Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 1333–1340
ISBN 978-83-60810-51-4

Granulated Code Generation of Interfering Functionalities

Igor Gelfgat
School of Computer Science,
Tel Aviv University
Email: igor.ge@gmail.com

Shmuel Tyszberowicz
School of Computer Science,
The Academic College of Tel Aviv Yaffo
Email: tyshbe@tau.ac.il

Amiram Yehudai
School of Computer Science,
Tel Aviv University
Email: amiramy@tau.ac.il

***Abstract*—The Model-Driven Software Development approach is becoming widely used as powerful model-driven tools are becoming available for the developer. Yet, it is not suitable to model, and therefore to generate the code, for all the aspects handled in the development stage. As a result, MDSD is not as widely used as it could.**

This paper presents a technique that extends the capabilities of Model-Driven Engineering with behavioral aspects, by modeling concerns and using them in code generation. Common concerns can be defined for design patterns, software infrastructures and other common aspects. Independent concerns can be effectively combined when applied to the same model element. Software architects are advised to apply common concerns to their system models and also to create system-specific concerns and apply them at the modeling stage. We name it *enriching a model with concerns*.

With the help of code definition for each concern, our tool automatically generates code for the enriched model. Thus, at the end of the modeling stage the developers will have the structure of the code and all the glue code ready, so they will only have to fill the business logic in the manual implementation methods created for them. They will also maintain the enriched model and not the code they would otherwise write manually.

***Index Terms*—Model-Driven Engineering, metaprogramming, code generation, aspects**

I. Introduction

THE model-driven approach to software development becomes more and more popular. Its primary goals are portability, interoperability and reusability, through architectural separation of concerns [1]. When the model is the primary artifact of the development process, the user works on the model, and a tool generates all or part of the code. This raises the productivity and the quality of the process. Productivity is raised as the auto-generation takes care of all the plumbing code. The generated code is of very high quality since it was fully tested beforehand. Though the model itself still has to be tested, it requires much less effort than testing the whole system. Thus, it is effective to maintain and fix the model instead of the code when possible [2].

It is easier to understand a model rather than the code, since it is a more abstract definition of the system. Of course, when the code is automatically generated we wish its behavior to be clearly understood from its model definition, to allow us to avoid looking at this code at all. The model-driven technique obligates to work with the model itself and not with the code, keeping the model consistent. Again, it shortens the development time and increases the implementation quality.

A software system has structural and behavioral aspects. Both should appear in various levels of abstraction in the system model, as it is more natural for a human to start from a high-level description and then gradually get down to details. Presently, using model-driven techniques mostly means constructing structural models, but not behavioral ones. Once behavior is modeled, it is often done at the same level of abstraction as implementing, and requires the same effort.

It is hard to find a collaboration of structural elements such that it will be possible to automatically generate their implementation. There are, however, some exceptions. For example, when we have a structural model where the behavior of some of its elements is similar. In this case one can apply stereotypes [3], enabling code generation for this behavior. Also, there are modeling techniques that define behavior for a single type of systems, but do not suit other types. For example, state diagrams can be used to model events-based systems [4] [5], but are not suitable for, e.g., data management systems.

We have developed a technique to define the behavior of a general system or part of it, which enables us to generate automatically as much of the code as possible, such that only the business logic will be left to implement. The intention is similar to that of Aspect-Oriented Programming (AOP) [6] that, among other things, targets at minimizing duplicated code. Here we aim to minimize the effort to apply similar behavior multiple times.

In our technique models are *enriched with concerns*, enabling automatic code generation for the structure and the behavior that concerns contribute to the applied model elements. The most valuable is the ability to pass data between two independent concerns. This gives us an ability to combine concerns on the same model element without an additional effort to define their combined behavior.

The Oxford Dictionary defines *granulated* as “formed into grains or particles”. The code generation technique is named *Granulated Code Generation* to emphasize the fact that the generated code is built of small parts coming from many functionalities applied to the model elements.

The rest of the paper is organized as follows. Section II demonstrates the expected results with examples. In Section III

978-83-60810-51-4/$25.00 © 2012 IEEE

we describe the technique, explaining how to model with concerns and how the generation process works. Section IV presents related work. Section V summarizes the paper.

