Toward automated model transformations in the A-MUSE design methodology

Steven Bosems
s.bosems@cs.utwente.nl

ABSTRACT
The usage of automated model transformations has proven to be useful in the model-driven software design methodology. In this paper, we will present a set of QVT Relations rules to perform automated transformations in order to obtain platform independent service design models from service specifications. This is done in the context of the Freeband A-MUSE project, which focuses on developing an MDA based approach to context-aware software design. We will find that QVT may be utilized successfully within the A-MUSE methodology.

Keywords
A-MUSE, context-awareness, decomposition, Model-driven Architecture, model refinement, Query/View/Transformations

1. INTRODUCTION
The ever growing demand for high quality software systems forces researches to develop new software development methodologies. They promise the best available IT solutions, against the lowest price, in the least amount of time. To achieve these project goals, new tools will have to be developed to aid their users to utilize their time optimally.

One specific class of applications for which no clear development methodology has been developed, are context-aware applications. The Freeband A-MUSE project aims to specify such a methodology. It is based on the model-driven architecture and utilizes models at different abstraction levels. In order to maximize user productivity, we are going to specify rules that will allow for automatically creating detailed diagrams from general input diagrams.

This paper is structured as follows: sections 1.1 through 1.4 describe the basic concepts of the model-driven architecture, the Query/View/Transformations language, context-awareness and the A-MUSE methodology respectively. Section 2 explains the example we will be using in our research, section 3 presents related work, and section 4 provides a detailed problem statement. In section 5 we describe our research approach and we discuss the results in section 6. Section 7 will provide a conclusion and recommendations for future work.

In this paper, we will be using a number of abbreviations and domain specific terminology. In order to improve the readability of this paper, we have compiled a list of definitions. Along with a short description, we have included the first occurrence of the defined item.

List of definitions
ASDL The A-MUSE System Description Language. Models that depict the service specifications in the A-MUSE project are structured using ASDL. First occurrence: section 6.7
Freeband A-MUSE A project that aims to provide end users with the tools to specify their own context-aware software applications. First occurrence: section 1.4
ISDL The Interaction System Design Language. Models that describe the platform-independent service description within the A-MUSE project are structured using ISDL. First occurrence: section 6.7
MDA The Model-driven Architecture is a model oriented software design methodology. First occurrence: section 1.1
PSM Platform-specific models depict how software systems are to be implemented on a certain hardware or software platform. First occurrence: section 1.1
QVT The Query/View/Transformations language allows developers to create automated model-to-model transformations. First occurrence: section 1.2

1.1 Model-driven architecture
New technologies and programming concepts are emerging faster than ever. The problem programmers are facing is that legacy systems are not always easy to integrate within new architectures and replacing them is costly. In order to prevent these problems in future system architectures, the Object Management Group (OMG) designed the model-driven architecture (MDA) methodology.

MDA allows software architects to model software systems and middleware platforms in a platform-, vendor-, and language independent way. At the core, it consists of one of the OMG modeling standards: Unified Modeling Language (UML), the Meta-Object Facility (MOF), and Common Warehouse Meta-model (CWM). These core models are platform- and language independent; they model the essence of the system. Even if the underlying implementation techniques change, the core models will continue to be used. These models are called platform independent models (PIMs).

