Integrating Legacy Systems with MDE

Mickael Clavreul, Olivier Barais, Jean-Marc Jézéquel

Integrating Legacy Systems with MDE*

Mickael Clavreul
INRIA
Campus de Beaulieu
35042 Rennes Cedex, France
mickael.clavreul@inria.fr

Olivier Barais
IRISA, Université Rennes 1
Campus de Beaulieu
35042 Rennes Cedex, France
barais@irisa.fr

Jean-Marc Jézéquel
IRISA, Université Rennes 1
Campus de Beaulieu
35042 Rennes Cedex, France
jezequel@irisa.fr

ABSTRACT

Integrating several legacy software systems together is commonly performed with multiple applications of the Adapter Design Pattern in OO languages such as Java. The integration is based on specifying bi-directional translations between pairs of APIs from different systems. Yet, manual development of wrappers to implement these translations is tedious, expensive and error-prone. In this paper, we explore how models, aspects and generative techniques can be used in conjunction to alleviate the implementation of multiple wrappers. Briefly the steps are, (1) the automatic reverse engineering of relevant concepts in APIs to high-level models; (2) the manual definition of mapping relationships between concepts in different models of APIs using an ad-hoc DSL; (3) the automatic generation of wrappers from these mapping specifications using AOP. This approach is weighted against manual development of wrappers using an industrial case study. Criteria are the relative code length and the increase of automation.

Categories and Subject Descriptors

D.2.12 [**Software Engineering**]: Interoperability

General Terms

Design, Reliability

Keywords

Legacy Systems, Aspects, Models, MDE

1. INTRODUCTION

As development techniques, paradigms, technologies and methods are evolving far more quickly than domain applications, software evolution and maintenance is a constant challenge for software engineers.

In a context of Enterprise Application Integration (EAI), a key concern is the translation of the outputs of some applications into inputs for other applications. Text-based formats can readily be handled with parsers, interpreters and text-based adapters. Even more easily, XML-based formats are nowadays supported by a wide range of tools or methods that help people convert information for a particular use, platform or software. The problem is quite different when one has to efficiently translate a continuous flow of data from one legacy API (Application Programming Interface) to another one. A possible solution could be based on an application of the Adapter Pattern [20] to map one call in API I_1 to slightly different calls that cope with API I_2, hence "*wrapping*" I_2 in such a way that it offers an interface compatible with I_1.

Multiple occurrences of the Adapter Pattern are then needed to integrate applications in such a way that they mutually inter-operate, including with previous versions of themselves. Even if a kind of *intermediate* API could be used to reduce the number of adapters from $2 * n * (n-1)$ down to $2 * (n+1)$ for n applications, that still leaves us with a lot of tedious and error-prone adaptation code to be developed when n is large. In some domains, such as the management of heterogeneous on-line equipments for digital video broadcasting, a steady flow of both requirements and third party new products makes it very hard to both keep up with evolutions and still guarantee backward compatibility and interoperability between versions: even the intermediate language has to be constantly refined, leading to expensive maintenance operations.

The contribution of this paper is to propose a model-driven approach to alleviate the implementation of multiple wrappers in that kind of context. Our approach is technically based on three steps: (1) the automatic reverse engineering of relevant concepts from APIs to high-level models; (2) the manual definition of mapping relationships between concepts in different models of APIs using an *ad hoc* Domain Specific Language (DSL); (3) the automatic generation of wrappers from these mapping specifications using Aspect Oriented Programming (AOP) to avoid changes in legacy APIs. Our proposition highlights how models, aspects and generative techniques might be used in conjunction with legacy code to reduce costs and increase efficiency in software development. The remainder of this paper is organized as follows. Section 2 introduces a motivating example to illustrate our approach. Section 3 outlines the global process of our solution. Section 4 illustrates the first step of our approach and describes how we use reverse-engineering to create API-specific models. Section 5 and Section 6 refer to the second step of the process and describe the mapping lan-

*This work has been partially supported by the MOPCOM-I Project from the Competitiveness Cluster of Brittany.

ICSE '10, May 2-8 2010, Cape Town, South Africa

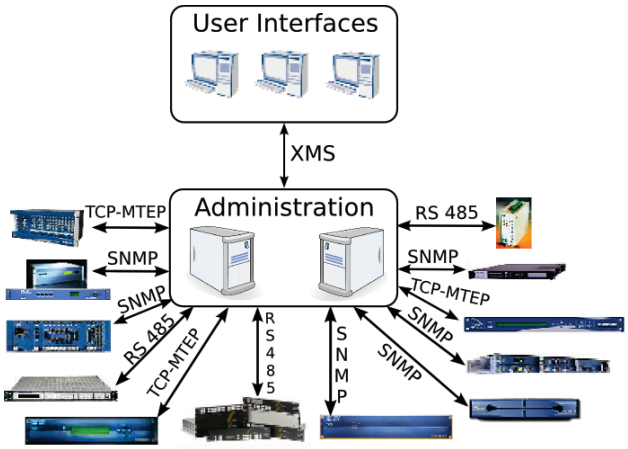

Figure 1: Global View of the Management Architecture with some examples of managed physical devices

guage, how it is used and how mappings implementation is achieved. Section 7 deals with the automatic generation of wrappers from mappings specifications using aspect-oriented techniques. Section 8 compares our solution on the case study to the manual implementation of adapters, in terms of effort. Section 9 discusses articles and ideas related to our work. Finally, Section 10 concludes this paper.

2. MOTIVATING EXAMPLE

2.1 General Description

As a motivating example, we are going to consider the domain of configuring and managing heterogeneous equipments for video and broadcasting, such as Thomson Extensible Management System for Digital TV. This management system deals with the intercommunication of legacy hardware devices (*i.e.* Network Adapters, MPEG Multiplexers, Encoders, Decoders,...). As shown in Figure 1, digital devices are designed by different manufacturers and from different technologies. Each one provides a specific API for management purposes. For management and configuration concerns, Thomson provides a distributed architecture with a set of remote user interfaces and administration servers that communicate with each other through an intermediate API called XMS. Administration servers main responsibility is thus to convert XMS orders into device-specific ones as shown in Figure 1.

