

Frameworks and Static-Semantic Analysis

Martin Fagereng Johansen, Birger Møller-Pedersen

Department of Informatics
University of Oslo
Post box 1080 Blindern 0316 Oslo, Norway

`martifag@ifi.uio.no`

Abstract

A domain may be represented either by a Domain Specific Language (DSL) or by a framework. While programs made in a DSL may be subject to static-semantic analysis and thereby ensured to be in accordance with the constraints of the domain, programs made on the basis of a corresponding framework cannot be analyzed with the same simplicity. Constraints of a domain are identified during domain analysis, and they are independent of whether a DSL or a framework is constructed. The reason that static-semantic analysis is harder for frameworks is that frameworks are implemented in general purpose implementation languages. The paper illustrates why Java is unsuited to support some static-semantic analysis for frameworks in Java, and it identifies which features a framework implementation language should have in order to support static-semantic analysis: An individual static constraint has an associated canonical form; therefore the implementation language must be able to implement this canonical form accurately, while the framework remains easy to use, in order to support static-semantic analysis.

1 Introduction

Domain specific languages (DSLs) and frameworks are different in that DSLs commonly support static-semantic analysis, while frameworks commonly do not. This paper investigates why this is the case.

Static semantic analysis is clearly of interest to both developers and users of a framework. Many frameworks have a large user base. For these users to be able to run a check on what they have written, potentially saves them time, and from the risk of encountering some faults at run-time. Static semantic analysis is not normally supported by frameworks today, even though static-semantic analysis can tell if certain properties of a program is wrong or dubious and give constructive feedback at compile-time.

If it is important with static-semantic analysis for a general solution, an application that is used as a basis for other programs, then it is important to know if the technology

This paper was presented at the NIK-2009 conference; see <http://www.nik.no/>.

used to implement it will support such analysis. The results of this paper can be taken into consideration when choosing a technology with which to implement a general solution.

This paper is structured as follows. In section 2 we have a look at background information and at an experiment. This experiment is used as the basis for the examples used in later sections. In section 3 a representation that can be used to model the requirements of a static constraint is presented. This representation is used to reason about why static semantic analysis is harder for frameworks than for DSLs. The paper ends with a look at related and future work.

2 Experiment

Domain Models

A domain model is the artifact created by domain analysis. Domain analysis gives insight into how a systems should be designed, which concepts it should include and how they should relate. The domain analysis used for the experiment in this paper is not done according to any specific method. It is simply a systematic look at a domain, which then results in a semi-formal representation of the domain. The domain model has a formal part consisting of an ECore model¹ and an informal part consisting of English text describing the things that is not easy to express with ECore.

The domain used as an example in this paper is a part of the e-commerce domain. The e-commerce domain contains the notion of an electronic-catalog. An electronic-catalog is an application that allows an e-commerce solution administrator to set up his or her offers in a structured way. A customer can browse the electronic-catalog through, for example, a web site.

The domain model for an electronic catalog was created by domain analysis. The analysis was performed by studying several other e-commerce solutions in addition to the author's experience on working with a production e-commerce solution. The study resulted in a list of 60 distinct requirements. A domain model was then created in order to fulfill these requirements.

Figure 1 shows an excerpt from the domain model. The catalog class is the class which contains all the commodities offered for sale and their structure². An instance of "Catalog" is a concrete catalog. An instance of "Commodity", however, is a commodity type which instances are concrete commodities. For example, the instance of "Commodity" can be "car", and an instance of "car" is a particular "car". The reason for this is that it is impossible to say which attributes and associations any given commodity will require; therefore the programmer programs the individual types first, for example cars, and then proceeds to add instances of this type according to which cars he or she has in stock.

An instance of "Commodity" can have a super-type. For example, the commodity "iPod Nano" has the super type "mp3-player", because all "iPod Nanos" are "mp3-players".

Commodities can be members of categories, which may in turn be member of other categories.