By using mapping tools, application programmers can convert PIMs into models for specific platforms (Java, XMI/XML,
QVT Core defines three QVT languages. The QVT specification way to perform model-to-model transformations, given the meta- the system has to be equipped with the appropriate sen- and react to changes of the environment. To allow for this be- We define 1.3 Context-awareness We define context-awareness as the ability of a system to sense and react to changes of the environment. To allow for this beha- Some examples: a patient suffering from occasional epileptic seizures is given a mobile device running a context-aware application. The application continuously monitors the patients condition, the current location of the patient, and whether certain utility services are available. If the patient suffers from a seizure, the application will detect this event and will react by sending a message through Short Message Service or SMS to a medical team that is in the vicinity of the patient. In order to do this, an SMS service is used. The message may include the location of the patient, his or her condition and the severity, thus allowing the medical staff to judge what action to take. This example is derived from [3]. Another example: a public transport passenger is waiting at the station for his train to arrive; however, an intercom announce- ment informs him that his train has been delayed. By using a context-aware travel planning application, he will get updated in- formation about his trip. This may include which platform his next train will depart from, the time of departure and the new es- timated time of arrival. All this information was generated with- out additional user input and utilizes the location of the current railroad station and a real time train schedule. Context-awareness enhances the way a user experiences an appli- cation: instead of being "dumb", the application is able to make intelligent decisions based on changes in the user situation, al- lowing it to be more helpful to the user without needing constant user interaction to provide data. 1.4 A-MUSE Currently, communication services are provided by telecommu- nications companies [3]; they fully control the way users and ap- plications work with these services. In order to allow users to create their own services, new technologies, methods, and tools have to be developed. These tools should allow the user to specify their service at a high level and largely automatically transform the service design into platform-specific implementations. The Freeband A-MUSE project [5] aims to define an MDA based software design methodology that will allow users to specify and realize their own services. It specifies three levels of models, each one with a higher level of detail and platform dependence [6]. Firstly, at the highest level of abstraction, we have the service specification (SS); models at this level describe the behaviour of the service from an external point of view. They describe the input and output of the system, but not the internal workings of it. Secondly, at the intermediate level, the platform independent ser- vice design (PISD) may be found. These models describe the service behaviour from an internal point of view; they illustrate the actual internal working of the application, in order to provide the inputs and outputs defined in the SS. Finally, at the lowest level, the platform specific models (PSMs) are defined. Models at this level describe context-aware services in terms of specific target technologies: web services, JAVA/RMI, .NET, J2ME, CORBA, and other implementation techniques. 2. EXAMPLE In the Freeband A-MUSE project, the Live Contacts case [7] is used to clarify and demonstrate new ideas and technologies. We
will use a subset of this case in our research. The focus of this paper will be on the requestBuddyList, removeBuddy, and the contactBuddy operations.

The Live Contacts case describes a context-aware mobile service that runs on Personal Digital Assistants, smartphones, and desktop computers. It enables users to maintain a list of contacts, accompanied by data that will allow the user to contact them in different ways: e-mail addresses, mobile phone numbers, working phone numbers and instant messaging information is stored.

By using the requestBuddyList, a user may request their buddy list in order to get a full view of the list of contacts they have. Additionally, a user might want to add or remove buddies from their list. If the removeBuddy operation is issued, the service will perform a check to ensure the buddy that is to be removed exists in the users contact list. Depending on the result of the check, the buddy will either be removed or the system issues an error to the user, informing him that the requested buddy is not in their list.

An important part of the Live Contacts case, is the way a user may contact their buddies. The system keeps track of the communications services available, and allows the user to initiate communications by the services that can be contacted. For example: a user utilizing a desktop computer that has internet connection will be allowed to send e-mails and chat with their buddies, but calling them using a the stored phone number or sending an SMS will not be available. If that same user switches to a smartphone, all services will be available; we assume that a modern smartphone has the capabilities to initiate chat sessions and send e-mails. For this feature, the contactBuddy operation is used.

### 3. RELATED WORK

After the publication of [1], numerous sources have described the usage of MDA in the software development methodology. However, MDA is most commonly used in “normal” application development: the developed programs are not context-aware. Context-awareness provides an additional layer of complexity to the development process; applications have to be developed in order to use data received from the application environment and should contain knowledge about the desired reactions to changes in the application environment.

Multiple papers have been written to explore the usage of MDA when developing context-aware applications [3, 4, 6, 8, 9, 10, 11, 12], some of which in the context of the A-MUSE project. Many sources mention model transformation in order to accelerate the software development process, but only [10] aims to automate it; it utilizes meta-models for input- and output models and a transformation language to define mapping constraints between these two. However, the transformation tool used in this paper is GREAT and the meta-models are defined in MetaGME, both are proprietary standards. The author notes that this is a drawback and accessible standards like MOF would be preferred.

A tool that does support the usage of open models, is mediniQVT [13]. This tool is an implementation of the QVT Relations standard; it accepts meta-models made in ecore, the meta-model type used by the Eclipse Platform [14], and allows the user to transform models written in XMI, which is used to represent meta-data. This information taken into account, it can be seen that mediniQVT may be a suitable candidate tool to be incorporated in the A-MUSE methodology. The abstract structure of a transformation process is described in Figure 1 when given the meta-models for ASDL, meta-data that is structured according to this model, and the ISDL meta-model, we may utilize QVT rules that combine information from these meta-files to generate meta-data according to the ISDL model. This model may be a decomposition of the source model.

Currently, the Grizzle tool [15] is most commonly used in A-MUSE. This tool allows the user to define ISDL models and export them to, among others, XMI files. These files may then be used by external applications. Since the QVT rules work on meta-data files that are structured in XMI, it becomes apparent that Grizzle can be utilized to perform the creation of the input models for QVT rules.