To allow integration with existing platforms and systems, Thomson has been developing APIs for an extensive set of protocols such as MTEP, SNMP, XMS, TCP/IP, RS232/485 and so on (see Figure 1). The evolution of legacy equipments induces the development and the maintenance of several versions of APIs, both for the specific protocols and for the intermediate language XMS. This situation leads to the combinatorial explosion of the number of wrappers to be developed.

2.2 Translation Issues

In Figure 2 we present the specification of the mappings between objects from the MTEP API to the XMS API. The MTEP specification defines groups of devices. Devices are either nominal or redundant. Nominal devices are preferably used by the system to perform some treatment. If nominal

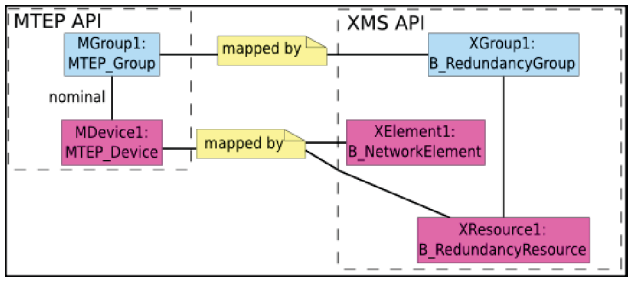

Figure 2: Example of Objects Diagrams with Mapping Information

devices are not available because of bugs or failures, the system uses redundant devices that offer equivalent functionalities. While the MTEP API qualifies the redundant or nominal characteristic in the form of a relation between a group and a device, the XMS API proposes two different objects. XMS defines nominal devices as B_NetworkElement and redundant devices as B_RedundancyResource. The concept of group (B_RedundancyGroup) is available only for redundant devices.

The semantics of "*mapped by*" is informally defined at the level of the specification while experts from the domain really define the semantics of "*mapped by*" at the code level only (see Listing 1). This situation often results in ambiguous interpretations of the mappings. Therefore we have a need to specify the formal semantics for the "*mapped by*" relation. The personal interpretation by the developers is critical to create wrappers for the API objects. The translation of MGroup1 to XGroup1 (see Figure 2) consists in copying all data from MGroup1 attributes to XGroup1 equivalent attributes. The translation of MDevice1 is more complicated since the semantics of *mapped by* does not provide enough information for the data transfer. The developers may create XElement1 or XResource1 or both. Moreover developers are not able to infer the sequence of the translation process from the mappings specification. For instance, developers could either implement the translation of MGroup1 first or start with MDevice1.

To summarize, we observe three issues in the current specification of the translation:

- The mapping descriptions are often ambiguous.
- Developers often introduce implementation bugs while coding wrappers for such specification.
- The automatic synthesis of the translation process from the specification is not possible.

3. GLOBAL SOLUTION PROCESS

We propose a semi-automated process to limit the ambiguity of mapping descriptions, to reduce bugs in the implementation and to automate the design of the translation process. The process is composed of four steps as illustrated on Figure 3:

① **- Model Abstraction from Legacy Code:**
We analyze the legacy code of the APIs to find all relevant classes. We automatically build an application model with the use of reverse engineering techniques (see Section 4).

```
class: Mtep_Device
---------- First object to create ----------
mapped class: B_RedundantResource
read attributs:
 comment (String)⇒B_RedundantResource.name(String)
read associations:
  EMPTY
---------- Second object to create ----------
mapped class: B_NetworkElement
read attributs:
 device_id (int)  ⇒ B_NetworkElement.deviceId(int)
 type (short)     ⇒ B_NetworkElement.type(short)
 extended_type
  (short)     ⇒ B_NetworkElement.extendedType(int)
 comment (String) ⇒ B_NetworkElement.name(String)
read associations:
      ...
```

Listing 1: Example of mapping instructions provided by experts between a MTEP_Device and XMS corresponding elements: B_NetworkElement and B_RedundantResource

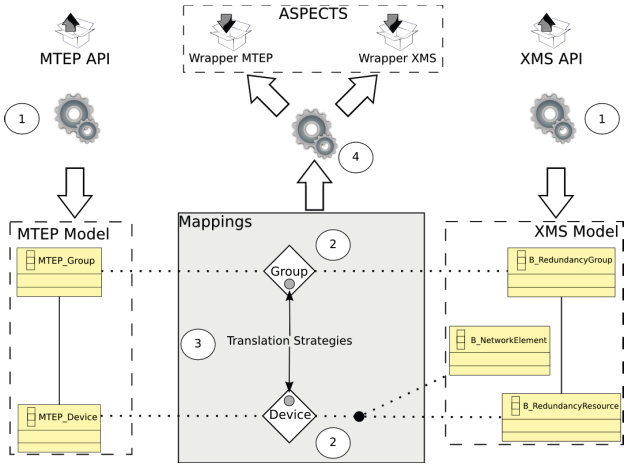

Figure 3: Our process is composed of four steps. We automatically extract models from the APIs. Users define mappings and select strategies for translating model elements. We automatically generate adapters for each API.

② - High Level Mapping Description:
Users define mappings at the model level between classes from the MTEP API and classes from the XMS API and vice versa (see Section 5). In Figure 3, white diamonds represent the mapping relations between classes from the APIs. Big black dots pinpoint that a mapping relation has multiple inputs or outputs. See Section 5 for more details on the mapping language.

③ - Translation Strategies:
Users choose strategies of translation to specify how the data should be transfered between model elements (see Section 6).

④ - Generation of Bidirectional Adapters:
We automatically generate bidirectional wrappers as aspects. We filter the information from the models and the mappings to generate code. We generate code only for the wrappers to avoid invasion of the legacy code.

4. MODEL ABSTRACTION FROM LEGACY CODE

In a context of Enterprise Application Integration (EAI), a key concern is the translation of the outputs of some applications into inputs for other applications. The problem is to efficiently translate a continuous flow of data from one API (Application Programming Interface) to another one. A possible solution could be based on an application of the Adapter Pattern [20] to map one call in API I_1 to slightly different calls that cope with API I_2, hence "*wrapping*" I_2 in such a way that it offers an interface compatible with I_1. We propose to move the mapping specification from the code level to the model level. The basic idea is to automatically build a model from the code of the API. The first step is to detect the relevant model elements that are valuable in terms of domain representativeness. We identify these model elements by analyzing the structural data (i.e OO classes) of APIs.

