Designing a Change-Driven Language for Model Consistency Repair Routines

Master Thesis Proposal of

Heiko Klare

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer: Prof. Dr. Ralf H. Reussner
Second reviewer: Jun.-Prof. Dr.-Ing. Anne Koziolke
Advisor: Dipl.-Inform. Max. E. Kramer
Second advisor: M.Sc. Michael Langhammer

Contents

1 Introduction 1
  1.1 Motivation .......................................................... 1
  1.2 Research goals .................................................... 2

2 Foundations 3
  2.1 Model-Driven Software Development ............................. 3
    2.1.1 Models and Transformations .................................... 3
    2.1.2 Domain-Specific Languages .................................... 4
  2.2 Eclipse Modeling Framework ...................................... 5
    2.2.1 Xtext .......................................................... 5
    2.2.2 Xbase and Xtend ............................................... 6
  2.3 Change-Driven Development ....................................... 6
  2.4 Vitruvius ................................................................ 7
    2.4.1 Change Meta-Model .............................................. 8
    2.4.2 Mappings, Invariants and Responses ......................... 8

3 Related Work 11

4 Concept 13
  4.1 Language Structure and Execution ............................... 13
    4.1.1 response Triggers ............................................... 13
    4.1.2 response Effects ............................................... 15
    4.1.3 response Transformation and Execution ...................... 16
    4.1.4 Implementation ............................................... 16
    4.1.5 Language Extensions ......................................... 19
  4.2 Evaluation ............................................................ 19
  4.3 Assumptions .......................................................... 20
  4.4 Restrictions .......................................................... 21
  4.5 Risk Management .................................................... 22

5 Process 23
  5.1 Advisors ............................................................. 23
  5.2 Artifacts ............................................................. 23
  5.3 Environment and Tools ............................................. 24
  5.4 Scheduling ........................................................... 24
    5.4.1 Iterations ....................................................... 24
    5.4.2 Gantt-Chart .................................................... 25
Contents

Bibliography 27
1 Introduction

1.1 Motivation

Model-driven development becomes a common approach for software engineering in various domains. Models raise the abstraction of software to a higher level by omitting language specific elements and proving the possibility to generate any desired code from the models.

The information of a software system is typically spread across several artifacts. These can be for example specification documents, UML diagrams and program code. All these artifacts are models for the same final executable software. They contain information of different aspects or abstractions of the system under development, but still have contents that are also present within other models. These semantic overlaps are often not formally persisted [5]. The problem arising from this redundant information can be recognized in traditional software development processes, where a software is designed in a diagram and implemented in program code. The tracing information, which specifies the diagram elements a code fragment corresponds to, is not recorded explicitly but just declared implicitly by equal naming. The absence of this tracing information can easily lead to inconsistencies because a modification may not be propagated to the corresponding elements in other diagram. A simple example is the renaming of class in program code, which is not propagated to a related UML class diagram.

A solution to this problem would be the definition of a single model, that contains all the information without redundancies that is necessary for a software system. Such a so-called single underlying model (SUM) was introduced by Atkinson, Stoll, and Bostan [3]. Nevertheless, in practice this approach is not applicable because there are always different, heterogeneous models involved in a development process. The Vitruvius project [52] addresses this problem by treating a collection of models as a virtually single model by allowing its modification only through well-defined views. Internally, the redundant information of the models is kept consistent by an explicit consistency mechanism. Therefore, the developer has to define the correspondences between different models that have to be kept consistent. The language for this purpose comprises a declarative approach, that allows the definition of mappings between elements from which routines are generated that automatically keep the elements consistent. Complex consistency constraints cannot be described with this approach, which requires the expensive implementation of plain program code. To simplify the ensurance of these constraints, an explicit language is needed that is capable of ensuring arbitrary consistency constraints.

The lack of such a language will be addressed in this thesis. Inconsistencies between models can only arise if one of the models is changed. Thus, a change-driven approach, which executes routines in reaction to a model change event, is selected to solve this
problem. To ensure that the limitations of the definition of a consistency repair routine is minimal as possible, a Turing complete language will be utilized to specify that routine. Although the development of this language is based on its need for complex consistency preservation, it can be also used isolated to ensure model consistency using a change-driven language.