As [1] teaches us, the usage of transformation automation tools is desired in MDA. Hence, it is logical that [2] tells us how QVT may be incorporated in such design trajectories, as do other technical reports and papers. Yet, little is written about the usage of QVT to describe model transformation constraints regarding models that specify context-aware software systems, let alone context-aware behaviour. Papers that do describe this specific set of applications, often note that automated transformation is possible, but provide no actual implementation. This is what we want to achieve in this paper.

[16] provides a classification of the functionality offered to the user at the SS level. This is done by specifying six classification rules that identify interaction types: simple receives user input and eventually presents output to the user, search retrieves externally stored information, update changes information stored in an external source, context retrieves information from the user context, invocation invokes an external service and discovery discovers an external service through means described by the SOA paradigm. These interaction types may also be combined to form more complex interactions.

### 4. PROBLEM STATEMENT

In this paper, we will be evaluating whether the QVT Relations language is suitable to specify model transformation rules for the automated refinement of context-aware application models.

The context of this paper is the Freeband A-MUSE software development trajectory. We will be focusing on the transformation of service specification level models into more detailed platform
5. APPROACH
As said in section 4, we will be focusing on finding a way to make service specifications more specific by using a given set of rules. This will result in a more detailed platform independent service design; given an input diagram at the service specification level, we aim to automatically generate a decomposed output model. Our input model defines user interactions, our output model depicts the structure of the system that will allow for these user inputs and outputs.

In order to successfully transform an SS into a PISD, we have to take several steps. We will explain these in the following sections.

5.1 Meta-model
If we want to transform a source model into a destination model, we have to know how these models are structured. When the inner configuration of a model is analyzed, we can draw a hierarchical diagram of the appearance of a diagram. If we do this for both the source and the destination model, we may see a resemblance between the two.

5.2 Transformation rules
Since we want to make the input model more specific, the first step we have to take in defining the QVT transformation rules, is a simple one-on-one copy transformation: the output of the transformation will be identical to the input. We may then start defining pattern matching rules. These rules will find distinct patterns in the input diagrams and output data accordingly: when a given pattern is found, the output will be a more specific version of this pattern. In Figure 2, Figure 3, and Figure 4, we have defined these patterns according to [16]. However, since some of the patterns that occur in the classification rules that are described in [16] are of a theoretical nature, they do not appear explicitly in the models. We will be matching a subset of the classification rules: the simple + search, the simple + update, and the discovery + invocation patterns.

5.3 Models
In order to check whether the QVT rules are correct, we will have to formulate an input model. [16] defines models for the Live Contacts case. We will use these models in our research. In section 2 we have described the context in which the Live Contacts application is used.

5.4 Transformation
Once the input and output meta-models and the input model have been defined, we can use the QVT transformation rules to generate the desired output model. This output model will be a PISD diagram that has more detail than the SS diagram that was the input of the transformation rules.

5.5 Correctness
If all transformations succeed, the correctness of the QVT rules has to be verified. This can only be done by manually checking the output model and comparing it to the reference output model, as described by [16]. If the output model is correct, the transformation rules are correct for that case. They will then be usable to perform transformations on additional input models that are structured in the same manner as the original models.

6. RESULTS
6.1 Meta-model
The A-MUSE guidelines [8] inform us that services are defined using A-MUSE System Description Language (ASDL) and the platform independent service description is done in the Interaction System Design Language (ISDL). However, when we look at Grizzle, we find that ASDL is a profile for ISDL: it provides a set of rules and constraints to ISDL, it does not add extra functionality. Grizzle even outputs ISDL models when the A-MUSE profile is selected. The model transformations that are done in this research will thus have the same input and output meta-model. The ISDL meta-model is available online [17].

In order to use the ISDL meta-model for transformation rules, its structure has to be evaluated. Since no official documentation is available for the model syntax, we chose to reverse-engineer the model; this is done by creating small example models in Grizzle and exporting them to XMI, the format the transformation rules will be working on. After creating small examples, containing the minimum amount of information need, we have enough information to start writing basic transformation rules.

6.2 Transformation rules

6.2.1 Mapping
The transformation rules are written in two phases. In the first phase, we construct a basic transformation that will copy the input model directly to the output model: only mapping is done, no
actual transformations are performed. An example may be found in QVT.

QVT 1: RepositoryMapping
1 transformation ss2pisd(ss : emf, pisd : emf) {
2 top relation RepositoryMapping {
3 varName : String;
4 5 checkonly domain ss ssRep : Repository {
6 name = varName
7 );
8 9 enforce domain pisd pisdRep : Repository {
10 name = varName
11 );
12 }
13 }

The syntax of QVT is quite similar and yet different from other programming languages. We will be using Java as a reference. QVT files are structured like Java Class files: where Java uses public class <name> {...}, the QVT syntax is transformation <name> (<from_model_name> : <from_model>,
<to_model_name> : <to_model>){...}

When looking at our example in QVT we see that the transformation name is ss2pisd (Service Specification to Platform Independent Service Design), the input model uses the emf meta-model, the top-level name of the ISDL meta-model, and is named ss. The same is holds for the output model.