From this structural information, we build a model of the application. The model of the application conforms to a high-level representation of the implementation language, i.e a meta-model. We filter the model to remove any unnecessary information.

The model we build only contains domain related model elements we want to align with the model elements of another API.

We performed the reverse engineering of the API code (see Listing 2) with SpoonEMF[1]. SpoonEMF offers a Java code analyzer that automatically produces a model of the code as an Abstract Syntax Tree (AST) (see Figure 4).
We filter the AST to remove irrelevant data and build a new model that conforms to the ECore[2] formalism. We choose the ECore formalism because it is the input of the tools we use in further steps of our process.

For instance, we analyze the Java code (see Listing 2) of the MTEP API to create the corresponding application model (see Figure 4). Through the analysis, we found *device_id* and *comment* are attributes of a Java class called "Mtep_Device" and $inputFromBuffer$ is a method of the same class. Our interest focus on structural data. Therefore we keep the attributes and drop the method *inputFromBuffer*. In a second time, we look for getter and setter methods to identify read-only or read-write attributes. Third step is to create the corresponding ECore class as shown in Figure 5.

5. HIGH LEVEL MAPPING DESCRIPTION

We propose to move the mapping descriptions from an informal text-based representation to a formal specification. We adapted and extended the formalism introduced in [11] to provide users with a graphical language. Hausmann *et al*'s formalism includes elements to express *consistency between models* or in another way *the conditions under which two models are compatible*.

We kept the following definitions (see Figure 6) to express various configurations of mapping:

- A white diamond indicates a mapping definition and has a name.

[1]http://spoon.gforge.inria.fr
[2]http://www.eclipse.org/modeling/emf/?project=emf

```
package mtep.mtep_9_20.entity.dmt.impl;
import ...
public class Mtep_Device extends MtepElement
             implements MtepElementInterface
{
    private int device_id;
    private String comment;

    public Mtep_Device(){}

    public int getDevice_id(){return device_id;}

    public void setDevice_id(int aDevice_id)
    {device_id = aDevice_id;}

    public String getComment() {return comment;}

    public boolean inputFromBuffer(InputStream buffer)
             throws ExConversionProblem,ExInvalidFormat
    {return true ;}
    ...
}
```

Listing 2: Extract of MTEP Device Java class

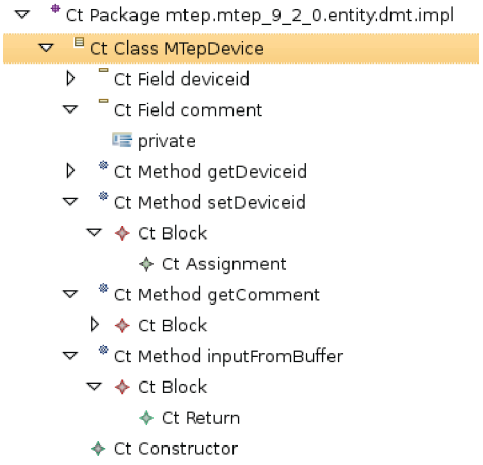

Figure 4: Application Model we got from the reverse engineering of the MTEP_Device Java class

- A model element is linked with a mapping definition with a dotted line.
- If a mapping definition involves multiple sources or multiple targets, a black circle is put between the mapping description and the set of model elements involved.

We propose to extend the current formalism (see Figure 7) to include the following concerns explicitly:

- We explicitly separated mapping descriptions into four types. Each type depends on the multiplicity of sources and targets model elements that we want to map.

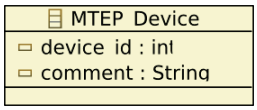

Figure 5: MTEP_Device Class in the ECore model produced from code

- We allow the creation of a mapping description between any model element such as classes (i.e real objects), attributes, and relations between model elements.
- We propose that each mapping description is bound to a strategy. A strategy is converted into a specific translation algorithm. We observed that a lot of translations share the same behavior and may be reused for several model elements translation. We conclude that users could use predefined strategies for common translation and only provide their own when necessary:
  - CloneStrategy is fully-automatic and consists in copying source attributes to target attributes with no changes
  - RenameStrategy is semi-automatic and takes data from users to align concepts by their names
  - ConstrainedStrategy is semi-automatic and provides a limited model-alignment language. The language offers atomic operations on models that guarantee the termination and the bijectiveness of the translation.
  - FreeStrategy is manual and relies on a Turing-complete language for arbitrary complex mappings.

Figure 6 illustrates the mapping language and the use of strategies on the MTEP to XMS translation example. This example is based on the mapping descriptions provided by experts to translate a device from MTEP to XMS. We create graphical mapping descriptions and link them to model elements from APIs. In the example we map a MTEP_Device to both B_NetworkElement and B_RedundancyResource and we associate a strategy of translation. The strategy of translation is contained in the mapping and is represented by a small light-grayed circle. The strategy we use in the translation is a RenameStrategy because we want to align the MTEP API model elements and the XMS API model elements on names. A RenameStrategy needs some additional input from users to automatically create a wrapper that works. We put such input in the light-grayed rectangle that is linked to the strategy. Inputs are defined with the directive language used in [9].

6. TRANSLATION STRATEGIES AND OPERATIONAL SEMANTICS DEFINITION WITH KERMETA

From this point, we manipulate one model for each API and an additional mapping model independently. We have to create wrappers to convert the API I_1 to the API I_2. It is not yet possible to automatically generate these wrappers because we need to know how to interpret the mapping language and the relation between the mapping language and the APIs model elements. We have to combine the model elements from the APIs and the mapping language elements to create a new language that describes how to execute the mappings. We give the operational semantics of this new language with Kermeta, which thus plays the role of the meta-language in which our mapping language is defined. The FreeStrategy presented in Section 5 can then be considered as an escape mechanism to the meta-language.