The remainder of this proposal will be structured as follows: The following section will present the research goals of this thesis. In Chapter 2 the foundations required for this proposal and the thesis will be explained. Chapter 3 gives an initial overview of other work that is related to the topic of this thesis. In Chapter 4, the concept for the developed language, its evaluation, as well as assumptions and restrictions of the thesis are presented. The final Chapter 5 summarizes the process of the thesis, defining the produced artifacts, the used tools and the scheduling of the performed tasks.

1.2 Research goals

The goal of this thesis is the development of a language that enables the user to define imperative code routines that restore model-consistency and are triggered by change events in the models. To design and implement that language the following research questions have to be answered during the realization of the thesis:

1. Which kinds of model consistencies cannot be ensured using declarative mapping approaches?

2. How can triggers for the execution of model-consistency restoration routines be defined based on atomic changes in a model-based change-driven development environment?
   2.1. Which atomic changes exist, how do their parameters differ and how can they be represented in the trigger specification?
   2.2. Which event information must be provided to the routine?
   2.3. Are other triggers more suitable for certain recurring cases?

3. Which language is appropriate for defining imperative code blocks that can restore arbitrary consistencies between models after atomic changes?
   3.1. How can the models, which have to be updated, be specified?
   3.2. Is an existing language adequate for this purpose?
   3.3. Can execution loops due to dependent changes be avoided by restricting the language constructs or using static code analysis?
2 Foundations

2.1 Model-Driven Software Development

Model-Driven Software Development (MDSD) is a generic term for techniques that automatically generate executable software from formal models [48, p. 11]. The central idea is to raise the level of abstraction of software development from program code to models, which are automatically transformed into executable programs. MDSD aims to increase the per-developer productivity as well as the quality and reusability of software components [48].

2.1.1 Models and Transformations

The main artifacts in MDSD are models and transformations. Models completely describe a certain aspect of a software system, whereas transformations specify how models can be converted into other models.

A model is a representation of objects and their relations restricted to the needs of a special use case. Several established model definitions have been proposed, whereof the most suitable in the context of software development is given by Stachowiak [47]. In MDSD, especially metamodels are important. They define how valid models have to be created [2] and are in turn models themselves. A metamodel consists of several artifacts [48, p. 28ff.]:

- The most essential component is the **abstract syntax** that represents the available model elements and relations between them. It defines the structure of documents that are written in that language and at the same time the structure in which such a document is internally represented, analyzed and persisted. A parser creates the representation of a document according to an abstract syntax which is the basis for machine processing of the contents, such as code generation.

- The **static semantics** specify well-formedness constraints a valid model must fulfill. Harel and Rumpe call these semantics “context conditions”, which are checkable conditions that reduce the set of valid language instances in addition to the restrictions of the abstract syntax [24]. These constraints can, for instance, be specified using the Object Constraint Language (OCL) [36]. The purpose is to prevent the definition of language instances that have no well-defined semantics.

- One or more **concrete syntaxes** specify how a language instance can be represented. Examples for different forms of concrete syntaxes are textual, graphical or tree-based representations. These syntaxes are essential for the human interaction with the language and for the persistence of language instances.
• The last component of a language is its semantics, which defines the meanings of the language elements, their relationships and representations. The semantics of a language can be defined in free text or in more formal ways. One example are transformational semantics [40], that allow the definition of semantics by specifying the transformation into another language that already has defined semantics. An example is the specification of a transformation of some developed language into Java which already has a defined semantics.

A concrete model is an instance of a meta-model, thus a set of objects and relations satisfying the abstract syntax and semantics. In fact, each model has a meta-model it conforms to and since a meta-model has a meta-model itself, that model can be called the meta-meta-model. The arising cascade of meta-models can be bound by a meta-model that is self-defining, which means that it is its own meta-model. An example for such a meta-model is the Meta Object Facility (MOF) [35].

Model transformations specify how a model can be converted into another one of the same or a lower level of abstraction. Model transformations are formalized in [1]. Transformations can be classified as model-to-model or model-to-code transformations. The former ones define conversions of a model into another model according to a new or the same metamodel, which is especially needed for lowering the abstraction of models to finally reach a level that can be transformed into program code. The latter ones specify how a model can be transformed into program code, such as Java.