II. Examples

We define *concerns* (or *functionalities*) as special model entities that can be applied to regular model entities (classes, members and operations). This enables us to specify general behavior of a class at the model stage and to generate automatically part or all of its code. We use both the terms concerns and functionalities interchangingly. Detailed descriptions of concerns are provided in Section III.

A. Examples of Resulting Generated Code

Following are examples of classes that we want to model using concerns and have their code generated automatically. Here we present the code as it would be written manually and explain how similar code could be generated. The examples demonstrate the results we want to achieve. See Section III-E for an example of generated code.

Example 2.1: This example describes the method *makeOperation* of the class *SomePrivilegedOperation* that, in addition to its main operation, implements also the following requirements: only privileged users may execute the method, it requires a database connection, and its success or failure is logged.

- *Operation within a transaction.* A transaction is an operation with external resources (for example, a database) that has only two possible outcomes: a success with all needed changes performed and a failure with no change performed. For the operation to be successfully performed, we should locate or create a transaction and a database connection within the transaction.
- *Privileged operation.* We want the method to run as a privileged operation, thus we use Role Based Access Control (RBAC) which allows us to assign privileges to users or user groups to run this operation. For a full description of how RBAC may be modeled in UML and how automatic security code may be generated from it, refer to [7]. We implement RBAC using a database, therefore, a database connection within a transaction is needed.
- *Operations log.* Every time the action is executed, the following information should be saved: the operation description, the acting user name and a success/failure/unauthorized status.

To apply the requirements for the method using concerns, the method model will look like:

```
<< operationAccounting ,
   currentTransactionConnection(connection),
   protectedOperation(connection , currentUser) >>
makeOperation(currentUser : User) : boolean
```

In the model, we see the method signature and the functionalities applied with concerns. In Sections III-C and III-D we define how concerns are modeled and applied to classes and methods. Listing 1 displays a Java implementation of the example.

Following is a brief explanation of how the implementation code can be generated from its model. First, the transaction code is inserted. Then permissions are checked using the parameter *currentUser*. To specify a user to be checked for privileges to run the operation, the *currentUser* is supplied as a parameter for the applied functionality at the modeling stage. The user implementation code is inserted after the permissions check. At every point before the return, the operation is logged including its success/failure status.

```
1  public class SomePrivilegedOperation {
2    public boolean makeOperation(User currentUser) {
3
4      try {
5       // locates current transaction and retrieves
6       // a connection within it
7       Connection connection = TransactionManager
8                             .getCurrentTransaction();
9
10      boolean authorized = checkPrivileges(
11                            connection, this, user);
12      if (!authorized) {
13
14       // removed: log ''unauthorized''
15       return false;
16      }
17
18      boolean success;
19      // removed: the operation implementation
20
21      // removed: end transaction
22
23      // removed: log success
24      return success;
25
26     } catch (SQLException e) {
27      // sql error
28
29      // removed: log failure
30      return false;
31     }
32    }
33 }
```

Listing 1. Example 2.1 implementation code. Lines 4-8, 18, 26-27 and 30-31 handle database related behavior. Lines 10-12 and 15-16 check access rights to the privileged operation. Lines 14, 23 and 29 log results. *Note: the "removed" lines in code fragments from here on represent code that was removed to save space.*

Example 2.2: The second example displays how we may affect code structure and, particularly, how a class can be altered to use multiple design patterns, just by applying concerns. See Fig. 1 for the example's model.

- A singleton class [8]. A single instance of the class is retrieved through a *getInstance()* method. In modeling and generation stages every reference to the singleton class is a reference to the same object.
- An object pool [9]. We use two object sets (a set of free objects and a set of objects in use) and methods to borrow them for work and to return them back to the pool.

An implementation code is shown in Listing 2. Both patterns are at a class level, but cause generation of different methods and fields. The singleton pattern brings in a static method and static fields, while the object pool pattern leads to generation of regular methods and fields.

