template. The configuration settings that vary, e.g., the position of the nodes and the fields, are calculated and filled in the right position. A DSL instance of 5 lines describing an existing topology leads to an XML file of 13128 lines. The output of the generator has been validated by generating ten existing network topologies and comparing the configuration files with the ones that are produced with the commercial tool.

5. Checking the Topology

To prevent that the user of the language makes faults in describing network topologies, validation rules have been added to check the validity of a network topology. For example, it is physically impossible to connect two branches of motion drives to the same port of the Hub. Figure 4 shows the validation rule that a port of a specific node can only be used once within a topology. The rule checks for every topology and for every pair of nodes which have the same predecessor that they are connected to different ports of this predecessor.

```
@Check
def CheckTagPortUnique(Topology topo) {
	for (var i = 0; i < topo.nextNode.length; i++) {
		var nodeA = topo.nextNode.get(i)
		for (var j = 0; j < topo.nextNode.length; j++) {
			var nodeB = topo.nextNode.get(j)
			if (nodeA.prev != null && nodeB.prev != null) {
				if (i != j && nodeA.prev.name == nodeB.prev.name &&
						nodeA.portOfPrev == nodeB.portOfPrev) {
					error("A port can only be used once", null)
				}
			}
		}
	}
}
```

Figure 4. Validation Rule

In addition to the rule in Figure 4, there are validation rules to check that:

- Within a DSL instance, a typology has a unique name.
- Within a topology, the types TYPE_A1 and TYPE_A2 are paired. TYPE_A1 and TYPE_A2 are either both present or both not present.
- Within a topology, TYPE_A1 comes before TYPE_A2 and is connected to TYPE_A2.

Using the commercial tool there are many ways to produce a faulty configuration file. The DSL and the above described validation rules provide enough confidence in the validity of the produced configuration files. Creating a hardware set-up in the lab to check the correctness of a network configuration is no longer needed.

6. DSL to Describe System Configurations

Once a year a new system release is being made. Because the system is a medical device, for such a release all functionality needs to be verified and validated using strict rules of authorities. Hence, all supported network topology configurations are part of the annual release.

When in the future many network topology configuration files are needed, it is still a gigantic and error-prone task to create them all. For this reason, we investigated the possibility to further raise the abstraction level. The result is a second language to represent system level configurations and generate a DSL instance of the previously described network topologies. The system configuration DSL consists of two parts: the first part describes building block definitions and the second part describes system configuration descriptions which consists of combinations of the build blocks.

Figure 5 shows a fragment of the building block definitions. Every building block has a unique id. Building blocks have a type, for instance, a stand can be based on the floor or the ceiling. Also the table is a building block.

```
building blocks
building block
id            BB1
type          CeilingStand
vendor(s)     VENDOR_A

building block
id            BB2
type          Table
vendor(s)     VENDOR_A VENDOR_B
```

Figure 5. Building Block Definitions

Depending on which options a customer chooses, the table can have one up to six motorized degrees of freedom. Hence, there are six combinations of motion drives for a table. For certain building blocks, nodes from multiple vendors can be used. Recall that in the topology descriptions of the first DSL, nodes of a certain type are used. The relation between vendors and types has been encoded in the generator, e.g., VENDOR_A corresponds to types TYPE_C and TYPE_D. Also information about which port of a building block needs to be connected is hard coded into the generator because this will never change.

```
configurations
configuration
name                Configuration1
building blocks     BB1

configuration
name                Configuration2
building blocks     BB1 BB2
```

Figure 6. System Configuration Descriptions

A fragment of two system configuration descriptions is shown in Figure 6:

- Configuration1 describes a system configuration consisting of a ceiling stand with motion drives of vendor A. This is a very basic example that results in a single network topology.
- Configuration2 consists of a ceiling stand and a table. The table can have from 1 up to 6 motion drives and each of these motion drives can be either from VENDOR_A or VENDOR_B. If a table has one motion drive it can be of two different vendors, leading to two network topologies. If a table has two motion drives, then four different combinations are possible, etcetera. When we sum all possibilities, we get 126 network topologies.