Since model transformations are of central significance, several languages and tools for specifying them have been developed. Two popular examples are Query/View/Transformation-Operational (QVT-O) [34], an imperative language for model transformations, and the Atlas Transformation Language (ATL) [26], a hybrid language containing declarative and imperative constructs.

2.1.2 Domain-Specific Languages

A domain-specific language (DSL) is “a computer programming language of limited expressiveness focused on a particular domain” [22]. In contrast to a general-purpose language (GPL), which is designed to implement arbitrary programs in any domain, a DSL is designed for a special domain. In that domain, the language provides constructs that provide a more compact or convenient way to implement scenarios that occur recurrently in that domain. However, it is possibly much more difficult or even impossible to implement scenarios that are not envisaged by the language.

Certainly, the given definition of a DSL does not provide a way of definitely classifying a language as DSL or GPL. There is even no common understanding in literature of what a DSL is. The main characteristics of a DSL is its purpose. According to Krahn [27], a language can be considered as a DSL if it is based on the semantic aspects of a special domain, if it allows to solve problems of the domain in a compact way and if its expressiveness is constrained to the needs of the domain.

Finally, a DSL can be considered as a meta-model since it consists of the same components a meta-model does: It comprises an abstract syntax, a static semantics, one or more concrete syntaxes and a defined semantics. In literature, there is no clear separation
between the two namings. To separate them in this thesis, the term meta-model will be used if the focus is the structural representation of data and its behavior with little importance of the concrete syntax, and it will be called DSL if the focus is the specification of a new language for the convenient specification of domain-specific concepts, where the concrete syntax is very important due to user interaction.

DSLs can be separated into internal and external DSLs. Internal DSLs are embedded into another language and reuse concepts of that host language. Thus, the implementation is generally easier in contrast to an external DSL but the language is constrained to the constructs the host language provides. An example for an internal DSL are UML Profiles, a mechanism of the UML [37], that allows the definition and dynamic application of extension to the UML language core. An external DSL is a completely independent language that has to define its own abstract syntax and semantics and, thus, potentially requires higher effort for its implementation. However, external DSLs are more expressive and flexible than internal DSLs are. In this thesis, external DSLs will be used.

### 2.2 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is a project extending the Eclipse platform with modeling capabilities [49]. It provides concepts and tools for defining models based on a meta-model called Ecore. The purpose of EMF is the unification and integration of models into the software development process as primary artifacts [49, p. 15].

Ecore is the meta-model that comes with EMF. It implements a subset of the Meta Object Facility (MOF) [35], which is the architecture on which the UML is based. It is compatible with Essential MOF (EMOF), a subset of MOF.

The EMF project contains several tools for the automatic generation of code from EMF models. This especially includes the generation of Eclipse plug-in projects that realize a graphical editor for instances of models specified according to the Ecore meta-model. Several tools have been developed that base on EMF and especially the contained Ecore meta-model. Two of them, which will be pervasively used in this thesis, are Xtext and Xtend, which is why they are described in the following sections in more detail.

#### 2.2.1 Xtext

Xtext is an EMF-based framework that supports the implementation of textual DSLs [15] and is developed by the itemis AG. It provides an editor, in which a textual language grammar can be specified in an EBNF-like notation. From this file, it automatically generates an EMF-based meta-model that represents the concepts and structure of the grammar, a parser that generates a corresponding model instance from a textual notation according to that grammar, and an editor with syntax highlighting, code completion, error checking and further features improving its usability. The validation, scoping and linking of language constructs can also be customized by implementing predefined extension points.

For a language defined in Xtext, code can be generated in two different ways: A manually written generator can produce arbitrary code for a model that the automatically generated parser delivers for a document written in that language. Xtext also supports
the generation of Java code from a model if the mapping of the language elements to elements of the Java language is specified. Then, also the scoping and return types of expressions are inferred by applying the specification of the Java language for the referenced concepts.

DSLs which are defined using Xtext can be reused since Xtext provides a mechanism for extending and using already existing languages. One basic expression language called Xbase is shipped with Xtext and is explained in the following section more detailed.

### 2.2.2 Xbase and Xtend

Xbase is an expression language that is shipped with Xtext [16]. It is tightly integrated with the Java type system and can be inherited by DSLs to provide a Java-like expression language. Xbase provides a parser for generating a model for textual notations in that language and a compiler that generates Java code for given Xbase expressions. Additionally to the concepts of Java, Xbase also provides type inference, operator overloading and extension methods.