Consider a more complex case, where a concern is applied to some other class *OtherClass* with a link to the class in the

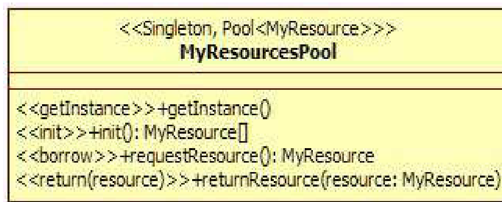

Fig. 1. The model for Example 2.2.

example. Also, the link is assigned a role, such that the link should be used in the generated code for *OtherClass*. Any such usage should take into account the singleton pattern and access the pool object using the *getInstance()* method.

```
/**
 * @concern Singleton
 * @concern Pool<MyResource>
 */
public class MyResourcesPool {
  /** @concern Singleton */
  private static MyResourcesPool theInstance;
  /** @concern Singleton */
  private static Object lock = new Object();
  /** @concern Pool<MyResource> */
  private List<MyResource> availableList;
  /** @concern Pool<MyResource> */
  private Set<MyResource> inuseSet = new HashSet();

  /** @concern Singleton */
  private MyResourcesPool() {
    availableList = Arrays.asList(init());
  }

  /** @concern Singleton */
  public static MyResourcesPool getInstance() {
    if (theInstance == null) {
      synchronized (lock) {
        theInstance = new MyResourcesPool();
      }
    }
    return theInstance;
  }

  /** @concern Pool<MyResource> */
  protected MyResource[] init() {
    // removed: resources initialization
  }

  /** @concern Pool<MyResource> */
  public MyResource requestResource() {
    // removed: if there is no available object -
    // wait for it

    MyResource resource = null;
    synchronized (this) {
      // removed: find object in availableList,
      // move it to inuseSet and return it within
      // resource variable
    }
    return resource;
  }

  /** @concern Pool<MyResource> */
  public void returnResource(MyResource resource) {
    synchronized (this) {
      // removed: move the object from inuseSet
      // to availableList
    }
    // removed: notify that there is a free object
  }
}
```

Listing 2. Example 2.2 implementation code. *Note: the @concern tag indicates either a concern is applied to a class, or a member/method is generated as a result of applying a concern.*

III. Granulated Code Generation – the Technique

In this section we describe a technique that enables us to automatically generate code from a model that includes behavior information.

A. General Description

The generic model technique and generation process that we have built enables system architects and designers to define common functionalities and to apply them to software model entities. As the result the target system is constructed, having part of it generated from modeled structure and behavior, and the other part is manually implemented business logic.

The functionalities that we call *concerns* are model entities (in our examples they are *UML classes*) that may be associated with (applied to) regular entities (other *UML classes*) that represent system entities and probably result in generated DB tables, classes in code, etc. Concern members may be associated with class members, and concern methods – with class methods.

Once a concern is associated with some class it affects the generated code for the class, adding some functionality (defined by a plugin, supplied with the concern), saving the developer's time needed to write this code manually.

To make this technique useful, it should be able to apply a wide range of generation functionalities to a model. Each functionality is generally independent from others or depends only on a small number of other functionalities. Thus, it should be easy to add new functionality definitions to the tool. The architecture of the tool is based on a "plugins framework", where a new functionality is added to the tool through a new plugin that is not aware of other plugins, except those it depends on.

B. As Part of the MDE Process

The technique incorporates completely into the MDE process. First, architects and software infrastructure teams identify common functionalities (in addition to those widely used or already existing within the organization), prepare their definitions and code generation plugins. Then, designers apply them to the model at the proper level of abstraction (model transformations enable us to easily use models on various abstraction levels). Some details for an already applied functionality may be added on a lower level. The lowest level model is used for the code generation. Developers then extend the generated code to complete the implementation.

At any time, changes may be easily done to the model and applied concerns. They directly affect the generated code, making maintenance easier.

In [10], Uhl, Koch and Weise discuss how a higher level of abstraction affects effort needed to develop and to maintain a system. Similarly, abstraction of behavior helps to make the model more intuitive. Generation of the code from the enriched model minimizes manual code, therefore increasing program quality and demanding less maintenance.

C. Defining Concerns

Following we describe how to define a *concern* in a model. Such a definition enables a designer to apply it to a class in the design model. In addition, a plugin should be supplied to the generation tool for a particular concern, to enable generating the code that implements the capabilities that the concern is meant to provide when applied to a class. This plugin is not to be seen by anyone during the system design or the implementation stages. Some documentation is required, however, to describe the semantics of applying the concern.