A commodity can have commodity parts, though only if these are direct instances of "Commodity". This is not specified in the diagram; only in a constraint. "GeneralCom-

¹Eclipse's version of the EMOF standard, see [Ecl08] for an overview.

²The relationships showing that the catalog contains instances of "Commodity", "Category" and "Interface" is not shown in the figure.

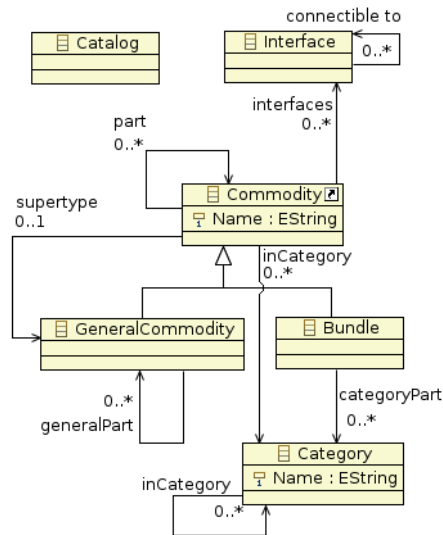


Figure 1: Excerpt from the Catalog Construction Domain Model

modity" and "Bundle" can have general parts and category parts, respectively. A bundle is a collection of generally unassociated commodities that are packed together for sale. A part of the bundle can be the option of freely choosing between members of a category. Such a part is called a "category part". Examples of this are "Freebies" and "Commodities on sale".

Finally, the two classes "Interface" and "Category" do not need concrete instances. Once declared they can be associated to the commodity types.

The domain model just presented forms the basis for an implementation of both a DSL and a framework: The DSL is called the catalog-construction language, and the framework is called the catalog-construction framework.

Domain Specific Languages

The definition of DSL used in this paper is from [Spi01]: "A domain-specific language is a programming language tailored specifically to an application domain: rather than being general purpose it captures precisely the domain's semantics." As with all programming languages, a DSL consist of a custom syntax and semantics. The syntax specifies how concepts of a language can be combined. The semantics of a DSL describes the meaning of terms and sentences within the DSL.

The catalog-construction language is a DSL that lets an e-commerce administrator set up and edit an electronic catalog. The administrator writes DSL code using a text editor. The code is then compiled into an electronic-catalog application with a web-view and an administrator interface. The web-view allows the customers to browse the catalog in many different ways according to the structure of the catalog contents. The application allows the administrator to edit the different parts of the source code at run-time. The administrator loads the source code of a part of the catalog into a text editor, edits it and saves the changes. If an electronic catalog is stored in a database system, for example, then the database schema can be changed by the electronic catalog application when the source code is changed.

Listing 1³ shows an example catalog application programmed with the catalog

³This example assumes there is a standard charger.

construction language. First of all, the "Catalog" keyword followed by the catalog name is used to declare a catalog. Everything that follows is a part of the catalog. There can only be one catalog per source file. The keyword "GeneralCommodity" is used to declare a new general commodity. An example of this is "Mp3Player". Inside, four parts are declared: the mp3-unit, which is the player itself, the headphones and the charger. Each of these are declared by typing the identifier name, a colon and the type. The three general commodities used as types for the parts of the mp3-player are declared next. In these declarations, we have a list of interfaces but no parts. Interfaces are declared with the "Interface" keyword, and commodities are declared with the "Commodity" keyword.

The commodity "iPodNano" is a sub-type of "Mp3Player" as expected. In it, the types of the three components in "Mp3Player" are refined.

Listing 1: Example Catalog Application Code Excerpt in the DSL

```
Catalog ElectronicsShopCatalog
Interface PhonoPlug
...
GeneralCommodity Mp3Player
  mp3Unit : Mp3Unit
  headphones : HeadPhone
  charger : Charger
GeneralCommodity Mp3Unit
  interfaces : ChargerPlug, PhonoPlug
GeneralCommodity HeadPhone
  interfaces : PhonoPlug
GeneralCommodity Charger
  interfaces : ChargerPlug
...
Commodity IPodNano
  supertype : Mp3Player
  mp3Unit : IPodNanoUnit
  headphones : WhiteAppleEarplugs
  charger : IPodCharger
Commodity IPodNanoUnit
  supertype Mp3Unit
...
```

Frameworks

Frameworks are considered in this paper as explained in [Bos98].