Xtend [16] is an application of the Xbase language. It is an object-oriented, statically typed programming language for the Java Virtual Machine (JVM) but claims to provide a more concise notation by omitting redundant information that has to be written using Java. In addition to the more compact syntax, Xtend provides some new features, for instance, multi-methods that allow the dynamic dispatch of method parameters, type inference and template expressions. Xtend classes can be compiled to Java classes and thus be used in a plain Java project.

Since Xbase is integrated with the Xtext framework and can be reused in any self-developed language and since Xbase, as part of the Xtend language, can be compiled to Java code, it can be used inside a DSL which will be executed in a Java program with minimal effort. The required scoping, validation and compilation is already provided for expressions in that language.

### 2.3 Change-Driven Development

Change-driven software development is a technique that focuses on the execution of program code whenever a specific change of data value occurs. Breu describes this principle in [10]. Although that paper is written in the context of changes in the development process rather than the in the developed software, the statements can be transferred.

In change-driven development, the occurrence of a data modification can trigger different kinds of reaction. In general, a particular piece of code is executed whenever a specific change happens. This procedure is similar to the observer design pattern [23], which implements a notification mechanism in object-oriented software development. Change-driven development is more abstract than that because it does not specify a concrete realization of the mechanism and, furthermore, is not restricted to object-oriented software design.

Change-driven development can use two different evaluation mechanisms: Push-based systems use notification mechanisms that call other code whenever a change occurs. Pull-
based systems have a main loop that iteratively checks for changes and in case calls the appropriate routines.

In object-oriented development, a similar approach called event-driven programming [39] is popular. It is a paradigm that focuses on events and handlers that are called whenever an event occurs. The real difference to change-driven development is the application context. Event-based programming is often used in graphical user interfaces, where the interaction with the interface produces events that lead to the execution of handlers. In that context, event-based systems are mostly pull-based. Except for that and the naming, there is no explicit distinction between change-driven and event-driven development. Finally, the names event and change can be used equally because a change is an event and an event in software development does always require the change of some state.

Reactive programming is another programming paradigm in this context [19]. Its purpose is similar to the event-driven approach but addresses some of its drawbacks. Reactive programming facilitates the declarative development of event-driven programs, rather than the classical imperative approach [4]. Only data dependencies have to be specified declaratively and the environment decides when and in which order changes have to be propagated. The advantage is that the developer must not explicitly ensure that update routines are called when a change occurs because it is automatically generated from the dependency. Furthermore, event-driven programs often use callback mechanisms with limited possibilities of parameter passing, which leads to side-effects performed by the callback routines. These problems are tackled by the reactive programming approach.

Reactive programming may not be mixed up with the term reactive system of the Reactive Manifesto [9]. There, the term reactive focuses on the responsiveness of a system which the user perceives, rather than the control flow of the program execution, which reactive programming focuses on.

The presented concepts can also be applied to model-driven development. Model modifications lead to change events that can initiate arbitrary reactions. The reactions can reach from simple value changes to complete code routines and model transformations [6].

### 2.4 Vitruvius

The Vitruvius framework [31] is an approach for the consistent development of software that is based on heterogeneous models using views. It is built upon the idea of a Single Underlying Model (SUM), which was introduced by Atkinson, Stoll, and Bostan in the Orthographic Software Modeling (OSM) approach [3]. The main idea is to have a single model, the SUM, containing the whole information a software needs and comprises. While the benefit of that approach is the absence of redundancy, in practice it is impossible to represent everything that is needed for a software in development within one model. This led to the idea of a Virtual SUM (VSUM), which consists of several models that can not be modified directly but only via well-defined views. These views together simulate a SUM although it in fact comprises several models. Since these models contain redundant, overlapping information, the model developer has to specify these overlaps, so that an internal consistency mechanism can ensure that these information is kept synchronized.
Vitruvius is based on the idea of a VSUM. It allows the definition of so-called flexible views [11][12] using the DSL ModelJoin [14][13], which was developed for this particular scenario. Flexible views can show information of different, even heterogeneous models within a single view. For ensuring consistency of redundant information in different models, the Mappings, Invariants and Responses (MIR) language [28][30][29] is currently being developed.