Line 2 indicates that we are ready to start a transformation. The keyword top indicates that the relation must succeed, otherwise the transformation as a whole will fail. If we compare the syntax to Java, we find that Java uses the keyword void to indicate we are performing a calculation that has no output; in QVT we use relation. However, there is a difference: Java methods are performed when called explicitly, yet, in QVT, all relations are performed implicitly if the contents of the input model matches the pattern described in the relation.

As in Java, variables may be used in QVT. These are defined as early as possible in a relation. In the example, a variable varName is defined in line number 3; varName is of the type String.

Line number 5 is the introduction to our first pattern: it specifies that in the domain of ss, the name defined in line 1, we try to find a Repository which we will be referring to as ssRep. The keyword checkonly indicates a checking relation, as described in section 6.2.2. In line 6, we tell our relation that the property name of the matched Repository has to be equal to varName. In other words: the value of varName will become equal to that of the name property. Line 7 closes the first pattern.

Lines 9 through 11 are similar to lines 5 through 7, yet there are some differences: the first one is that, instead of the keyword checkonly, enforce is used. Again, this term is explained in section 6.2.2. Line 10 appears to be similar to line 6, however, the meaning is quite different. Since we are working in an enforced pattern, the Repository we are matching will probably not exist and will have to be instantiated. The name of this Repository will thus initially be empty. Line 10 ensures that the value of the name property of the new Repository will be identical to the value of the varName variable. When combining lines 6 and 10, we see we have defined a transitive relation: varName = ssRep.name, pisdRep.name = varName, so varName = ssRep.name = pisdRep.name.

We have now discussed the whole example and can now understand what the example transformation rule accomplishes: it accepts a Service Specification model that uses the ISDL meta-model and outputs a Platform Independent Service Design model that also uses ISDL as a meta-model. The input and output models will be identical:

Input
<repository name="isdlRepositoryName"/>

Output
<repository name="isdlRepositoryName"/>

Since ISDL is more complex than just a Repository, we have defined mapping rules like these for all commonly used classes in ISDL.

6.2.2 Transformations
The actual transformation rules that are written in the context of this paper are more complex then the basic example given in QVT, yet utilize the same basic syntax. We will expand the first QVT example. The result can be found in QVT.

QVT 2: Repository transformation
1 transformation ss2pisd(ss : emf, pisd : emf) {
2 top relation RepositoryMapping {
3 repName : String;
4 modName : String;
5 6 checkonly domain ss ssRep : Repository {
7 name = repName
8 model = ssModel : Model {
9 name = modName
10 };
11 }
12 13 enforce domain pisd pisdRep : Repository {
14 name = repName
15 model = pisdModel1 : Model {
16 name = 'to_' + modName + '_'1'
17 model = pisdModel2 : Model {
18 name = 'to_' + modName + '_'2'
19 };
20 }
21 }
22 }
23 }

The overall syntax of QVT appears to be the same as QVT, however, some lines are added. When looking at lines 8 through 10, we find that the property model of the Repository with the name ssRep is being evaluated. This property is of the type Model and has a property name: we store the value of this property in the variable modName.

Although it may not appear this way, we are actually pattern matching in the first part of this relation: in the input model, we are looking for a Repository that has a name and a model property. This is described in lines 7 and 8. The model1 from line 8 also has a property name. If we look at the ISDL meta-model, we will find that a Repository may have an arbitrary number of Models. This pattern will thus match any Repository that has any number of Models greater than 0.

Now we have found a Repository that matches our criteria, we can start constructing an output model. For the purpose of this example, we minimize the output; we construct a Repository with the same name as the input Repository has, but containing two Models: both have the String to_ prepended and a number.
appended, either being _1 or _2. The lines that perform these String operations are 16 and 19 respectively. The name of an output model will thus have the form to_<name>_number.

Input
```
<repository name="isdlRepositoryName">
  <model name="p1"/>
</repository>
```

Output
```
<repository name="isdlRepositoryName">
  <model name="to_p1_1"/>
  <model name="to_p1_2"/>
</repository>
```

The full patterns for transforming the rules from [16] are more complex, matching complete classification rules consisting of multiple classes and properties at the same time.

