In: Software Engineering and Development
Editor: Enrique A. Belini, pp. 25-35
ISBN 978-1-60692-146-3

Short Communication A

EMBEDDING DOMAIN-SPECIFIC LANGUAGES IN GENERAL-PURPOSE PROGRAMMING LANGUAGES

Zoltán Ádám Mann
AAM Consulting Ltd.; Budapest University of Technology and Economics

ABSTRACT

In recent years, domain-specific languages have been proposed for modelling applications on a high level of abstraction. Although the usage of domain-specific languages offers clear advantages, their design is a highly complex task. Moreover, developing a compiler or interpreter for these languages that can fulfil the requirements of industrial application is hard. Existing tools for the generation of compilers or interpreters for domain-specific languages are still in an early stage and not yet appropriate for the usage in an industrial setting.

This paper presents a pragmatic way for designing and using domain-specific languages. In this approach, the domain-specific language is defined on the basis of a general-purpose programming language. Thus, general programming mechanisms such as arithmetics, string manipulations, basic data structures etc. are automatically available in the domain-specific language. Additionally, the designer of the domain-specific language can define further domain-specific constructs, both data types and operations. These are defined without breaching the syntax of the underlying general-purpose language. Finally, a library has to be created which provides the implementation of the necessary domain-specific data types and operations. This way, there is no need to create a compiler for the new language, because a program written in the domain-specific language can be compiled directly with a compiler for the underlying general-purpose programming language. Therefore, this approach leverages the advantages of domain-specific languages while minimizing the effort necessary for the design and implementation of such a language.

The practical applicability of this methodology is demonstrated on a case study, in which test cases for testing electronic control units are developed. The test cases are written in a new domain-specific language, which in turn is defined on the basis of Java. The pros and cons of the presented approach are examined in detail on the basis of this case study. In particular, it is shown how the presented methodology automatically leads to a clean software architecture.

1. INTRODUCTION

In the last decades, the requirements toward software have become tougher and tougher. The complexity of the problems that are solved by software is growing, while at the same time the expectations concerning numerous other, non-functional, aspects (for instance, maintainability, usability, fault-tolerance, parallelism, throughput etc.) have also increased significantly. Moreover, in today's highly competitive software market, it is crucial to minimize time-to-market for software, to be able to quickly add fixes or new features to products.

Since the human brain has not evolved significantly in this time, the only way to create more complex software more quickly is to raise the level of abstraction for software development. Just imagine how it would be to develop software that should fulfil today's requirements, if you had to keep in mind which piece of data is in which register of the processor!

In order to cope with increasing complexity, the profession moved from machine code to assembler, from assembler to high-level programming languages, then to object orientation, to component orientation etc. Today, we think in terms of high-level programming abstractions, such as components, threads, GUI elements etc., and not in terms of what the hardware can offer (registers, memory addresses, interrupts).

Despite all this development, the requirements are still ahead of what we can deliver safely with our current software development practices. So, what will be the next quantum leap in increasing the level of abstraction?

Many researchers agree that the destination of this journey will be some kind of *model orientation* [6]. Software development will mean creating an abstract, logical model of *what* the software is supposed to do, without technical details on *how* it will fulfil those aims. As formulated by Brooks in his seminal paper "No silver bullet," the essence of software development is the construction of abstract, conceptual structures; the difficulties arising from the representation of these structures within the framework of a programming language are just accidental and are decreasing with scientific progress [1].

There are some debates in the research community on what the future model oriented software development process will look like[*]:

- One possibility is to define a universal modelling language that can be used for the development of any software application. Most notably, the Object Management Group (OMG) follows this path with the Unified Modelling Language (UML) [12]. In contrast, others argue that modelling at a really high level of abstraction is only possible with domain-specific concepts, which can be best accomplished by a domain-specific language (DSL). In recent years, this latter approach has gained tremendously in popularity [2] and is also the topic of this paper. More on DSLs can be found in Section 2.
- Another question is how to bridge the gap between the abstract model and the real features of the available platform. Two main approaches can be distinguished, similar to compiled vs. interpreted programming languages. The first approach

∗ Also, there are minor differences in the terminology, e.g. model-based vs. model-driven vs. model-oriented.

consists of *generating* (possibly in more than one step) program code from the model, after which the code can be executed using traditional mechanisms. For instance, the OMG's Model-Driven Architecture (MDA) paradigm falls into this category [11]. The other approach consists of *executing* the model itself with a suitable model interpreter. As an example, the Executable UML (xUML) approach belongs to this category [9].