[...] an object-oriented framework is a kind of reusable software architecture comprising both design and code. [JF88] defines a framework [as] *a set of classes that embodies an abstract design for solutions to a family of related problems*. In other words, a framework is a partial design and implementation for an application in a given problem domain. The central part of the framework design comprises both abstract and concrete classes in the domain.

The notions of frozen and hot spots in a framework are important. This is explained in [BMR⁺96]:

According to [Pre94] an application framework consists of frozen spots and hot spots. Frozen spots define the overall architecture of a system, its basic components and the relations between them. These remain unchanged in any instantiation of the application framework. Hot spots represent those parts of the framework that are specific to individual systems.

In this paper the frozen spots are seen as the similar to the run-time system of a DSL, and the hot spots are seen as similar to DSL source code. For example, when one uses a DSL one writes DSL code. This code is run with the DSL run-time system. Similarly, when one uses a framework one programs code for the hot spots. This code is then run with the frozen spots of the framework.

The framework functions similarly to the DSL. However, for using the framework, one writes Java code instead. The Java code initializes the framework and sets up the initial catalog contents.

In order to alter the catalog at run-time, the implementation language must support adding classes at run-time. This feature is supported by Java. The administrator can write a new class or change an existing one. These classes are then loaded into the application which handles them so that updates to existing classes are handled properly. New classes can also be added.

Listing 2 shows the same example as shown for the DSL in listing 1. For the framework the catalog is declared by overriding the framework catalog class. The catalog contents are declared in a sequential fashion by creating objects and filling them with information. The finished objects are not, as in the DSL, automatically added to the catalog. Therefore, they are added manually.

Listing 2: Example Code Excerpt that Uses the Framework

```
public class ElectronicsShopCatalog extends CatalogImpl {
    public ElectronicsShopCatalog(){
        addInterface(new InterfaceImpl("PhonoPlug"));
        ...
        GeneralCommodity mp3Unit = new GeneralCommodityImpl("Mp3Unit");
        mp3Unit.addInterface("PhonoPlug");
        mp3Unit.addInterface("ChargerPlug");
        addGeneralCommodity(mp3Unit);
        GeneralCommodity headPhone = new GeneralCommodityImpl("
            HeadPhone");
        headPhone.addInterface("PhonoPlug");
        addGeneralCommodity(headPhone);
        GeneralCommodity charger = new GeneralCommodityImpl("Charger");
        charger.addInterface("ChargerPlug");
        addGeneralCommodity(charger);
        GeneralCommodity mp3player = new GeneralCommodityImpl("
            Mp3Player");
        mp3player.addPart("mp3Unit", "Mp3Unit");
        mp3player.addPart("headphones", "HeadPhone");
        mp3player.addPart("charger", "Charger");
        addGeneralCommodity(mp3player);
        Commodity iPodNanoUnit = new GeneralCommodityImpl("IPodNanoUnit
            ", "Mp3Unit");
        addCommodity(iPodNanoUnit);
        ...
        Commodity iPodNano = new GeneralCommodityImpl("IPodNano", "
            Mp3Player");
        iPodNano.addPart("mp3Unit", "IPodNanoUnit");
        iPodNano.addPart("headphones", "WhiteAppleEarplugs");
        iPodNano.addPart("charger", "IPodCharger");
        addCommodity(iPodNano);
        ...
    }
    ...
}
```

Static-Semantic Analysis

Semantics and static-semantic analysis are described in [WG84, p. 12 and p. 183] as follows.

Semantics include properties that can be deduced without executing the program, as well as those only recognizable during execution. Following [Gri75], we denote these properties static and dynamic semantics, respectively.