6.3 Models
As said in section 5.3, the models we use in this research are derived from the example described in section 2. The input model, at the Service Specification level, is depicted in Figure 5(a). This model visualizes the essence of the Live Contacts case: it specifies a context-aware application that allows its user to, among others, chat, e-mail, send SMS messages, and call “buddies”. The availability of these services depends on the context of the user.

6.4 Transformation
The input model described above is used as input for our transformation rules, resulting in a more detailed model: we know we can add certain information based on fixed patterns given by our classification rules. However, when further examining these patterns, we find that some classification rules utilize external data sources. Information about these sources is not known to our transformation rules, thus it is not possible to fully generate the PISD diagrams; we need additional information about the external sources in order to integrate them in the output.

6.5 Correctness
To ensure our transformation rules are correct, we compare them with the Platform Independent Service Design model given in [16]. This model can be found in Figure 5(b).

The main difference between the output our transformation rules generates and the given output model from Figure 5(b) is that our diagram does not contain detailed information about external data sources. The writers of [16] had additional information on these sources, our transformation rules do not have access to this data.

6.6 Discussion
The transformation rules we create for the enhancement of SS models to more detailed PISD models follow the same rules as the examples from sections 6.2.1 and 6.2.2 but contain more detail and are more complex. However, in spite of the rules being generally applicable to all SS models, some pattern matching constructions will only work for input data that is similar to the Live Contacts case. An example is the combination of the simple and the search classification. In [16], these patterns are used in cohesion, even though they are separate patterns: after a user input event, an external datasource is always queried. This information is then used for the subsequent output event. It is therefore demanding to recognize these patterns separately. However, if these two patterns are always used in combination, then the pattern is quite distinct and requires little effort to find.

As we already pointed out in section 6.4, it is not possible to fully generate the PISD model to the detail our reference PISD model has. Since external data sources are used and the SS model does not contain information about these, we can not include this information in the model generation process; the only plausible solution to this problem is to mark the position a datasource will be needed, allowing the user to quickly identify these places in order to manually add datasource information. We consider this an acceptable trade off between the added complexity of acquiring the required additional data and usability.

7. CONCLUSION AND FUTURE WORK
7.1 Conclusion
In this paper, we have proposed a set of Query/View/Transformations Relation rules, that will allow for partial automated enhancement of service specification level diagrams of context-aware mobile services that utilize the Interaction System Design Language meta-model; by pattern matching according to a combination of five predefined classification rules, enhancements are performed, or places where manual enhancements should be done are tagged accordingly.

When looking at the current situation, we can say that the QVT
Relations language is suitable to perform automated model enhancements in the domain of context-aware software systems. However, we encountered some problems. We will be discussing these in the next section.

### 7.2 Future work

As said in [6.5], not all information we want to add can be added to the PISD model. Instead, markers are placed to indicate actions needed by the user. For the transformation rules to be as helpful as possible, new ways of including external information in the QVT rules should be explored.

When looking at the input models with an editor like Grizzle, different kinds of objects appear differently in the diagram. However, when looking at a syntactical level, we find that almost all objects are ActionInstantiations. When looking at a single ActionInstantiation, it is not possible to determine whether we are dealing with a user input, output, or database access event. The actual meaning can only be determined by looking at the context in which it is used.

Another problem we encounter when observing the textual representation of the diagrams, is that a ChoiceRelationCondition is duplicated for all objects that originate from it. This is the cause of much overhead: if the choice exists between four objects, then within the object declarations, the ChoiceRelationCondition for the other objects is also mentioned, resulting in four times more code than strictly needed. To prevent errors resulting from these fundamental design problems, changes will have to be made to the ISDL meta-model: distinct object classes should receive distinct names, and redundancy in the diagram code should be avoided.

While working with the input and output models, it becomes apparent that the Grizzle tool lacks the ability to import XMI-based models, even though it is possible to export them. To stimulate the usage of our QVT rules in combination with Grizzle, we suggest that either Grizzle gives the possibility to import XMI files, or alternative tools for working with ISDL models are created. When choosing the latter, the development platform of choice ought to be Eclipse; by doing so, working with the models in a graphical way and performing automated refinements can be done in a single, consistent development environment.

### Acknowledgements

Firstly, we would hereby like to thank Laura Daniele for her technical expertise and useful comments on my paper. Secondly, Luís Ferreira Pires for his insights and pointers. Thirdly, we thank Lodewijk Bergmans and Marielle Stoejinga for their help in the overall process of writing this paper. Finally, our thanks go out to the students of the “Building High-Quality Systems” bachelor project track for their comments and reviews.

### REFERENCES