- When hearing the word 'model,' one tends to think of a graphical representation, like an UML model. However, graphical modelling has its limitations. Not only is a graphical representation less appropriate for machine processing, but also for the human reader, it is quite hard to understand hundreds (or more) of pages of graphical models. Usually, a textual model is more concise and can therefore scale better in model size when readability is concerned. Thus, textual modelling languages became more popular in recent years [5].

In the rest of the paper, textual domain-specific languages are considered. The issue of generating code from the model vs. interpreting the model itself will be discussed in more detail.

1.1. PAPER ORGANIZATION

The rest of the paper is organized as follows. In Section 2, the concept of DSLs is described in more detail, with special emphasis on the challenges associated with the development of a DSL. Section 3 contains a case study, introducing the domain of testing electronic control units. In this domain, there is a need for a DSL for the specification of test cases. Section 4 describes the proposed pragmatic way of defining a DSL based on a general-purpose language in principle, followed by the second part of the case study in Section 5, in which the practical applicability of the proposed approach is presented for specifying test cases for electronic control units. Section 6 contains a discussion of the lessons learned in the application of the proposed methodology, while Section 7 concludes the paper.

2. DSLS

2.1. GENERAL PROPERTIES OF DSLS

A DSL is a language for the description of programs, or of models of programs*, on a specific field of application (called a *domain*). Since the language is tailored to one domain, complex constructs and abstractions of the domain can be supported directly by the language. A number of benefits are expected from this clear focus on one domain, such as:

- Concise representation of complex issues;
- Gain in productivity;

* From a theoretical point of view, the distinction between a program and a model of the program is artificial, since a model can be defined as an abstract representation of a system, and thus the program itself can also be regarded as a model.

- Improved maintainability;
- Better communication between IT and functional departments;
- Efficient development of variants and of software product lines.

The idea of domain-specific languages is not new. There are several languages that are already widely used and can be regarded as a DSL, for instance:

- SQL (Structured Query Language) for the definition of database queries and manipulations;
- XSLT (eXtensible Stylesheet Language Transformation) for the definition of transformations between XML (eXtensible Markup Language) files;
- `sed` scripts for string manipulations;
- `make` scripts for directing the software build process.

As can be seen from this list, these widely used DSLs are usually tailored to a *technical* domain. For *functional*[*] domains, the idea of DSLs can also be leveraged; however, by the nature of functional domains, these languages are usually known and used only by a limited set of experts. Examples include:

- CPL (Call Processing Language) for the definition of Internet telephony services;
- BPMN (Business Process Modeling Notation) for the definition of business processes;
- OWL (Web Ontology Language) for the definition of ontologies for the Semantic Web;
- VoiceXML for the definition of interactive voice dialogues between a human and a computer.

2.2. CREATING DSLS

Developing a DSL and the supporting tool chain is a time-consuming process requiring much care and deep expertise [10]. The process can be divided into five phases: decision, analysis, design, implementation, and deployment [5]. Out of these, especially challenging are the analysis phase and the implementation phase.

In the analysis phase, the constructs of the domain that should be integrated into the language have to be identified and formalized. Although there are several methodologies for this kind of *domain engineering*, this phase is definitely time-consuming and requires special expertise.

In the implementation phase, the necessary tool chain must be developed for the language: editor, compiler / interpreter, debugger, profiler etc. Parts of this can be automated (e.g., parser generation), and there are also *language workbenches* [3] for facilitating the

* In this context, the distinction between technical and functional is as follows. Functional issues are those intrinsic properties of the system which result directly from the end user's functional requirements. In contrast, technical issues are related to the implementation of the system with a specific technology. Accordingly, a technical domain is relevant for the IT expert, whereas a functional domain may also be relevant to the end user.

whole process (e.g., Eclipse Modeling Framework, Microsoft Visual Studio DSL Tools). However, not all the steps can be fully automated, so that creating efficient tools for a non-trivial DSL remains a difficult process with a lot of manual work. In particular, developing a compiler or interpreter for the language that can fulfil the requirements of industrial application is hard.