There exist different kinds of concerns. Some interfere at method level[1], some at class level[2], and some at both[3]. Some introduce parameters, others introduce fields or methods. There are also more complicated concerns that are related to other concerns (interfere at inter-object level). Following is a description of an approach to define a concern.

Defining a concern means to provide its name, to describe fields and methods it can be applied to as well as relationships with other concerns. We do it by creating a *UML class* with a <<concern>> stereotype. The name of the UML class that represents the concern is the name of the concern, its attributes and operations describe fields and methods the concern may or should be applied to within the applied class (*enriched class*). UML meta-model of a concern is presented in Fig. 2.

A UML attribute in a concern (i.e., in the UML class that represents it) defines an annotation that should be applied to an attribute in the enriched class in the design model. We call it *concern field* hereafter. The attribute it is applied to is called *enriched field*.

A UML operation in a concern (named *concern method*) defines an annotation that should be applied to an operation of the enriched class (named *enriched method*). Concern methods may have one or more of the following stereotypes applied:

- <<optional>> If not otherwise specified, a concern method is required for the concern. Only when the <<optional>> stereotype is used for the concern method it does not have to be specified in an enriched class.
- <<full_implementation>> This concern method supplies full implementation for a method it is applied to. It should be the last one applied, because further applied concern methods are meaningless. Similarly, no additional manual implementation of the enriched method is required.
- <<frontend>> When applied, the parameters of the enriched method should conform to the parameters of the concern method: the same order of parameters and each enriched method parameter is assignable from the corresponding concern method parameter. This kind of a concern method enables calling an enriched method from the generated code.

[1]The concerns in the Example 2.1.
[2]The concerns in the Example 2.2.
[3]Pool example, see Fig. 3.

A defined concern method may be parameterized to pass information about the desired functionality: data structures, types, etc. Some parameters require special notation, achieved by adding a stereotype to a parameter, as following:

- <<introduced>> A value of specified type is calculated by the concern and supplied as an additional argument to the subsequent concern methods and the implementation method. When used in a model, only a name of the new argument is specified in place of the parameter.
- <<erased>> If a method argument or an introduced parameter (see Section III-D) is passed to an <<erased>> concern parameter, the argument/introduced parameter is used only by the concern, and is not available for use by neither concern methods applied after it, nor the implementation method. This stereotype is for convenience - to avoid developers seeing unneeded parameters (used by the concern only).
- <<returns>> A value of specified type is calculated by the concern and returned to the caller. The concern method hides the return value it receives from the called method and returns the calculated one. It is helpful when the received value is processed by the concern method and a value of another type or even nothing (*void*) is returned by the concern method. A concern method can be defined as returning a value also when it receives *void* or in the case it is a <<full_implementation>>.
  At most one parameter of this kind may be present in a single concern method. It is possible to define it as a return type of the concern method (see Fig. 3), except for the case of *void* type.
- <<return_type_required>> Type of an enriched method that the concern method may be applied to. The concern method uses the value it accepts from the method called. At most one parameter of this kind may be present in a single concern method.
- <<exception>> Exception thrown by the concern. Should be declared in the signature of the enriched method or being handled by one of the previous concerns. By default, the signature of the implementation method will not contain this exception.
- <<handled_exception>> An exception of this type, if thrown from the subsequent concern method/implementation method, is handled by the concern and is not seen by the caller.

Note that a parameter can not be annotated with more than one of the types.

The last four types do not require a parameter to be supplied with a value, so the parameter appears only in the concern definition.

An example of a concern definition is shown in Fig. 3. It defines a concern for a *pool of objects* that allows reusing objects instead of recreating them (when it is expensive to create them for each use). The pool has a template parameter that defines the type of stored objects and three methods each pool of objects has: *init()* for the first initialization of objects,