Kermeta [14] provides a way to compose meta-models concepts declarations using Aspects at the modeling level. This

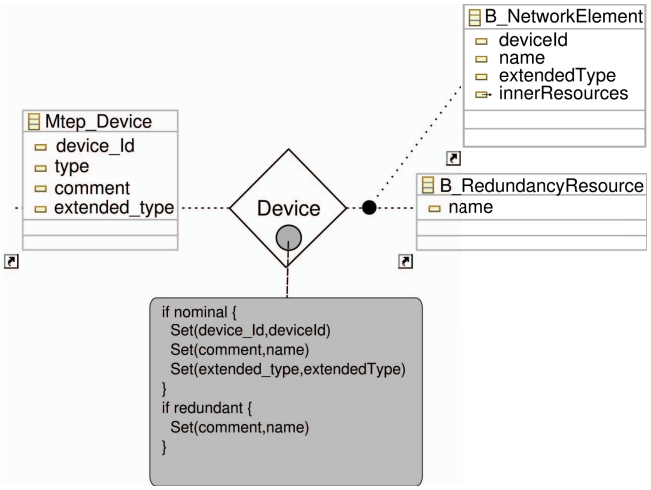

Figure 6: Graphical description of mappings between mtep and xms elements.

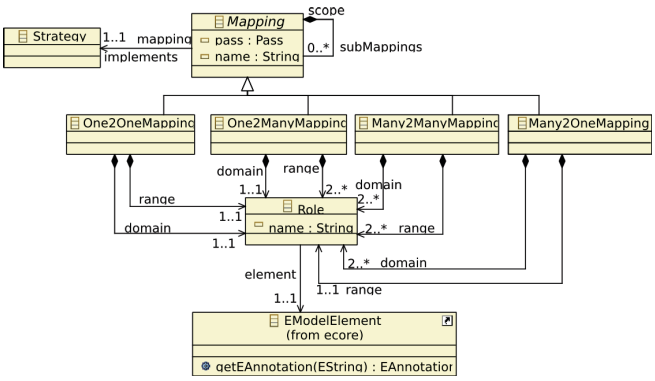

Figure 7: Graphical Mapping Language Specification as a meta-model

composition allows developers to manipulate concepts from different meta-models. The Kermeta language has also been specifically developed to express DSL operational semantics. We are then able to describe how the mapping language and its syntactic elements have to be implemented. For instance, we describe the semantics of mappings, the semantics of the relations with API model elements, and the semantics of the strategies.

To illustrate the use of Kermeta, we present the implementation of two strategies with the Thomson's case study as a background. Listing 3 show the implementation of the CloneStrategy and Listing 4 show one implementation of the FreeStrategy.

6.1 CloneStrategy Automation for Simple Mappings

The default translation strategy of our process is the CloneStrategy. This strategy needs to be fully automated to alleviate users efforts in designing the translation between APIs. We propose a Kermeta Visitor approach to create main concepts and their attributes. Listing 3 is an example of the implementation of the strategy. The implementation is achieved through two methods: "toDomain()" method literally "clone" a class from the first API to create a class from the second API; "toRange" method is the reverse.

```
aspect class CloneStrategy {
 operation toDomain() : Void is do
  var m : One2OneMapping
  m ?= self.mapping
  m.range.element.getMetaClass().ownedAttribute
    .each{ra|
     m.domain.element.getMetaClass().ownedAttribute
       .each{da|
        if da.name.equals(ra.name) then
         m.domain.element.getMetaClass().~set(
          da,m.range.element.getMetaClass().get(ra)
         )
        end
     }}
 end
 operation toRange() : Void is do
  var m : One2OneMapping
  m ?= self.mapping
  m.domain.element.getMetaClass().ownedAttribute
    .each{da|
     m.range.element.getMetaClass().ownedAttribute
       .each{ra|
        if ra.name.equals(da.name) then
         m.range.element.getMetaClass().~set(
          ra,m.range.element.getMetaClass().get(da)
         )
        end
     }}
 end
}
```

Listing 3: Example of the CloneStrategy in Kermeta language

6.2 Bidirectional FreeStrategy for Complex Mappings

Some mappings cannot be achieved by using available strategies such as CloneStrategy, RenameStrategy, or ConstrainedStrategy. Users need some complex algorithms to create corresponding model elements so they use the full expressiveness of the Kermeta language to define the translation. For simplicity's sake, we present only one type of XMS network device that is a B_NetworkElement. There exists four other types of network devices the translation has to consider. The type of a device is given by an integer in the MTEP API whereas each type of device is a different class in the XMS API. Listing 4 shows the implementation of the strategy to perform a search-and-test activity that creates various types of devices. To conclude this subsection, FreeStrategy are very useful to handle very specific translations. Moreover we are able to produce new custom strategies that can be reused in further developments.

7. BIDIRECTIONAL ADAPTERS GENERATION

The generation of the wrappers is the last step of the global process. This step is similar to well-known generative techniques that produce code from models. The code generator uses two input parameters: the mapping design model and the Kermeta code that supports the translation strategies. We adapted code generation methods to use aspects-weaving techniques, i.e, we generate wrappers as aspects to avoid the invasion of the original code of APIs. The generation step is composed of three stages (see Figure 8):

- We use the Kermeta Compiler to merge the structural definitions from the APIs with the behavioral definitions from the mappings and strategies. The output is an ECore model.

```
class FreeStrategyDevice inherits FreeStrategy {
 reference dom : Mtep_Device
 reference ran : B_NetworkElement

 operation toDomain() : Void is do
  var m : Mapping
  m ?= self.mapping
  self.dom := m.domain.element
  self.ran := m.domain.element

  if self.ran.isKindOf(B_Switcher) then
   self.dom.type := 0
  else if self.ran.isKindOf(B_TsProbe) then
    var ne : B_TsProbe
    ne ?= self.ran
    ne.tsProbeInputName := self.dom.device_Id
                           .substring(ne.tsProbeInputName.size-1,1)
                           .toInteger
    self.dom.type := 1
        ...
  end end

 operation toRange() : Void is do
  var m : One2OneMapping
  m ?= self.mapping
  self.dom := m.domain.element
  self.ran := m.domain.element

  if self.dom.type == 0 then
   self.ran := B_Switcher.new
  else if self.dom.type == 1 then
    var ne : B_TsProbe init B_TsProbe.new
    ne.tsProbeInputName := "TSProbe_" +
       self.dom.device_Id.toString
    self.ran := ne
   ...
 end end
}
```