2.4.1 Change Meta-Model

The Vitruvius framework is based on the reaction to changes. It observes models for modifications and generates change events from these modifications that lead to the update of views and consistencies. To keep this mechanism simple as possible, only atomic changes are observed and handled. Atomic changes comprise the creation, deletion and update of single model elements. Because single model elements can be values, references and even collections of them, there are several possible changes that can occur. Figure 2.4.1 shows the meta-model of atomic changes that is currently implemented in Vitruvius.

2.4.2 Mappings, Invariants and Responses

The MIR language is a change-driven approach for ensuring and repairing consistency constraints of models. Based on the constraints and operations defined with the language, appropriate update routines are executed when the framework reports an atomic change event. The mappings part of the MIR language allows the definition of declarative mappings between elements of different models that constrain their relationship. The routines that restore these constrains after a change are automatically generated from the mapping. The invariants part allows the specification of invariants that must always hold. The developer can also enrich invariants with parameters that are bound to concrete values that are causal for the violation of the constraint. If an invariant is violated, it will be the purpose of the responses part of the language to react to that in the future.

The responses part of the MIR language is the topic of this thesis. Its purpose is to specify imperative code routines that are executed in the case of defined atomic change event or an invariant violation prospectively. They shall contain complex Turing complete logic to address consistency constraints that can not be ensured using mappings.
Figure 2.1: Change meta-model in Vitruvius, from Vitruvius SVN Repository [52]
3 Related Work

Due to the increasing significance of model-driven development and especially the implementation of multi-model scenarios, the ensurance of model consistencies gets more and more important. Change-driven model consistency is an application of change-driven techniques in the context of model consistency. These topics have already been investigated and discussed in several research works.

The term change-driven is suitable in the context of model-driven development and especially model consistency ensurance since necessary operations arise from changes that are performed in models. The concepts are similar to event-based or event-driven software development [39]. Events lead to the execution of some program logic as a reaction to that event. Event-driven programs are often implemented manually using common imperative object-oriented language constructs and possibly fitting design patterns like the observer pattern [23]. In contrast, reactive programming focuses on language constructs to define data dependencies that are automatically updated. The mechanism of calling the reactions to events is not implemented manually but is provided by the environment, which is what change-driven model consistency also aims for.

The reactive programming idea comes especially from Elliott and was originally a part of a virtual reality modeling language [17]. Later on, he used this idea for a reactive animation framework [19] and recently in the context of general functional reactive programming [18]. Many reactive programming frameworks have been developed, especially for functional languages. Popular implementations for Haskell are Fran [17], NewFran [18] and Yampa [25]. Reactive programming constructs for Scala have been introduced in Scala.React [33]. The frameworks comprise essential ideas that can be transferred to change-driven model consistency frameworks but often also consider aspects that are not necessary or easy to reuse in that context. One example is the so called glitch avoidance, which is a mechanism for ensuring that no inconsistent states are visible during updates. This is achieved by ordering and updating the dependent expressions topologically [4].

Concerning model consistency, a lot of research has been done in the recent years. It can be separated into two categories: Research that only investigate model consistency checking and others that address the ensurance and repair of consistency constraints. Consistency checking approaches reach from validating consistency rules inside a model [53], over incremental techniques that claim to improve the performance [43] to multi-model approaches, which check constraints that specify the consistency of different models [38].

For this thesis, approaches for ensuring and repairing model consistencies are of special interest. Most of them are concerned with the generation of repair actions from defined constraints. The language Beanbag [54] provides an OCL-like syntax for defining constraints inside a single model. For several expressions in the language, routines are provided that restore consistency in the case that a model change violates the constraint. If a
change is performed, it is checked if all constraints hold or can be automatically repaired. Otherwise, the change is rejected. An approach for ensuring multi-model consistency is provided in [32]. Instead of an OCL-like syntax, constraints have to be specified in an XML-like syntax for first-order logic, from which repair routines are automatically generated. An approach of Szabo and Chen [50] also addresses multi-model consistency. In their approach, the repair mechanism bases on the specification of relations between elements in different models. For specific changes and specific relations, repair actions are provided that automatically restore consistency. All these approaches are not capable of reacting to any event. Reder claims to provide a framework that detects inconsistencies in a model and calculates possible solutions as well as the effects of the provided solutions and lets the user select an appropriate one [42]. The focus of this work are consistencies inside the design model of a software development process and, thus, it only considers consistencies inside one model. Another rather interactive approach is provided in [21]. It uses the concept of state automatons to calculate the transitions from an inconsistent to a consistent model state. Because this mechanism can not automatically detect the suitable modifications to reach a consistent state, user interaction is needed.