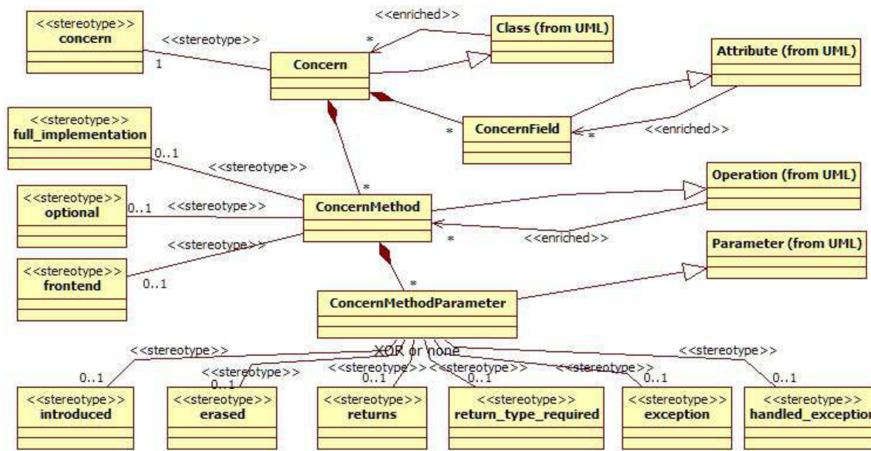

Fig. 2. Concern meta-model.

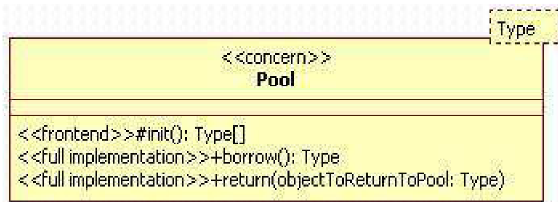

Fig. 3. Definition of an object pool (from Example 2.2).

borrow() – taking an object out of the pool and *return()* – returning previously borrowed object. The initialization method is a <<frontend>> method, because it is called from inside the automatically generated code. The other two are <<full_implementation>> – no manual implementation is required.

D. Applying Concerns

An element in a model can be described in various ways and by various relationships, annotations, etc. Each description is meaningful for the code generation and should be expressed in the generator definition. This allows the generation tool to identify the element's behavior.

A class can be instrumented with one or more concerns (see Fig. 4). In this case, every non-optional concern method defined in the concern should be applied to one of the methods in the enriched class. Every concern field should be applied to a field in the enriched class. When a number of concerns are applied to a class, each of them is taken into account separately, but they should not have collisions (i.e., fields with identical names or methods with identical names and parameter types).

Each method may have a number of concern methods applied to it. The concern methods are treated in their order of appearance for the enriched method: on a call they will run in that order, and on return their "after" parts will run in the reverse order. It is reminiscent of aspects (of AOP). Concern methods are applied in the following way:

- A method's signature is defined as it should appear to the caller, i.e., it is the signature of the full method including all applied concern methods. Hence a developer, who has to call the generated code for the enriched method, sees the same method signature as it appears in the model. Please note that it is the same signature as if the class was modeled without using concerns and rather was implemented manually.
- From an applied concern method's point of view, it is running in a "method" (which does not exist really) that has a list of parameters, a set of exceptions and a return type. The rest of a structure it is applied to (i.e., the following concern methods and the implementation method) is also a "method" with another list of arguments and a return type. The rules and the restrictions are as follows:
  - Every parameter supplied to a concern method should be accessible by it. Therefore it should be defined in the signature or introduced by one of the previously applied concern methods, and should not be hidden (<<erased>>) by any of them. See (1) and (2).
  - The implementation method will have its abstract signature generated in the code and should be implemented manually, except for the case where the last applied concern method is a <<full_implementation>>.
  - The implementation method will contain all non-hidden parameters of the method's signature as it is defined and all introduced (but not hidden) parameters in all applied concern methods.
  - If none of the applied concern methods requires a return parameter, the return type of the implementation method will be the same as of the enriched method. Otherwise, the return type will be as needed by the latest concern method that has a return parameter. See (3).
  - Chain of return types. Every concern method that

Fig. 4. A Web Service with 3 authorized-only methods authorization and database connection behavior, defined by applying concerns. The methods are similar to the method in Example 2.1.

expects a certain return type should comply with the return type of a following concern method with a return parameter, if exists. See (4).

A signature of a method is defined as (P, r, Ex), where P is a list of parameters, r is a return type and Ex is a set of exceptions. Enriched method signature is marked as (P_0, r_0, Ex_0). Similarly, its implementation method signature is marked as: (P_n, r_n, Ex_n), where n is the number of applied concern methods.