The Configuration2 example makes the need for the system configuration language clear. The number of network topologies grows exponentially with the number of different vendors that need to be supported.

Using the system configuration language, the generation of the configuration files takes a two step approach. In the first step, an instance of the system configuration language generates an instance with network topology descriptions. From the network topologies, XML configuration files are generated.

7. Concluding Remarks

We have presented an approach to improve the maintenance of a legacy component using two DSLs. The first DSL describes network topologies from which XML files are generated for the master of a fieldbus network. Because of the expected large number of topologies in the future, we further raised the abstraction level by means of a second DSL that describes system configurations and generates an instance of the first network topology DSL. The experiences with these DSLs at Philips leads to the following observations on the questions posed in Section 1.

- *Is it financially feasible to extend the life of a legacy component using a DSL?*
  We calculate the Return On Investment (ROI) for the presented DSL. First we compute the required investment for the DSL approach. To learn the domain and the structure of the configuration XML files took 10 hours. The construction of the DSL took about 40 hours including the creation of the validation rules. In total it took about 50 hours to create the DSL. We expect new vendors in the future and estimate that it will take 30 hours to extend the DSL framework with support for these vendors.

  Next we compare the DSL approach with the current way-of-working. We estimate that approximately 8 hours are required to manually create a topology file, build a physical hardware set-up and test if the master can start the fieldbus. Of the 8 hours approximately half an hour is needed required to create the topology file. If we multiply these 8 hours of work with the the 2000 network topologies we need in the future, it takes 16000 hours which is 10 man-years.

  Using the DSL we expect it takes around 20 hours to create instances describing the system configurations for the 2000 topologies. These 20 hours plus support for new vendors (30 hours) plus the 50 hours to create the DSL itself leads to 100 hours of investment. ROI = (gain from investment – cost of investment) / cost of investment = (16000 - 100) / 100 = 159. Hence, the DSL has a high ROI which indicates that the investment in the DSL will be preferred above keeping the current way-of-working.
- *What are the pros and cons of using a DSL compared to the current way of working?*
  We list a number of advantages and disadvantages of using a DSL compared to the current way of working. We start with the advantages, in addition to the large ROI computed in the previous point:
  - Our DSLs are simple and easy to use; the users of the commercial tool, which are software engineers, should be able to create a new network topology and a new system configuration in a short amount of time.
  - Creating a network topology can be done in less time than with the current way of working, i.e., using the commercial tool.
  - The validation rules check if a network topology is valid, while with the commercial tool faults can be introduced that can only be found when a topology is build and tested.

  Below a list of disadvantages:
  - The generators of the DSLs contain additional code that needs to be archived, supported and maintained.
  - C++ is the programming language that is used at Philips. The generators of the DSLs can be programmed in Xtend and/or Java. The switch in programming language will create a barrier for some software engineers although the generator only needs to be supported by a few software engineers. There will be more users for the language than there are software engineers that need to maintain the language.
  - The preferred Integrated Development Environment (IDE) at Philips is Microsoft Visual Studio (MSVS). We have investigated and compared multiple solutions to create DSLs using MSVS, but the outcome of the investigation is that we can only use Eclipse for our needs. Installing a second IDE and switching between IDEs is a disadvantage.

At Philips, we clearly have a maintenance challenge when 2000 network topologies need to be supported. In this case, the DSL approach has a large ROI and, despite a few draw-

backs, provides a very good solution for this future maintenance problem. In general, due to the challenges with maintaining legacy components and the experiences presented in this paper, Philips will continue with the DSL approach.

Acknowledgments

We would like to thank the anonymous reviewers for a number of very useful remarks.