The last topic, that is of interest for this thesis, is change-driven model transformations. The idea is to run small transformations whenever a model element is changed and only update a small part of the target model instead of rerunning the complete transformation. An approach implementing this idea is the language PMT [51]. This language is stateless and calculates the elements of the target model that are affected by a change within the source model on demand and re-executes only the part of the transformation whose source changed. Another approach is provided by Bergmann et al. They propose VIATRA 3 [6] [41], a reactive model transformation platform, which can perform model transformations in reaction to a model change. The platform is based on EMF-IncQuery [7], a framework for incremental queries. In that framework, models are treated as graphs and a query can be specified by declaratively defining a graph pattern that is matched with the model graph. In VIATRA 3, a change can be specified through such a graph pattern, a so called change query, that describes which elements must exist, disappear or appear in case of that change. This allows the specification of complex change events but even a simple rename is complicated to describe. The transformation in reaction to a change is specified using the change query as a precondition and a graph pattern that specifies the postcondition of the transformation. The drawback of that approach is the complexity of defining graph patterns for specifying changes and postconditions and the limited expressiveness of graph patterns.
4 Concept

The goal of this thesis is to develop a language for specifying routines containing imperative code for repairing model consistencies and the change events that triggers their execution. The initial concept of the language will be introduced in this chapter. After presenting the prototypical structure of the language, the proposed realization of its execution and the evaluation of the results, the assumptions that are made for this thesis are summarized. The chapter closes with the explanation of topical restrictions that are made to keep the goals satisfiable and the treatment with potential risks that may inhibit the successful process of the thesis.

4.1 Language Structure and Execution

The language that will be developed in this thesis shall provide constructs for specifying a code block that is executed whenever a defined change event occurs. These constructs will be called responses since they define a kind of response to the occurrence a change event.

The language will consist of two major parts: The first part are the triggers that specify in response to which change events a response will be executed. The second part addresses the effects of a response operation. It comprises the specification of the models that have to be modified in response to the change event and the code that is to be executed for each of them. Furthermore, responses have to be executed in a run-time environment. Therefore, they have to be transformed into executable code that fulfills the semantics of the specified responses during run-time.

The following sections describe the currently planned characteristics of the first language increment that will contain a minimal amount of elements that completely fulfill the functional requirements to the language without considering its usability. The improvements to the language in the second iteration will be developed and planned after evaluating the first language increment with case studies. This ensures that more complex language elements match the needs of realistic usage scenarios.

The planned grammar of the initial version is sketched in Figure 4.1. A response starts with the keyword “response to” followed by the trigger and the effects. These components will be explained in the following sections.

4.1.1 response Triggers

The trigger of a response specifies the event after which the code of the response shall be executed. An event consists of the event type, for example, a property modification or an
4 Concept

\[
\langle \text{document} \rangle \ ::= \langle \text{root-element} \rangle^* \\
\langle \text{root-element} \rangle \ ::= \langle \text{import} \rangle \\
\quad | \quad \langle \text{response} \rangle \\
\langle \text{import} \rangle \ ::= \text{'import}' \langle \text{uri} \rangle \text{'as'} \langle \text{id} \rangle \\
\langle \text{response} \rangle \ ::= \text{'response to'} \langle \text{trigger} \rangle \langle \text{effects} \rangle \\
\langle \text{trigger} \rangle \ ::= \langle \text{change-event} \rangle \text{'of'} \langle \text{id} \rangle \\
\langle \text{effects} \rangle \ ::= \langle \text{affected-model} \rangle? \langle \text{code-block} \rangle \\
\langle \text{affected-model} \rangle \ ::= \text{'affects'} \langle \text{id} \rangle \text{('when'} \langle \text{boolean-expression} \rangle?) \\
\langle \text{change-event} \rangle \ ::= \text{A concrete change event type name} \\
\langle \text{id} \rangle \ ::= \text{Identifier according to the used language} \\
\langle \text{uri} \rangle \ ::= \text{Unified resource identifier for addressing a resource} \\
\langle \text{boolean-expression} \rangle \ ::= \text{Expression returning a boolean value according to the used language}
\]

Figure 4.1: Sketched grammar of the response language

object creation or deletion, and eventual parameters that constrain the execution of the response depending on the context in which the change event occurs.