The rules described beforehand are explained using the equations below. The equations are for each i, $i = 0\,..\,n{-}1$.

$$P_{i+1} = P_i \bigcup \text{Introduced}_i \setminus \text{Erased}_i \tag{1}$$

$$Ex_{i+1} = Ex_i \bigcup \text{Handled}_i \setminus \text{Thrown}_i \tag{2}$$

$$r_{i+1} = \begin{cases} r_i, & \text{if} \quad \text{returns}_i = \perp \ \bigwedge \\ & \quad \text{returnTypeReq}_i = \perp \\ \perp, & \text{if} \quad \text{returns}_i \neq \perp \ \bigwedge \\ & \quad \text{returnTypeReq}_i = \perp \\ \text{returnTypeReq}_i, & \text{otherwise} \end{cases} \tag{3}$$

$$r_i \leq returns_i \tag{4}$$

Fig. 4 presents an example of an HR service that creates, adds and removes an employee in an organization database. The service is defined as a <<*WebService*>> so it will be created and deployed as a Web Service and all its parameters will be presented in its Web Services Description Language (WSDL) file [11]. A connection is supplied to every method, so it will be able to perform database queries. Each method is declared with authorization checks. The only thing that is left to be implemented manually is to perform a query for each of the methods.

Fig. 5 illustrates the enriched method *getAllEmployees()*, generated parts for concern methods applied to it, and the arguments and the return type of each part and the implementation method. These would be actual method signatures for all parts, if we were creating them in separate methods.

The application on the signatures is as the following. Concern method *currentTransactionConnection* adds its <<introduced>> parameter *connection* that becomes available for the subsequent concerns. Concern method *protectedOperation* removes the <<erased>> parameter *user*. Finally, both *protectedOperation* and *protectResultObject* throw *ProtectionException*.

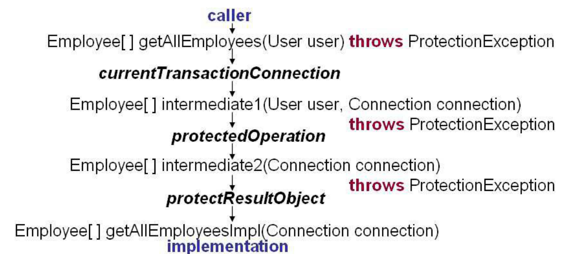

Fig. 5. Schematic view of the enriched getAllEmployees() method, the arguments and the return type of each generated part (if they were separate methods) and the implementation method.

E. Resulting Code

In this section we present a *MySet* example, a set of objects, combined with a *Visitor* design pattern: the method *checkAllObjects()* passes over all elements in the set and performs a manually implemented operation on each object. Listing 3 shows its generated code in the class *MySetBase*. The manual code is placed in the class *MySet*, implementing the abstract method, and adding any fields and methods not mentioned in the model. Thus, the manual code is not affected by regeneration of the automatic code, since they are in separate files. Moreover, all the automatic code is in a separate source folder.

```
/** Generated base class of MySet class
 * in the model. */
public abstract class MySetBase {
  private MyObject[] myObjects;

  /** Generated method for the one specified
   * in the model. */
  public final void checkAllObjects() {
    for (MyObject element : myObjects) {
      this.checkAllObjects(element); }
  }

  /** It is a generated method that should be
   * implemented. @param object the parameter
   * introduced by the visitor */
  protected abstract void checkAllObjects(
      MyObject object);
}
```

Listing 3. Generated code for MySet

F. Tool Prototype

We have implemented a prototype [4] to check our ideas. It receives a model that contains concerns and enriched classes. Having plugins supplied for the concerns, the tool generates

[4]See http://sites.google.com/site/gcgtool/

the code. We have successfully generated code for two examples: a small data management system and a distributed algorithm.

The first example is a bank account management system. It uses concerns for database connectivity, security, multithreading, error checking and design patterns. The application has the following features: managing multiple accounts (check balance, deposit and withdraw), managing access rights and access log, checking legal account state (no overdraw) and committing operations per time period (business day) in a separate thread. Its manual code consists almost only of database queries (that may be handled by existing MDE techniques, e.g., AndroMDA [12]) and GUI.