Listing 4: Example of FreeStrategy for translating a MTEP_Device to the right type of B_NetworkElement and vice versa

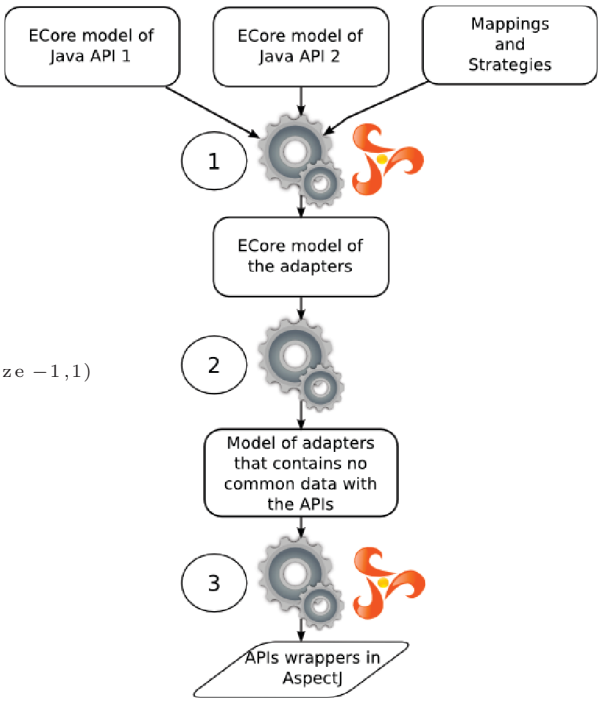

Figure 8: The production of executable code is composed of three steps. The Kermeta Compiler merges structural and behavioral definitions. We filter the resulting model. We generate the wrappers between APIs as aspects.

- We filter the ECore model to remove all data that are in conflict with existing APIs. We obtain an ECore model that only contains the definitions related to the translation.
- We use an AspectJ generator written in Kermeta to generate aspects from the ECore model we filtered.

7.1 Merging Structure, Mappings and Strategies

We convert the mapping specifications into a Kermeta model. Each mapping is converted into two Kermeta aspects: one for the source model elements of the API I_1 and one for the target model elements of the API I_2. These aspects contain Kermeta operations that encapsulate the adaptation between the two APIs. We analyze strategies and additional alignment directives provided by users and translate them into Kermeta code. This code is the actual body of the methods previously created.

As an example, we consider the conversion of a MTEP_Device to a B_NetworkElement. We create two Kermeta aspects for both MTEP_Device and B_NetworkElement. We analyze the strategy associated to this mapping and produce corresponding Kermeta code (see Listing 4). We combine the definitions of the *device_id* and *comments* attributes (see Figure 5) with this additional behavior (*toDomain*() and *toRange*() operations) to produce the final adapter. Result is shown on Listing 5.

7.2 Avoid Invasion by Filtering

One key concern of our approach is to be non-invasive regarding legacy API code. The merged model we get from previous stage contains definitions of both API classes and wrappers. The generation of code from this model would build a set of new classes that would overwrite legacy classes. We avoid classes to be overwritten by removing class definitions that are equivalent between the models of the APIs and the model of the wrappers. We apply a filtering method that only keeps new class members or additional utility classes.

The method checks names of classes and their signature [9] to identify equivalent class definitions that we remove from the merged model. We get a new ECore model that only contains the methods and classes that define the wrappers between one API and another. In the example of the

```
aspect class MTEP_Device {
 /* Attributes from MTEP_Device class */
 attribute device_id : int
 attribute comment : String

 /* Kermeta code from Listing 4 */
 operation toDomain() : Void is do
  ...
 end
 operation toRange() : Void is do
  ...
 end
}
```

Listing 5: Aspect in Kermeta: combination of legacy data and adapters data

```
//classes exist in Legacy
classesinMM1 ?= generator.allClasses
//All classes
classesinCompiledEcore ?= generator1.allClasses
var classExistInLegacy : Sequence<EClass>
classExistInLegacy := classesinCompiledEcore.select{
 c| classesinMM1.exists{c1 | c1.name == c.name}
}
classExistInLegacy.each{c |
 var c1 : EClass init classesinMM1.detect{c2 |
  c2.name == c.name}
 /* Operations that should be introduced in a legacy
  * classes (operation that are in the class of the
  * compiled ecore but not in the original ecore */
 var operationToIntroduce : Sequence<EOperation> init
 //match by signature
 c.eOperations.select{op | not c1.eOperations.exists{
  op1 | op1.name == op.name}}
 /*Fields that should be introduced in a legacy
  *classes (fields that are in the class of the
  *compiled ecore but not in the original ecore*/
 var fieldsToIntroduce :
   Sequence<EStructuralFeature> init
    c.eStructuralFeatures.select{f |
     not c1.eStructuralFeatures.exists{f1 |
      f1.name == f.name}}
 //Generate aspects
 generateAspect(c,fieldsToIntroduce,
  operationToIntroduce, outputFolder)
}
 //newclasses => Standard POJO Generation
 classExistInLegacy.each{c |
  classesinCompiledEcore.remove(c)}
 classesinCompiledEcore.each{c|
  generatePojos(c,outputFolder)}
```

Listing 6: Kermeta code for filtering models elements that are in conflict with existing one in legacy APIs

MTEP_Device, the filtering process removes elements that exist in legacy APIs. As a consequence, *device_id* and *comment* attributes are removed from the definition of the MTEP_Device aspect.

7.3 Executable Code Production

We process the model of aspect definitions we got from the previous stage. The Kermeta compiler uses code templates to produce AspectJ executable code. Classes that do not exist in the legacy code are created whereas classes that already exists are augmented with inter-type declarations. The inter-type declarations encapsulate the translation behavior between existing classes. Adapters between the two APIs (see an example for a MTEP_Device in Listing 7) are composed with the original legacy code (available as a JAR) at load-time using the AspectJ compiler. Load-time weaving is deferred until the point that a class loader loads a class file and defines the class to the JVM. As a consequence, additional capabilities we brought through the adapters production does not pollute existing code embedded in legacy APIs.