References

H. Vliet, Software Engineering: Principles and Practice, John Wiley & Sons, 2008.

V. Vyatkin, Software Engineering in Industrial Automation: State-of-the-Art Review, IEEE Transactions on Industrial Informatics, Vol. 9, No. 3, 2013, pp. 1234-1249.

R. Froschauer, D. Dhungana, and P. Grunbacher, Managing the lifecycle of industrial automation systems with product line variability models, in Proc. 34th Euromicro Conf. Software Eng. Adv. Applic., 2008, pp. 35-42.

H. Pei Breivold, I. Crnkovic and M. Larsson, A systematic review of software architecture evolution research, Information and Software Technology, Elsevier, 2011, pp 16-40.

K. Bennett and V. Rajlich, Software Maintenance and Evolution: A Roadmap, in Proc. Conf. The Future of Software Engineering, 2000, pp. 73-87.

L. Bettini, Implementing Domain-Specific Languages with Xtext and Xtend, Packt Publishing, 2013.

D. Geer, Eclipse Becomes the Dominant Java IDE, IEEE Computer, Vol. 38, No. 7, 2005, pp. 16-18.

J. Coplien, D. Hoffman and D. Wiess, Commonality and Variability in Software Engineering, IEEE Computer, Vol. 15, No. 6, 1998, pp. 37-45.

R. Sarnath and D. Brahma, Object-Oriented Analysis and Design, Springer, 2010.

A. van Deursen, Domain-Specific Languages versus Object-Oriented Frameworks: A Financial Engineering Case Study, in Proc. Smalltalk and Java in Industry and Academia, 1997.

D. Batory, C. Johnson, B. Macdonald and D. von Heeder, Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study, ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 2, April 2002, pp. 191-214.

S. Fehrenbach, R. Paige and E. van Wyk, Software Evolution to Domain-Specific Languages, in Proc. 6th International Conference on Software Language Engineering, LNCS, Vol. 8225, 2013, pp. 96-116.

P. Wolfgang, Design Patterns for Object-Oriented Software Development, Addison-Wesley Publishing Co., 1994.

M. Ward, Language Oriented Programming, Software - Concepts and Tools, 1994.

S. Sobernig, M. Strembeck and A. Beck, Developing a Domain-Specific Language for Scheduling in the European Energy Sector, in Proc. Conference on Software Language Engineering, LNCS, Vol. 8225, 2013, pp. 19-35.

S. Erdweg, S. Fehrenbach and K. Ostermann, Evolution of Software Systems with Extensible Languages and DSLs, IEEE Computer, Vol. 31, No. 5, 2014, pp. 68-75.

B. Basten, J. van den Bos, M. Hills, P. Klint, A. Lankamp, B. Lisser, A. van der Ploeg, T. van der Storm and J. Vinju, Modular language implementation in Rascal - experience report, in Journal Science of Computer Programming, Vol. 114, Elsevier, 2015, pp. 7-19.

A. Bergmayr, H. Bruneliere, J. Cabot, J. Garcia, T. Mayerhofer and M. Wimmer, fREX: fUML-based reverse engineering of executable behavior for software dynamic analysis, in Proc. of the 8th International Workshop on Modeling in Software Engineering, ACM, 2016, pp. 20-26.

A. Van Deursen and P. Klint, Little languages: little maintenance?, Journal of software maintenance, Vol. 10, No. 2, 1998, pp. 75-92.

J.-P. Tolvanen and S. Kelly, Defining domain-specific modeling languages to automate product derivation: Collected experiences, International Conference on Software Product Lines, Springer, 2005, pp. 198-209.

L. Cao, B, Ramesh and M. Rossi, Are domain-specific models easier to maintain than UML models?, IEEE software, 2009, Vol. 26, No. 4, pp. 19-21.

T. Berger, R. Rublack, D. Nair, J. Atlee, M. Becker, K. Czarnecki and A. Wasowski, A survey of variability modeling in industrial practice, Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, ACM, 2013, pp. 7:1-7:8.