In the first language increment, only atomic model changes will be implemented as triggers. A meta-model with the representation of all modifications that can be atomically changed in a model was already developed in the context of the Vitruvius project and is described in Section 2.4.1. That meta-model will be used as the basis for the triggers that are developed in this thesis. The language assumes that these change events are generated by a framework, which will be Vitruvius in the actual implementation.

Two models can only get inconsistent if one of them changes. Thus, if all possible changes in a model can be used as a trigger for a response, one can write responses for each case in which an inconsistency can arise. Although atomic changes may not provide a convenient way of describing a situation in which a response shall be executed, it is functionally sufficient to repair all potential model inconsistencies for the stated reasons. In the second language increment further triggers will be implemented depending on the needs that the examination of case studies expose.

Additionally to the change event itself, associated parameters will be available to restrict cases in which the response is executed. If no parameter is specified, the response would be executed whenever a change event of the specified type occurs, even independent of the meta-model which the modified model is an instance of. The parameters constrain the actual model elements whose modification lead to the execution of the response. This can be the meta-model whose instances shall be observed for the specified change event or the model elements that have to be modified to invoke the response. Considering a rename event, the meta-model and its renamed element are reasonable parameters. To allow a most compact and non-redundant specification of the parameter, it is planned to summarize the causal model element and the meta-model in one parameter
that contains the meta-model and the path to the model element. For example, the parameter of a class rename event in a UML model could be expressed as `uml.class.name` if `uml` is a reference to the UML meta-model, `class` is a model element that represents a class and `name` is a property of a class containing its name. The available parameters will depend on the actual change type. Thus, the presented grammar will be extended by a clear specification of the change events during the language design.

The initial language increment will provide only atomic changes, according to the Vitruvius change meta-model, as triggers. Therefore, only one parameter is needed in each case. If a non-root element is modified, the meta-model and the concrete model element have to be named, which can be combined in one parameter since the model element specification also contains the meta-model. Even the modification of a root element, although differently defined in the meta-model, bears on the root model element that is modified and thus has to be specified.

The specification syntax of a trigger in the response language is defined in Figure 4.1. It starts with the specific change event, which will be one of the atomic changes defined in the Vitruvius change meta-model. This is followed by the declaration of the model element whose change shall be reacted to and also contains the affected meta-model like described earlier. To make the change event referenceable in the effects the keyword `change` will be introduced.

The second part of a response specification comprises its effects. The effects of a response affect a specific model, to be exact the one that is not consistent with the modified model, and performs modifications in it to repair consistency constraints.

The affected model will be defined by its root element and constraints it must fulfill. The integration into the language is defined in Figure 4.1. The model element will be referenceable in the constraints and the code block using the keyword `target`. Following the keyword `when`, constraints can be specified to restrict the matching model instances. If no constraints are specified or if more than one model satisfies the constraints, the response will be executed for all of them.

If a change event requires modifications in instances of different meta-models, a response has to be defined for each of them. Combining the effects on instances of different meta-models in one response would cause problems in the case that multiple model instances match the constraints. In that case, the code would potentially be executed multiply for a single model instance because a model can occur in several combinations of matching models the code can be executed for.

The modifications in the affected models will be specified inside a block of imperative program code. In the developed language, an Xbase expression block will be used for specifying the effects of a response. Since the language Xtend is based on the Xbase language (see Section ??) and uses the same construct for specifying an expression block, the block can be copied into an Xtend method as it is and be executed there. In this code block, the change event which is specified in the trigger as well as the model instance that matches the previously defined constraints will be addressable using the defined keywords `change` and `target`.  