8. DISCUSSION

As an evaluation of our solution, we propose to compare efforts between classic development techniques (followed by domain experts) and using our semi-automated process. As a benchmark, we are comparing our solution to Thomson Extensible Management System evolution on the specific example of MTEP to XMS protocol translations.

8.1 Impact of Automation on Wrappers Production

```
package net.thomson.protocol.mtep;
aspect Mtep_DeviceAspect {
 declare parents: Mtep_Device
   implements kermeta.language.structure.Object;

 public Mtep_Device_Input inputsLinkedWithBoards;
 public Mtep_Device_Output outputsLinkedWithBoards;
 public B_NetworkElement output;
 public B_RedundancyReplaceableResource outputRedundant;
 public void  mtep2xmsPass1(){
   ...
 }
 public void  mtep2xmsPass2(B_XmsBusinessData xbd,
    Mtep_Device.Object out, JavaBoolean nominal){
    ...
 }
}
```

Listing 7: AspectJ code produced by the Kermeta compiler for the MTEP_Device example

The case study is based on a subset of Thomson MTEP to XMS conversion. This subset contains seven MTEP-related concepts and twenty XMS-related concepts defined in their respective APIs.

From the correspondence specifications provided by experts, Thomson developers implemented twenty mappings to carry out the bijective translations between MTEP and XMS (see Figure 9 for details about mappings ratio). The application of our process on the same case study involves only eight bidirectional mappings, whose distribution is represented in Figure 10.

We can draw two conclusions from these figures:

1. We needed fewer mapping descriptions to handle the same case study.

2. Mapping descriptions complexity is reduced since Many-to-many mappings are not used and 75% of mapping descriptions are One-to-one mappings.

The first point comes from the use of a more adequate DSL to express mappings at the right level of abstraction. At the code level, developers implement complex mappings as one-to-one mappings because the implementation language do not offer higher-order mapping operators. The mapping language we propose offers more expressiveness to declare mappings so some of them, identified by experts, are not expressed anymore as single mappings but are encapsulated into higher-order mappings. The second point is related to the expressiveness of the DSL versus the implementation practices in Java. We observed that when people use low-level correspondence languages, they are more tempted to violate implementation practices of the Visitor Pattern to access incidental information (information from multiple concepts that do not take part in the original described mappings). Our approach limits this problem since the mapping DSL offers higher abstractions to retrieve data in a proper way: users are able to describe relations between mappings, so concepts involved in a mapping are confined to the original inputs and outputs of the wrappers.

Figure 11 is to be compared with Thomson implementation of adapters (100% of strategies would be FreeAlignment). This figure shows that most of strategies (63%) used to handle the case study are automatic or semi-automatic. Of course, it is not possible to automatically define all mapping implementations: that is why we provide a way to use

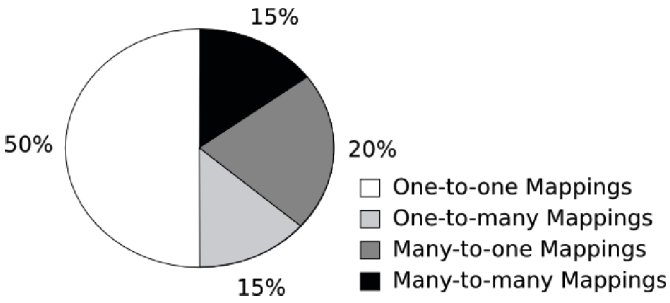

Figure 9: Distribution of the twenty mappings identified by the experts of the domain and implemented with Java

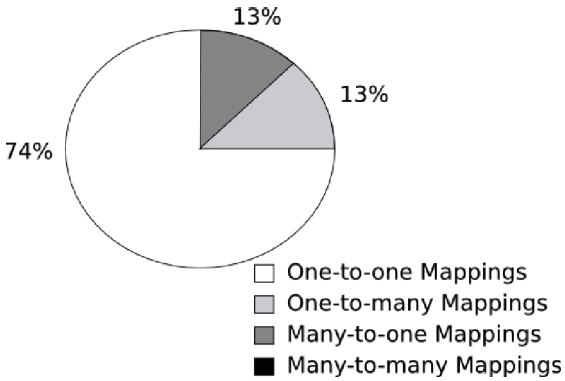

Figure 10: Distribution of the eight mappings identified by experts of the domain and implemented with the mapping language presented in Section 5)

a more powerful language to implement the remaining mappings.

These results are a first indication, on a relatively small example, that the use of a high-level language for mapping descriptions helps reduce the number of adapters to be implemented. It also gives additional evidence that the use of generative techniques cuts down global complexity and effort to produce adapters.

8.2 Effort Comparison

The second stage of our evaluation deals with effort estimation in terms of the number of lines of code (LOC). Thomson global adapter size for the case study is about 5350 LOCs to realize the bi-directional translation between MTEP and XMS protocols. Our approach is implemented using only 510 Kermeta LOCs. The effort has been evaluated to 136 hours of person work for the manual implementation of a

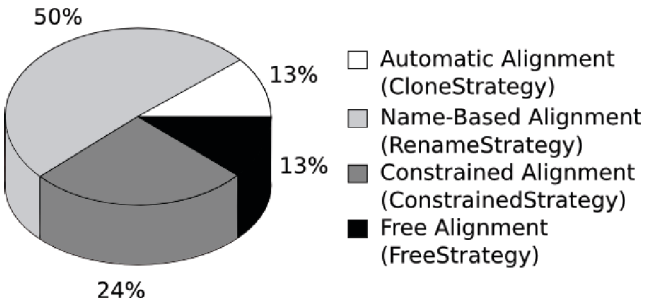

Figure 11: Ratio of strategy types used to implement adapters through the process we propose

Production of a new adapter	Manual Approach		Generative Approach	
	v1	v2 (avg)	v1	v2 (avg)
Code length (LOC)	5350	-	570	-
Total TDEV (Hours)	136	**+57**	150-200	**+9**

Table 1: Effort for manual and generative approaches for the production of a new adapter. The production of a second version (v2) increases the effort by an average of 57 hours for the manual approach and by an average of 9 hours for the generative approach.