Semantic analysis determines the properties of a program that are classed as static semantics, and verified the corresponding context conditions the consistency of these properties.

Static-semantic analysis is the compilation phase that commonly follows lexical analysis and parsing. According to [WG84] the semantic analyzer typically performs "name analysis, finding the definition valid at each use of an identifier. Based upon this information operator identification and type checking determine the operand types and verify that they are allowable for the given operator."

For our domain model we have selected three example static constraints. The enumerations later in this paper refer to these static constraints.

1. *The parts of a general commodity must all be connectible.* The purpose of a general commodity is partly to allow users to select the individual parts of the commodity, while still ensuring that the commodity as a whole still functions. For example, an mp3-player is a general commodity. The reason is that you cannot buy an mp3-player as such. What you can buy is, for example, an iPod. An iPod is an instance of an mp3-player.
2. *The structure of the catalog must be right.* Concrete commodities cannot have general parts. If they do then they are really general commodities and should be declared as such.
3. *Cycles are not allowed in the composition nor in the super-type or the catalog membership structure.* It is meaningless with cycles in these structures, and they can therefore be reported as mistakes.

3 Canonical Forms for Static-Semantic Analysis

Canonical Forms

In order to reason about why DSLs support static semantic analysis and why frameworks do not, a thought experiment is considered. It starts with a framework. A part of the framework is converted into a part of a DSL. This new DSL is then compared to a corresponding part of the original DSL.

Imagine an arbitrary catalog application written using the catalog construction framework. Take a particular constraint that is possible to check for the framework, and which has a corresponding constraint in the DSL. We start by copying the program and then remove the parts of the program that are not relevant for the evaluation of the constraint. This will eventually leave us with only those parts that are required to check the constraint. It is important to remember that we cannot remove constructs that affects the result of the static check. Examples of this is if- and loop-constructs that depends

on run-time results. When we have removed everything that can be removed, we can make an abstract representation for the remaining parts of the program. Instances of this representation are all the possible inputs to the constraint's check. These also respects that there can exist unresolved constructs from the framework implementation language. The representation can be used as a basis for a new DSL. The same constraint can now be checked equally well for the framework and for the new DSL. The difference between the generated DSL and the original DSL can be seen through the difference between the representations of the required input of the two corresponding static constraints mentioned initially in the example.

The representation of the input to a static constraint is called the constraint's *canonical form*. This data structure is *implementation independent* as it can be determined with reference to the domain only. The static constraints presented in this paper are no exception. The following are the canonical forms for the example constraints. (Note that the enumeration refers to the previous listing of static constraints.)

1. If we interpret parts as nodes and two connectible interfaces as an edge then we get a collection of graphs. If there is only one graph in this collection then we know that all the parts are connected into one unit.
2. We interpret the catalog content types as nodes, and the composition relations as directed edges. The nodes are given an attribute that corresponds to the catalog content type. The pairs of nodes with a directed edge between them are compared to a collection of valid pairs.
3. We set up three graphs. These graphs have catalog content types as nodes. One graph has directed edges as composition relations, one has directed edges for the sub-type relation and the last has catalog membership as edges. All these graphs must be directed acyclic graphs (DAGs) in which case there are no cycles.

DSLs and Canonical Forms

Listing 3 shows the grammar for the catalog construction language. It will, with reference to this grammar, be shown how it is known that the canonical forms of the constraints mentioned earlier are extractable from any program that is in accordance with this grammar.

In the beginning of the grammar we see the production of a catalog. The catalog can contain several instances of five different things: commodities, general commodities, bundles, categories and interfaces.