The second example is a mechanism for distributed algorithms, based on asynchronous events. The mechanism uses concerns for multithreading and communicating with events. The mechanism enables sending named events with additional data asynchronously to any interested object. There is also a concern for defining distributed nodes. It provides methods to manage node's neighbours and a way to define events sent to / received from the neighbours. With its help we have implemented the Chang and Roberts ring-based election algorithm [13]. Its manual code consists only of the algorithm logic and code that builds the ring (the algorithm logic could be generated from statecharts [5]).

To turn the prototype into a complete and useful tool, we have to extend its errors handling, allow multiple models as an input and integrate it with other MDE tools (model editing, testing tools, etc.).

IV. Related Work

Lodderstedt, Basin, and Doser [7] describe a way to model security requirements for a software system that enables generating security code automatically. This approach is based on applying roles to UML classes. It does not address other types of requirements besides security.

Harel and Politi [5] show how reactive systems may be modeled using statecharts. This technique enables the creation of a complete model of event-based systems, but in non event-based systems, statecharts are useful only for a small part of the system.

The Rhapsody tool enables full cycle model-driven development [14]. In Rhapsody, model and code are strongly associated, thus the model is never outdated, while the code remains the most important artifact at the development stage as it handles all the details. Some model elements also include code. Rhapsody enables model execution and model-based testing. Although Rhapsody is one of the most powerful MDE tools, it does not include an ability to effectively reuse common design patterns and logic, besides some built-in structures.

Comparing our technique to Aspect-Oriented Modeling (AOM) [15] [16] [17] [18] and Feature-Oriented Modeling (FOM) [19], AOM and FOM do not change method's signature (except for exceptions) and do not allow passing parameters between two aspects/features applied on the same method. Moreover, AOM does not provide aspects that may be applied to a class, enriching the class along with its methods with a complex behavior at once. Thus, our technique is more powerful, meaning more code is easier generated for non crosscutting concerns.

Compared to non-MDE techniques such as AOP and Feature-Oriented Programming (FOP), our technique makes the development process easier and more straightforward, as the association is done in the model stage.

A. Comparing to AOM techniques

There are various existing AOM techniques. Here we compare our technique with some of them in particular.

AspectOPTIMA [20] is a powerful language independent, aspect-oriented framework that was built as a case study for Reusable Aspect Models (RAM) [18]. To apply aspects to an application model, RAM uses Sequence Diagrams that define pointcuts and advices (the aspect behavior at pointcuts).

RAM, similarly to other AOM techniques, has the following disadvantages:

- Weaving (the term used in RAM for models, although the actual weaving is done at the runtime by AOP implementation) of aspects with application models, is done at the aspects level, using interfaces to describe entities that should be affected. This way it serves only crosscutting concerns.
- Weaving is done at pointcuts (methods) as in AOP. Our technique enables applying a concern as a whole, thus adding a general behavior that takes into account the enriched class with all its enriched methods.
- There is no change in signature for application methods.
- To pass parameters between two applied aspects, or to make them aware one of another for some other purpose, it is required to create an aspect that unites the two (see Figure 5 in [18]).

For a survey on some UML-based AOM approaches see [21].

V. Conclusions and Future Work

We have described an MDE technique to define common functionalities (concerns) and use them in modeling to get automatically generated code for an enriched model. It reduces the effort needed for software system development. We showed how different concerns can be applied to a single model element and how they can solve different programming and design problems (security, design patterns, exceptions handling, etc.).

Our technique handles both behavioral and structural concerns, allows to define generally any functional or non-functional concern and to combine multiple concerns at a single class or method. Other MDE techniques either do not handle the behavioral aspect, or do not suit for all software systems (data management, real-time, distributed, etc.). We successfully applied our technique using a tool prototype to generate and implement examples of data management and distributed event-based systems.

In the future we plan to deal with complex concerns and automatic ordering, to check alternative ways to represent concerns and to prepare a case study.

We can think about concerns that are aware of each other, for example generalizing of another concern, aggregation of concerns and other dependencies. When enriching a model it is currently required to state the full order for applying concerns. It may not be necessary in every case. Moreover, frequently two unrelated and non colliding concerns are applied together and may be executed in any order. Further research may be done on how it is possible to automatically determine (for a particular use) necessary order of concerns, even if it is not stated explicitly (also for colliding concerns).

We plan to conduct a wide case study to assess the contribution of the technique to the development process and to further check its applicability. Also an empirical data is needed to confirm this contribution.