3. CASE STUDY – PART 1

As a case study for a domain-specific language, the domain of testing electronic control units (ECUs) in vehicles is considered.

An ECU is an embedded computer system with a specific control function within a bigger mechatronic system. For instance, a high-end car contains nowadays up to 80 ECUs (e.g., ABS, tuner, night vision camera control, airbag control etc.). The ECUs within a car are interconnected so that they can exchange messages. For the interconnection of ECUs several bus technologies are in use, from which two are the most common ones: Controller Area Network (CAN) and Media Oriented System Transport (MOST). CAN supports the transmission of 8-byte messages with a data rate of up to 500 kilobit/sec and a non-negligible error rate. MOST is a more expensive technology, supporting the safe transmission of messages of up to 4 kilobyte in length and a data rate of up to 23 megabit/sec. Moreover, the two technologies differ significantly in their addressing scheme.

Car manufacturers spend huge amounts of resources with testing whether every ECU obeys its specification in every possible combination and under all imaginable circumstances. Testing ECUs has several flavours and there are several methodologies. In this paper, we will focus on the testing of generic *system functions* (e.g. power management, security, personalization) that have to be implemented in every ECU according to the same logical specification, but with different technical details, e.g. depending on the bus technology used by the ECU (CAN/MOST) [4]. It should also be noted that, in order to find errors in the earliest possible stage, these generic system functions are usually first implemented and tested in the form of a software simulation on a PC.

The testing of these functions basically consists of sending different sequences of messages to them and comparing the replies from the ECU with the expected behaviour (whether there was a reply at all; whether timing requirements were met; whether the data in the reply were as expected etc.).

Now the challenge is the following. The test cases for testing (a) the PC simulation of the function; (b) the implementation of the function in a CAN ECU; (c) the implementation of the function in a MOST ECU are almost the same at the logical level. However, at the level of the communication technology, the three cases are quite different. The aim is to define the test cases only once, at a sufficiently high level of abstraction, and use them in all three cases (see Figure 1).

Thus, the goal is to define a DSL with the following main concepts:

- Sending of messages with defined content to the System Under Test (SUT);
- Waiting for messages from the SUT with given timing constraints;
- Comparing the contents of a received message with a predefined pattern.

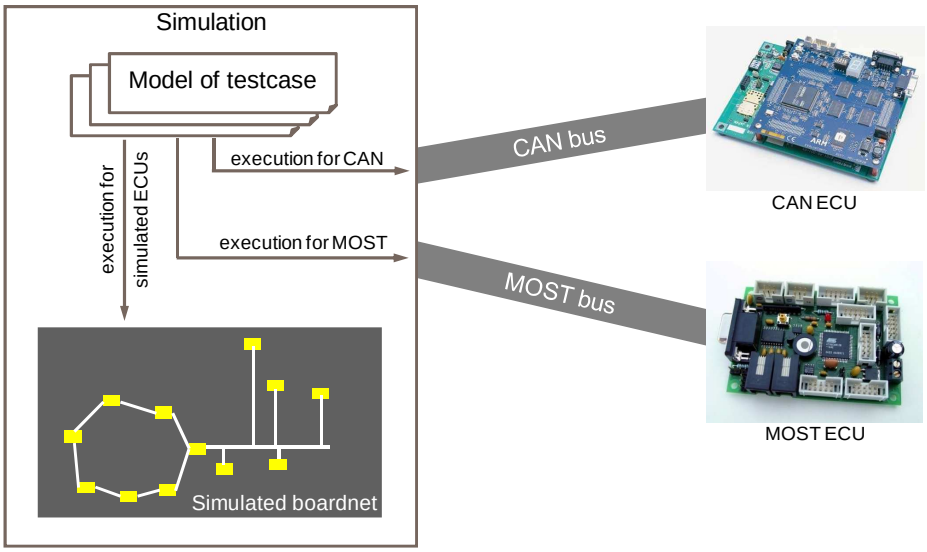

Figure 1. The same logical test cases should be executed for different ECU implementations

The DSL should be free of any references to the specific communication technology; however, it should be possible to run the test cases without modification on any of the supported technology platforms.

The resulting DSL is described in Section 5.