Listing 3: Grammar of the Catalog Construction Language

```
catalog: 'Catalog' Identifier catalogcontent*;
catalogcontent: commodity | category | interfaceDef;
commodity: concretenesscommodity | generalcommodity | bundle;
concretenesscommodity: 'Commodity' Identifier supertype? incategories?
    interfaces? content*;
category: 'Category' Identifier incategories? attribute*;
generalcommodity: 'GeneralCommodity' Identifier supertype?
    incategories? interfaces? content*;
bundle: 'Bundle' Identifier incategories? content*;
interfaceDef: 'Interface' Identifier connectible? attribute*;
connectible: 'connectible with' ':' Identifier (',' Identifier)*;
content: part | attribute;
```

```

incategories: 'in categories' ':' Identifier (',' Identifier)*;
interfaces: 'interfaces' ':' Identifier (',' Identifier)*;
supertype: 'supertype' ':' Identifier;
attribute: Identifier ':' atomictype ('=' Value)?;
atomictype: 'Integer' | 'String' | 'Date' | 'Float' | 'Double';
part: Identifier ':' Identifier;

```

The following explains how each example canonical form is ensured by the grammar.

1. If we look at the concrete and the general commodity productions, we see that they both have the "interfaces" element. These lists of interfaces cannot be affected by any other language construct. Therefore, we can know what they contain at compile-time: The catalog contains a list of commodities that cannot vary at run-time. Hence, we can construct our canonical form at compile time and check that there is only one graph.
2. Commodity composition is done through the "part" production. Similarly to the list of interfaces, the composition-structure cannot change at run-time: It either is a part of the commodity or it is not.
3. The productions required for this constraint are also unconditional at compile time: the parts, the super-types, specified with the "supertype" production, and the category memberships specified with the "incategories" production.

Frameworks and Canonical Forms

In contrast to the ease of checking the static constraints for the DSL, the following is an example where the canonical form cannot be extracted, and, hence, the corresponding constraint cannot be checked.

The example shown in listing 4 describes a digital TV as the combination of an analog TV and a digital decoder with a remote control. Listing 5 shows the same example but with two if-tests added. This is perfectly legal for the catalog-construction framework but is disallowed by the DSL version.

Listing 4: Digital TV Example (a). The resulting canonical form for constraint no. 1 is shown in figure 2(a).

```

...
GeneralCommodity tv = new GeneralCommodityImpl("Digital TV");
tv.addPart("analogTV", "Analog TV");
tv.addPart("digitalDecoder", "Digital Decoder");
tv.addPart("decoderRemoteControl", "Decoder Remote Control");
addGeneralCommodity(tv);
...

```

Listing 5: Digital TV Example (b). The resulting canonical form for constraint no. 1 is shown in figure 2(b).

```

...
GeneralCommodity tv = new GeneralCommodityImpl("Digital TV");
tv.addPart("analogTV", "Analog TV");
if(caseA)
    tv.addPart("digitalDecoder", "Digital Decoder");
if(caseB)
    tv.addPart("decoderRemoteControl", "Decoder Remote Control");
addGeneralCommodity(tv);
...

```

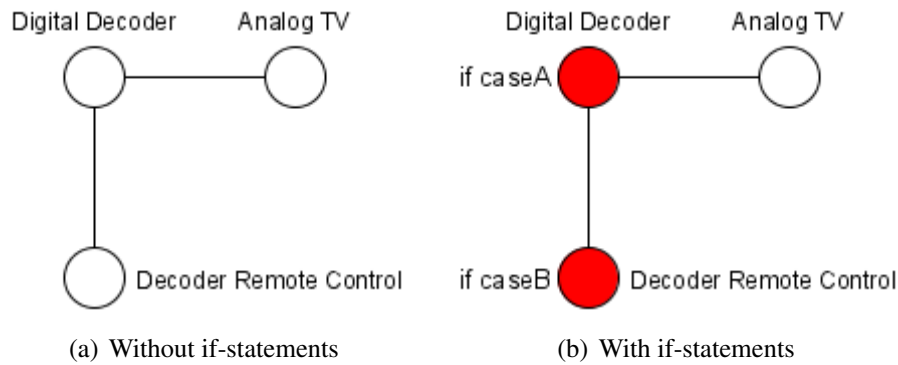


Figure 2: Two Canonical Forms for Constraint Nr. 1

For a Java framework, we have no mechanisms to ensure that the canonical forms are fixed at compile-time. If a programmer adds an if-statement then the canonical form cannot be properly extracted from the code without resolving the if-statement. The same is true for other language constructs that cannot be resolved at compile-time.