We also seek for an alternative representation of concerns model with expression strength similar to the one described. We actually want to define a similar graphical representation to make the model more intuitive.

REFERENCES

[1] J. Miller and J. Mukerji, *MDA Guide Version 1.0.1*, 2003. [Online]. Available: http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf Visited July 2012.

[2] M. Azoff, *The Benefits of Model Driven Development*, 2008.

[3] A. Schleicher and B. Westfechtel, “Beyond stereotyping: Metamodeling approaches for the UML,” *Hawaii International Conference on System Sciences*, vol. 3, pp. 3051–3060, 2001.

[4] J. Lavi and J. Kudish, “Systems modeling & requirements specification using ECSAM: an analysis method for embedded & computer-based systems,” *Innovations in Systems and Software Engineering*, vol. 1, no. 2, pp. 100–115, 2005.

[5] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The Statemate Approach*. New York: McGraw-Hill, Inc., 1998.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, “Aspect-oriented programming,” in *Proceedings European Conference on Object-Oriented Programming*. Springer-Verlag, 1997, pp. 220–242.

[7] T. Lodderstedt, D. A. Basin, and J. Doser, “SecureUML: A UML-based modeling language for model-driven security,” in *UML ’02: Proceedings of the 5th International Conference on The Unified Modeling Language*. London, UK: Springer-Verlag, 2002, pp. 426–441.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[9] M. Kircher and P. Jain, “Pooling,” in *EuroPLoP*, 2002, pp. 497–510.

[10] T. Koch, A. Uhl, and D. Weise, *Model Driven Architecture*. [Online]. Available: http://www.omg.org/cgi-bin/doc?ormsc/2002-09-04 Visited July 2012.

[11] J. M. Vara, V. de Castro, and E. Marcos, “WSDL automatic generation from UML models in a MDA framework,” in *NWESP ’05: Proceedings of the International Conference on Next Generation Web Services Practices*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 319–324.

[12] *AndroMDA Framework*. [Online]. Available: http://www.andromda.org Visited July 2012.

[13] E. Chang and R. Roberts, “An improved algorithm for decentralized extrema-finding in circular configurations of processes,” *Commun. ACM*, vol. 22, pp. 281–283, 1979.

[14] E. Gery, D. Harel, and E. Palachi, “Rhapsody: A complete life-cycle model-based development system,” in *Proceedings of IFM*, ser. Lecture Notes in Computer Science, vol. 2335. Springer, 2002, pp. 1–10.

[15] Y. Wang, S. Singh, J. Hosking, and J. Grundy, “An aspect-oriented UML tool for software development with early aspects,” in *EA ’06: Proceedings of the 2006 international workshop on Early aspects at ICSE*. New York: ACM, 2006, pp. 51–58.

[16] N. Ubayashi, G. Otsubo, K. Noda, J. Yoshida, and T. Tamai, “AspectM: UML-based extensible AOM language,” in *ASE ’08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 501–502.

[17] D. Stein, S. Hanenberg, and R. Unland, “A UML-based aspect-oriented design notation for AspectJ,” in *AOSD ’02: Proceedings of the 1st international conference on Aspect-oriented software development*, New York, 2002, pp. 106–112.

[18] J. Klein and J. Kienzle, “Reusable aspect models,” in *Proc. of the 11th Workshop on Aspect-Oriented Modeling.*, 2007, p. 11.

[19] S. Trujillo, M. Azanza, and O. Diaz, “Generative metaprogramming,” in *Proceedings of the 6th international conference on Generative programming and component engineering*, ser. GPCE ’07. New York: ACM, 2007, pp. 105–114.

[20] J. Kienzle, E. Duala-Ekoko, and S. Gélineau, “AspectOptima: A case study on aspect dependencies and interactions,” in *Transactions on Aspect-Oriented Software Development V*, ser. Lecture Notes in Computer Science, A. Rashid and H. Ossher, Eds. Springer, 2009, vol. 5490, pp. 187–234.

[21] M. Wimmer, A. Schauerhuber, G. Kappel, W. Retschitzegger, W. Schwinger, and E. Kapsammer, “A survey on UML-based aspect-oriented design modeling,” *ACM Computing Surveys*, vol. 43, no. 4, pp. 1–33, 2011.