4. A PRAGMATIC APPROACH TO DSL DEVELOPMENT

In light of the challenges associated with the development of a DSL (see Section 2), we suggest that DSLs should be developed from scratch only if (a) there are some specific requirements concerning the tool chain that are otherwise hard to fulfil (e.g., hard requirements concerning performance may require a very specific optimizing compiler) and (b) the foreseen wide-spread usage of the DSL justifies the efforts. Otherwise, we propose using a pragmatic approach in order to leverage the benefits of DSLs even in projects with very limited budget, as follows.

The DSL should be defined on the basis of an existing general-purpose programming language (GPL). Thus, general mechanisms such as arithmetics, string manipulations, basic data structures etc. are automatically available in the DSL. Additionally, the designer of the DSL will of course define further, domain-specific, constructs. These can be categorized as data types and operations. Both can be defined without breaching the syntax of the underlying GPL, as data types and operations in the GPL. Finally, a library has to be created which provides implementation in the GPL for the defined domain-specific data types and operations. In other words: the DSL is nothing but a GPL enriched with domain-specific data types and operations, which are defined in the GPL themselves. A program written in the DSL is thus at the same time also a program in the GPL.

The representation of the DSL within the GPL is possible, because there are so many degrees of freedom in the design of the DSL. Usually, the requirements concerning the future DSL are very high-level: what kinds of domain constructs should be available in the language

and what kinds of operations should be possible on these constructs (see for instance the requirements formulated in Section 3 in connection with the DSL for ECU test case specification). There are usually no strict constraints on the syntax of the language, so any logical, readable, and coherent syntax can be used. Thus, the syntax of a GPL is usually applicable.

This approach has several major advantages. First of all, since a program in the DSL is at the same time also a program in the GPL, the whole tool chain of the GPL can be used for programs written in the DSL. This way, the efforts involved in the creation of the DSL are drastically reduced. Moreover, it is safe to assume that the tool chain of a GPL is significantly more mature – concerning comprehensiveness, correctness, documentation, etc. – than the tools that would be created for the sake of the DSL. Furthermore, many useful features of the DSL can simply be inherited from the GPL "free of charge," such as macros, inheritance, etc. – features that you might not bother to include in the language if developed from scratch.

Of course, this approach also has some limitations. If, for some reason, there are very specific requirements concerning the syntax of the DSL that cannot be found in any available GPL, then this approach cannot be applied. Also, this approach does not yield a clear separation between code and model, which can be a problem if some team members are supposed to work on the model only.

In any case, since the presented approach allows for the quick and easy construction of DSLs, it can be used as a rapid prototyping methodology. Suppose for instance that for a given domain DSL1 is created using the above methodology. The language can be tried in practice and fine-tuned based on the experience in an early stage of the design. Afterwards, a second language DSL2 can be created which is semantically equivalent to DSL1, but its syntax is closer to the logical syntax of the domain instead of the syntax of the GPL. Then, only a DSL2-to-DSL1 compiler must be created, in order to have a full-fledged DSL (namely, DSL2) with moderate efforts.

5. CASE STUDY – PART 2

We have applied the presented approach to the ECU test case specification domain presented in Section 3. That is, we developed a DSL for the black-box testing of ECUs, based on Java as the underlying GPL.

5.1. DESIGN OF THE DSL

The building blocks of test cases consist of the sending of messages to the ECU and the waiting for the reply from the ECU. The most common scenario is that the test case sends a message to the ECU and expects a reply in a given time frame. Such a building block can have a variety of parameters, such as:

- The message that should be sent to the ECU;
- Criteria for the acceptance of the reply (e.g. bit 12 must be 0 in the reply);
- Minimum and maximum reply time.

There are other, similar building blocks. For instance, it is possible to specify that, after having sent a given message to the ECU, no reply must arrive within a given time frame. From such building blocks, complex test cases can be compiled, as shown in Figure 2.

```
//Inquiry + 1st reply
messageToSend=new MessageInquiry(TEST_ID);
idOfExpectedReply=TEST_ID;
minimalWaitTime=0;
maximalWaitTime=100;
sendMessageAndWaitForReply();

//2nd reply
minimalWaitTime=175;
maximalWaitTime=225;
waitForReply();

//3rd reply
waitForReply();

//wait one more period; no further reply should arrive
waitNoReply();
```

Figure 2. Example test case.

This test case tests that an inquiry sent to the ECU results in exactly 3 replies, from which the first one must arrive within 100 milliseconds after sending the inquiry and the next two with a gap of approximately 200 milliseconds between them.