If we try to extract the canonical form for static constraint nr. 1 then we get as far as figure 2(b). What we really want is 2(a). The difference are the colored nodes that are conditional on "caseA" and "caseB", respectively. These cannot be resolved at compile-time, because the values of "caseA" and "caseB" are not available at compile-time. If they were then the canonical forms could be extracted and the constraint checked.

Implementation Accuracy

How well the canonical forms are extractable from code is called an implementation's *implementation accuracy*. The catalog construction language allows all programs written in it to have the three canonical forms extracted. Hence, the implementation accuracy with respect to the canonical forms is 100%. For the framework there are a number of things a user is allowed to do by the framework which makes checking the constraints impossible. For example, by adding an if-statement as in listing 5. This makes the implementation accuracy of Java with respect to the canonical forms of the example domain less than 100%.

Discussion

Since the framework failed to support one of the example constraints, one may question the design choice of the framework. One could, for example, represent certain parts of the framework-based programs as a tree of objects. This tree could be interpreted by the framework as an abstract syntax tree. This would be very odd to use, and one would question if one is really programming a framework and not a DSL.

The reason why the framework in this paper was designed the way it was is because it is a natural way to make and to use a framework: It utilizes as many of the language abstraction mechanisms as is practical to solve the problem, and with which a programmer who uses the framework, programs as usual in the programming language. It is surely the case that not all static constraints are impossible to check, but the range of static constraints possible to check is different from DSL programs in the case of the experiment in this paper.

Grammars are used to specify how to construct sentences in a language. It is precisely this mechanism that allows a language developer to restrict what is allowed in the language. DSL programs must strictly follow the grammars of the DSL. The semantics of a DSL is even less restricted than the power of the grammars. The semantics of valid sentences in the language can be chosen freely and is usually implemented with the full power of a general purpose programming language. This indicates that the DSLs in general have the power to implement complex domain models. When using DSL technologies that are more limited than this then one may face bigger problems checking the static constraints.

When mapping a domain model to a framework, one is bound by the grammars and by the semantics of the implementation language and the technology. A framework is, by its nature, tightly integrated with the implementation language structures. One must look at these structures in order to judge what is practical to map from the domain model, while still making the framework fit into the implementation language. For the Java language and for the implementation chosen in this paper, there is a failure in restricting the code that can be written sufficiently to be able to extract the example constraint's canonical form.

Reflection on statements is not common in programming languages today in any form, but statements of structured programming are common in the usage of framework. This indicates that it is difficult to limit which user actions can be restricted when using statements, and, hence, which canonical forms can be extracted. If one does not utilize statements in the usage of a framework, but just other constructs that can be commonly reflected upon, then the power of limiting user actions are clearly higher. This, however, may be awkward for a user of the framework. It depends on the domain model and its relation to the implementation language used to implement a framework.

4 Related Work

It is taken for granted that static-semantic analysis is not an issue for frameworks, while it is obvious for a DSL because it is a language. In [MHS05] it is said that DSLs have their power in being languages (being more expressive than frameworks, and being subject to static-semantic analysis), but that most of them never come beyond being frameworks that are embedded in general purpose languages.

In [BV04] the idea is that class libraries and frameworks encapsulate domain knowledge, e.g. in terms of interfaces to these classes, and that this ought to be available in terms of concrete domain specific syntax, not just in terms of method calls in the form of the general purpose language in which the classes are made. Therefore, the domain specific syntax is embedded into the host language. The benefit with embedding is that the features of both the host language and its tools are available, e.g. static semantics and type checkers.

[ABC07] investigates how to perform "automatic extraction of framework-specific models from framework-based application code". These framework-specific models are linked to the canonical forms and to the domain model which was a basis for the implementation of the framework.