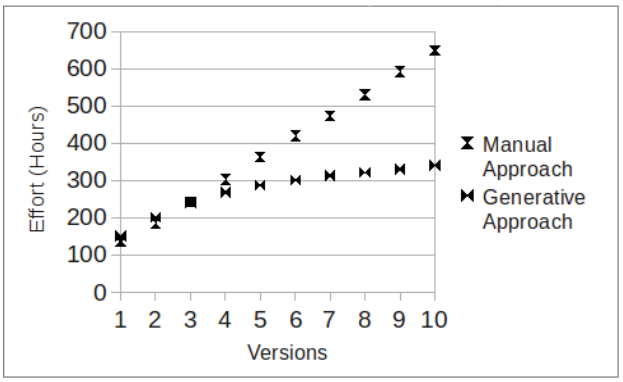

Figure 12: Cumulative effort for the production of new versions of adapters using manual or generative approach: effort (time of development in month) is on the y-axis and versions on the x-axis.

new adapter, compared to 150 hours to handle the same example with our approach. Considering up to 50 extra hours to take into account the introduction of a new mapping language and a new language for strategies definition for users, the effort needed to use our process is of 150 up to 200 hours (see Table 1) for the very first version of an adapter, which is slightly more than the manual approach.

However, the mean of the effort to produce a new version of an adapter for both approaches are 57 hours for a manual implementation versus 9 hours with the semi-automatic process.

These results have several consequences: First, we are able to say that our approach needs less manual implementation from users. Second, thanks to generative techniques, we were able to reduce the number of bugs in code and thus time spent in debugging has been drastically reduced. These improvements allow users to save some maintenance effort on the code in further evolutions. Figure 12 illustrates the effort reduction in the production of ten successive versions of this adapter. A manual approach induces a constant effort to develop and test new adapters versions. Our semi-automatic approach is expensive on the very first version (learning overhead and complex mapping definitions) but costs decrease with time as learning overhead decreases in further evolution. Even though benefits, in terms of efforts, observed on Figure 12 are relatively small, we have to keep in mind that this process is to be repeated, for instance, on the five APIs definition presented on Figure 1 with, let us say 10 versions each. Since we want these API to be integrated with each other, it ends in the production of $(5 * 10)^4$ adapters. A potential extrapolation of our results would give an effort reduction of 87% by using our approach.

8.3 Runtime Overhead

The adapter generation process does not raise much runtime overhead for the translation execution. Indeed, the generated AspectJ only contains inter-type declarations to introduce new attributes and methods used in the visitor. The weaving between the aspects and the legacy code is performed at load-time using AspectJ 5 that does not introduce significant overhead. Still the Java behavior code generated from the Kermeta compiler could be improved, in particular the compilation of the lexical closure used in Kermeta in the free mapping are rather naive. Nevertheless, in studying existing translations in the legacy code developed in Java, we have also highlighted that some basic Object-Oriented principles are often violated. For example, the bad use of the visitor pattern introduces the use of lots of Runtime Type Information tests which also decreases the efficiency of the translator. For these three reasons, there is no significant waste or saving of performance during the translation execution, in the considered case study.

9. RELATED WORK

Software evolution and maintenance of legacy systems is a constant challenge for software engineers. Most research works focus on evolution and maintenance of one given legacy system. Various techniques such as Design Patterns [4] [3], and more recently models and program transformations [24, 21] have been used for this purpose. While these works aim at modifying the legacy for technical upgrade or even migration, we aim at connecting several existing legacies through the alignment of their API. We propose a Domain Specific Language to generate an adapter pattern for a set of classes. In this context, the constraints we consider are different from legacy evolution since the existing code must be kept unchanged. The process we propose for that is related to two main domains: model-driven engineering, for specifying the mapping at a high-level of abstraction; aspect-oriented programming for integrating the wrappers with the legacy code in a non-intrusive way.

The model-driven engineering community has developed several languages to specify transformations based on mappings between concepts such as QVT [5], ATL/AMW [8] or graph-based languages such as Viatra [22], VMTS [23] or AGG [1]. These approaches are oriented to meta-models correspondence and to transformation rules production in order to perform data transformation or migration. In comparison, the process we propose is specific to APIs translation and inter-operation. Our graphical representation of mappings is based on ideas taken from the work of Hausmann [11]. We kept the graphical representation of mappings and we use some concepts to include mappings within each other. However, as mentioned in previous section 5, we did not keep simple relations between mappings because these relations introduce more complexity in the mapping description activity that users could have difficulties to manage.

Even if several papers discuss the issue of evolving aspect-oriented applications [15], several works have shown the relevance of adopting Aspect-Oriented paradigm to work with legacy systems. Belapurkar [6] use aspect-oriented programming to comprehend and maintain complex legacy systems. It shows how AspectJ can be used to perform static or dynamic analysis of legacy code to evaluate the impact of an interface change, identify dead code, generate a dynamic call graph or evaluate the impact of exceptions. Ng *et al.* [16], discuss how simple yet effective AOP constructs can facilitate the process of program comprehension. Contrary to these work, we use AOP as a composition operator to enrich existing classes in the legacy without modifying their code.

Hannemann *et al.* have illustrated the relevance of AspectJ to implement GoF design patterns [10]. They demonstrate modularity improvements in 17 of 23 patterns. In our approach, we use AspectJ to integrate new methods and attributes that are necessary to implement the visitor pattern in existing legacy code. We extend the interface of the *Element* class via AspectJ's open class mechanism as suggested in [10]. Moreover we differ from the previous approach because we generate the AspectJ code from a domain specific language that declares the mapping between two APIs.

Currently, we use Inter-Type Definition of AspectJ mainly for modifying existing API without modifying their code. We could obtain the same results with other approaches. Scala [17] for example proposes a mixin-class composition that can be used to introduce new member definitions of a class. In association with its "implicit" mechanism that allows extension of existing classes through a lexically scoped implicit conversion, Scala can replace Java and AspectJ [19] as a back-end for generation. Another similar approach for the introduction is the use of Composition Filters [2] where filters are aspects that act as proxies to messages and impose additional functionalities. These functionalities are activated based on conditions specified in the filters.