The grammar of the DSL is specified in EBNF notation in Figure 3.

```
<test case> := <first-block> <block>*
<first-block> := <param-spec> <command>
<block> := <param-spec-opt> <command>
<param-spec> := <inquiry-spec>
                <reply-spec>+
                <min-time-spec>
                <max-time-spec>
<param-spec-opt> := <inquiry-spec>?
                    <reply-spec>*
                    <min-time-spec>?
                    <max-time-spec>?
<inquiry-spec> := "messageToSend=new MessageInquiry("
                  <inquiry-params> ");"
<inquiry-params> := <id> | …
<reply-spec> := "idOfExpectedReply=" <id> ";" | …
<min-time-spec> := "minimalWaitTime=" <number> ";"
<max-time-spec> := "maximalWaitTime=" <number> ";"
      <command> := "sendMessageAndWaitForReply();"
           | "sendMessageNoReply();"
           | "waitForReply();"
           | "waitNoReply();"
```

Figure 3. Grammar of the DSL for ECU testing

5.2. IMPLEMENTATION OF THE DSL

The domain-specific constructs of the language – the attributes `minimalWaitTime`, `messageToSend` etc. as well as the operations `waitForReply()` etc. – are specified in an abstract Java class called `ECUTest`. All test cases are Java classes that inherit from this abstract class, so that these attributes and operations can be used in all the test cases (see Figure 4).

Of course, the exact behaviour of these operations depends on the used technology (calling Java routines vs. using CAN messages vs. using MOST messages). Hence, the operations in the class `ECUTest` do nothing but delegate the work to an adapter. The class `ClientTest` can be parameterized with different adapters according to the used technology. All details concerning the technology are encapsulated in the relevant adapter. The test cases themselves are free of any technology-related details.

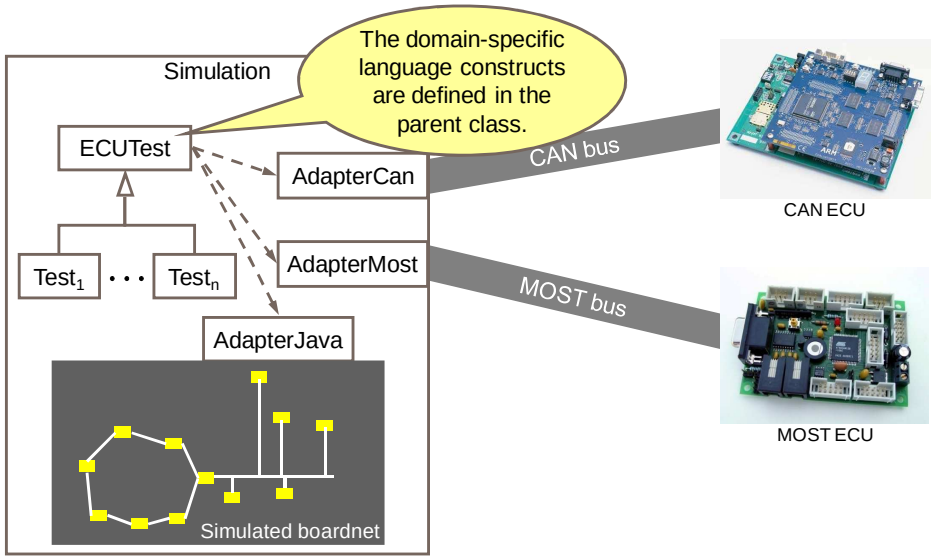

Figure 4. Embedding the DSL into Java via inheritance

This way, the challenge described in Section 3 is met: the test cases are specified at a logical level, only once, but can be used without any modifications with the different technologies.

6. DISCUSSION

Based on the presented case study, two issues are discussed:

- The consequences of using an underlying GPL on the DSL;
- The resulting software architecture.

6.1. CONSEQUENCES ON THE LANGUAGE

As can be seen from Figure 2 and Figure 3, the syntax of the language is sufficiently simple, and contains only constructs of the given domain. Thus, expressing test cases in the DSL is really simple and easy to understand. In particular, it is much simpler than its implementation in Java, in particular because the latter involves at least two threads: one for receiving the asynchronous incoming messages, the other for checking the elapsed time and interrupting the first one after the specified amount of time. The chosen syntax elegantly hides this complexity from the user, who can thus focus on the logic of the test case instead of the difficulties of multithreaded Java programming.