[DDL07] is a paper about having sub-method reflection. This is interesting with respect to frameworks in that the contents of a method can be checked at compile-time using reflection.

5 Conclusion

This paper showed DSL and framework implementations of one example domain model. Three example static constraints were identified within the domain such that all were properly implemented by the DSL, but at least one of them failed to implement for the framework. It was then investigated into why the framework could not properly support checking the constraint. A thought experiment ended in the conclusion that this depends on the ability of the framework to restrict the user's code sufficiently to generate the required input for the static constraint checker. The input was termed the constraint's canonical form, and the extent of supporting it was called the framework's implementation accuracy. It was argued that one can look at the canonical form of a constraint in order to judge whether the constraint can be supported by a particular implementation or not. The comparatively lower control of a user's usage of a framework was argued to be the reason why frameworks do not support static semantic analysis as well as DSLs do.

6 Future Work

If we look at Java as a framework implementation language compared to, for example, C then we see that Java has potential for higher implementation accuracy (as the term is used in this paper). Java has the same structured programming mechanisms. It also has object-oriented mechanisms that increase the amount of things that can be done to limit what the users can do. Examples are encapsulation and polymorphism. Encapsulation allows the framework programmer to chose which parts the user has access to. Polymorphism lets the framework programmer select certain hot spots the users may modify. General solution enabling mechanisms such as callback methods are available also in C.

There are possibly mechanisms which when added to Java makes it possible to support static-semantic analysis for Java frameworks. Object-orientation was a big step forward in being able to control what the user of a general solution is allowed to do. Such additional mechanisms will probably come with drawbacks. If framework programmers are allowed to control what users can do on this low level then it may threaten the ability to base a solution on several frameworks. This issue is linked with the problem of combining DSLs. If a framework could get the benefits of DSLs and still retain the benefits of using several frameworks in one application then we are closer to the goal of being able to combine DSLs. This suggests that such mechanisms are probably difficult to find or comes with drawbacks similar to the drawbacks of using a DSL.

One possible mechanism that can control what the user is allowed to do inside a method is sub-method reflection. Using this technique a framework programmer can disallow, for example, if-statements within certain methods. Sub-method reflection is discussed in [DDL07].

7 Acknowledgements

The material presented in this paper is a part of the master thesis of Martin Fagereng Johansen presented at the University of Oslo in June 2009, supervised by Birger Møller-Pedersen.

References

- [ABC07] Michal Antkiewicz, Thiago Tonelli Bartolomei, and Krzysztof Czarnecki. Automatic extraction of framework-specific models from framework-based

application code. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 214–223, New York, NY, USA, 2007. ACM.

- [BMR⁺96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, and Peter Sommerlad. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, August 1996.
- [Bos98] Jan Bosch. Design patterns & frameworks: On the issue of language support. In *Object-Oriented Technologys*, pages 133–136. Springer Berlin / Heidelberg, 1998.
- [BV04] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2004. ACM.
- [DDL07] Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. *Journal of Object Technology*, 6(9):275–295, 2007.
- [Ecl08] Eclipse Foundation. The eclipse modeling framework (emf) overview, July 2008. <http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html>.
- [Gri75] M. Griffiths. Relationship between definition and implementation of a language. In *Software Engineering, An Advanced Course, Reprint of the First Edition [February 21 - March 3, 1972]*, pages 77–110, London, UK, 1975. Springer-Verlag.
- [JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.
- [MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.
- [Pre94] Wolfgang Pree. Meta patterns - a means for capturing the essentials of reusable object-oriented design. In *ECOOP '94: Proceedings of the 8th European Conference on Object-Oriented Programming*, pages 150–162, London, UK, 1994. Springer-Verlag.
- [Spi01] Diomidis Spinellis. Notable design patterns for domain-specific languages. *J. Syst. Softw.*, 56(1):91–99, February 2001.
- [WG84] William M. Waite and Gerhard Goos. *Compiler Construction*. Springer-Verlag, 1984.