In this trend of AspectJ generation, Meta-AspectJ [12] and XAspects [18], which advocate the use of AspectJ as a back-end language for aspect orientation, are similar with our approach. Nevertheless, the main goal is different. The value of Meta-AJ or XAspects is found in the reasons why neither AspectJ nor code-generation alone is sufficient. Meta-AJ provides syntactic features to deal with the dynamic creation of aspects, such as wildcard characters in names specifications. Using Meta-AJ, a user can leverage the full power of Java to create complex joinpoints that AspectJ does not support. We use AspectJ with load-time weaving to bypass the absence of open-classes [7] support in the legacy code.

10. CONCLUSION AND PERSPECTIVES

We have presented a process to semi-automatically produce adapters from existing APIs. Our approach is based on pooling existing techniques together (reverse-engineering, model-to-model transformation, code generation and aspect weaving) to alleviate tedious and error-prone developments. Although the specific domain of APIs interoperability looked first quite unsuitable for applying techniques such as Domain Specific Modeling (DSM) or Model-Driven Engineering (MDE), we successfully achieved to provide another way of producing interoperability adapters by combining model-related and code generation techniques.

Our process has been successfully applied on the example presented in Section 2, paving the way for large scale deployment on Thomson Extensible Management System for Digital TV.

This idea of smoothly combining models, aspects and generative techniques with legacy code goes actually well beyond the problem of API adaptation. Since very few developments start from scratch nowadays, the idea of using MDE to generate all the code of an application looks quite naive, and thus rightfully induces some skepticism in programmer's minds. A key issue of extending the usage of MDE beyond specialized domains then lies in its ability to

smoothly blend with legacy applications or components [13] as illustrated here. We hope that the approach presented in this paper could constitute a first step into reconciling modeling level approaches with programming level ones.

Acknowledgments

We would like to thank Nicolas Christian from Thomson Green Valley for providing us with details about the MTEP and XMS API specifications and for passing on his experience regarding former manual practices to develop adapters.

11. REFERENCES

[1] AGG. The attributed graph grammar (agg) system. http://user.cs.tu-berlin.de/ gragra/agg/.

[2] M. Aksit and B. Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters, 1998.

[3] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 265–279, New York, NY, USA, 2005. ACM.

[4] S. Barkataki, S. Harte, and T. Dinh. Reengineering a legacy system using design patterns and ada-95 object-oriented features. In SIGAda '98: Proceedings of the 1998 annual ACM SIGAda international conference on Ada, pages 148–152, New York, NY, USA, 1998. ACM.

[5] W. Bast, M. Belaunde, X. Blanc, K. Duddy, C. Griffin, S. Helsen, M. Lawley, M. Murphree, S. Reddy, S. Sendall, J. Steel, L. Tratt, R. Venkatesh, and D. Vojtisek. Mof qvt final adopted specification. OMG document ptc/05-11-01, October 2005 http://www.omg.org/docs/ptc/05-11-01.pdf.

[6] A. Belapurkar. Use aop to maintain legacy java applications. IBM developer work, (March 2004) http://www.ibm.com/developerworks/java/library/j-aopsc2.html.

[7] C. Clifton and G. T. Leavens. Multijava: Modular open classes and symmetric multiple dispatch for java. In In OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 130–145, 2000.

[8] M. D. Del Fabro and P. Valduriez. Semi-automatic model integration using matching transformations and weaving models. In SAC '07: Proceedings of the 2007 ACM symposium on Applied computing, pages 963–970, New York, NY, USA, 2007. ACM.

[9] F. Fleurey, B. Baudry, R. France, and S. Ghosh. A generic approach for automatic model composition. In Models in Software Engineering: Workshops and Symposia At MoDELS 2007, Nashville, Tn, Usa, pages 7–15. Reports and Revised Selected Papers, Septembre-October 2007.

[10] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 161–173, New York, NY, USA, 2002. ACM.

[11] J. H. Hausmann and S. Kent. Visualizing model mappings in uml. In SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization, pages 169–178, New York, NY, USA, 2003. ACM.

[12] S. S. Huang, D. Zook, and Y. Smaragdakis. Domain-specific languages and program generation with meta-aspectj. ACM Trans. Softw. Eng. Methodol., 18(2):1–32, 2008.

[13] A. Misra, J. Sztipanovits, G. Karsai, M. Moore, A. Ledeczi, and E. Long. Model-integrated computing and integration of globally distributed manufacturing enterprises: Issues and challenges. Engineering of Computer-Based Systems, IEEE International Conference on the, 0:225, 1999.

[14] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In S. K. L. Briand, editor, Proceedings of MODELS/UML'2005, volume 3713 of LNCS, pages 264–278, Montego Bay, Jamaica, Oct. 2005. Springer.

[15] F. Munoz, B. Baudry, and O. Barais. Improving maintenance in aop through an interaction specification framework. In the proceedings of the 24th International conference on Software Maintentance, ICSM08, 2008.

[16] D. Ng, D. Kaeli, S. Kojarski, and D. Lorenz. Program comprehension using aspects. IEE Seminar Digests, 2004(902):89–96, 2004.

[17] M. Odersky, V. Cremet, I. Dragos, G. Dubochet, B. Emir, P. Haller, S. Mcdirmid, S. Micheloud, N. Mihaylov, L. Spoon, and M. Zenger. A Tour of the Scala Programming Language, May 2007.

[18] M. Shonle, K. Lieberherr, and A. Shah. Xaspects: an extensible system for domain-specific aspect languages. In OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 28–37, New York, NY, USA, 2003. ACM.

[19] D. Spiewak and T. Zhao. Method proxy-based aop in scala. JOT: Journal of Object Technology, 8(7), 2009.

[20] S. A. Stelting and O. M.-V. Leeuwen. Applied Java Patterns. Prentice Hall Professional Technical Reference, 2001.

[21] W. Tansey and E. Tilevich. Annotation refactoring: inferring upgrade transformations for legacy applications. In OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, pages 295–312, New York, NY, USA, 2008. ACM.

[22] VIATRA. Viatra2. http://www.eclipse.org/gmt/VIATRA2/.

[23] VMTS. Visual modeling and transformation system (vmts). http://vmts.aut.bme.hu/.

[24] J. Zhang. Supporting software evolution through model-driven program transformation. In OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pages 310–311, New York, NY, USA, 2004. ACM.