It should also be noted that, although embedding the DSL into Java imposes some constraints on the syntax of the DSL (e.g., every command must be followed by a semicolon), these restrictions are not disturbing at all.

What is more, the embedding in Java provides a lot of powerful features free of charge. For instance, comments can be added to the test cases according to Java syntax, although this was not explicitly defined in the language grammar. More importantly, when judging the acceptability of the data contained in an incoming message, the full power of Java can be used to perform sophisticated computations (e.g. take a substring of the data field, interpret it as a hexadecimal number, compute a formula based on this number etc.). Defining these features from scratch, without relying on the existing features of Java, would be a quite tedious and time-consuming task.

6.2. THE RESULTING ARCHITECTURE

When looking only at the result in Figure 4, one could argue that this is a pure Java system, without any use of a DSL. In a way, this is indeed the case: through the embedding in Java, at the end all artefacts are in Java, and the DSL is not visible at all.

However, when assessing Figure 4 thoroughly, one can also state that the result is a really clean architecture in which technology-related code and functional (i.e., test-related) code is successfully separated, in the sense of *separation of concerns*. It should also be noted that this feature is guaranteed automatically by the usage of the DSL, since the DSL only contains constructs of the domain, and no technology-related issues. Therefore we can conclude that even if the DSL is not visible in the final product, its use is justified also by the final product because the consequent use of the DSL leads automatically to the presented clean software architecture.

Moreover, as mentioned in Section 4, the option is still available to extract the test cases from the Java program into separate non-Java files, and transform them in an automated way to Java or interpret them on-the-fly, if a looser coupling is needed.

7. CONCLUSION

This paper has presented a pragmatic approach for the development of DSLs, in which the DSL is not created from scratch, but rather on top of an existing GPL. The GPL is extended with domain-specific constructs that are defined and implemented as data structures

and operations in the GPL. This way, the tools for the GPL can be used directly also in connection with the DSL, which drastically reduces the efforts of implementing the DSL.

The presented case study showed how this approach can be applied in practice. A DSL has been devised for the black-box testing of ECUs on the basis of Java. With the presented approach, it is possible to specify the test cases at a high level of abstraction, without making any reference to the underlying technology (whether CAN, MOST, or direct Java function calls). The resulting DSL is simple and easy to use; moreover, its use leads automatically to a clean software architecture.

To sum up: the presented approach helps to leverage the power of DSLs even in small projects in which creating a full-fledged DSL from scratch would not be feasible.

8. REFERENCES

[1] Brooks, F. P., Jr.: No Silver Bullet – Essence and Accidents of Software Engineering, *Computer*, 1987

[2] Cook, S.; Jones, G.; Kent, S.; Wills, A. C.: Domain-Specific Development with Visual Studio DSL Tools, *Addison-Wesley*, 2007

[3] Fowler, M.: Language workbenches – the killer app for domain specific languages?, http://www.martinfowler.com/articles/languageWorkbench.html

[4] Heider, A.; Mann, Z. Á.; Staudacher, B.: Verteiltes System, *Automobil Elektronik*, 03/2006

[5] Karlsch, M.: A model-driven framework for domain specific languages, *Master's thesis, Hasso-Plattner-Institute of Software Systems Engineering*, 2007

[6] Kempa, M.; Mann, Z. Á.: Aktuelles Schlagwort: Model Driven Architecture, *Informatik Spektrum*, August 2005

[7] Ludwig, F.; Salger, F.: Werkzeuge zur domänenspezifischen Modellierung, *OBJEKTspektrum*, 03/2006

[8] Luoma, J.; Kelly, S.; Tolvanen, J.: Defining Domain-Specific Modeling Languages – Collected Experiences, *Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling*, 2004

[9] Mellor, S; Balcer, M: Executable UML – A foundation for model-driven architecture, *Addison-Wesley*, 2002

[10] Mernik, M; Heering, J.; Sloane, A. M.: When and how to develop domain-specific languages, *ACM Computing Surveys*, volume 34, issue 4, pages 316-344, 2005

[11] Object Management Group: Model Driven Architecture, http://www.omg.org/mda/

[12] Object Management Group: Unified Modeling Language, http://www.uml.org/