15
4 Concept

4.1.3 response Transformation and Execution

The execution of a specified response requires its transformation into executable program code. The implementation assumes the existence of a framework that provides the functionality to send notifications whenever one of the atomic changes presented in Section 2.4.1 occurs in any observed model. Furthermore, a repository of relevant models is assumed. The notifications will be handled by a repository that contains the responses, each represented as a mapping of its trigger to the effects. If a trigger matches the change event that is delivered by a notification, the corresponding effects are executed. The registered models are browsed for the existence of a model matching the constraints in the effects of the response. For all matching models, the effect code of the response is executed. The described workflow is shown in Figure 4.2.

The described run-time workflow will be achieved by transforming the specified responses into adequate program code. The general concept is based on the essential workflow in change-driven environments. For each response, a separate class is created. Each class contains a method for the execution of a response that expects an object that represents the change event. The execution is separated into three steps: At first, it is checked if the specified trigger of the response matches the actual change event. Afterwards, the method searches for all models inside the repository that match the conditions for the affected models specified in the response. Therefore, the conditional expression in the response language has to be transformed into a check operation. For the code that shall be executed, another method will be generated containing a copy of the original code. This method will be called by the initial execution method for each found affected model. A realization of such a response implementation is shown in Pseudocode in Listing 4.1. In a concrete case, the pseudo-class Model has to be replaced by the concrete model type of the affected model.

Furthermore, another class must exist in the run-time environment that maps a change event to corresponding responses. The mapping can also take the meta-model that is specified in the trigger into account. The mapping must be at least based on the change event type. This class must be registered at the notification mechanism for model changes, so it can call the response that has to be executed in reaction to a specific change event. Every time a change event occurs, all responses for that event have to be retrieved from the map and their execution methods have to be called.

4.1.4 Implementation

The developed language will be implemented using the framework Xtext (see Section 2.2.1). For the specification of code fragments, especially the code block, parts of Xtend (see Section 2.2.2) will be used. Xtend is predestined to be used in this context because it is implemented with Xtext, which allows its easy reuse in the developed language. Furthermore, it can be completely copied into an Xtend class or compiled into a Java class for its usage in the run-time environment. A primitive example response is presented in Listing 4.2. It describes the update of a Java class name whenever the name of a corresponding class inside a UML diagram changes.
4.1 Language Structure and Execution

Figure 4.2: Response execution process
Listing 4.1: Pseudocode template of a class generated for a Response

class Response {
    // Check if the change event matches the condition specified
    // in the trigger
    checkPrecondition(event : ChangeEvent) { ... }

    // Get instances of the affected metamodel from repository,
    // check which ones satisfy the specified constraints
    // and return those
    findAffectedModels() : Collection<Model> { ... }

    // Contains a copy of the code block specified in the
    // original Response
    runCodeBlock(event : ChangeEvent, affectedModel : Model) { ... }

    execute(event : ChangeEvent) {
        if not checkPrecondition(event) then return;
        affectedModels = findAffectedModels();
        for each model in affectedModels do
            runCodeBlock(event, model);
    }
}

Listing 4.2: Example Response ensuring consistency of a class name in UML and Java

import [...] as uml
import [...] as jamopp

response to UpdateSingleValuedEAttribute of uml.class.name
affects jamopp.classifiers.Class when target.name = change.oldFeatureValue {
    target.name = change.newFeatureValue
}
4.2 Evaluation

The transformation described in the previous section will also be implemented using Xtend. The generated run-time classes will be integrated into the Vitruvius framework. Vitruvius already contains a mechanism for executing commands whenever a change occurs. It also contains a correspondence mechanism that describes which meta-models potentially have correspondences. In the case of the developed language, these correspondences must exist for the meta-model specified in the trigger and the affected one specified in the effects.

4.1.5 Language Extensions

The first language increment can be extended in different ways. Extensions can be assigned to two categories: The ones that improve the response triggers and the ones that extend the effect specifications.

Regarding the triggers, it is possible to introduce new events that allow to use more complex triggers than atomic changes. One potential other type of trigger are invariant violations. If invariants are used to define consistency constraints on models, they can be evaluated in the case that an atomic change occurs and then trigger a violation event. Invariants and their evaluation are already implemented in the MIR language of the Vitruvius framework and can be used for this purpose.

The specification of atomic changes can also be simplified. The change meta-model of Vitruvius contains a single change type for each combination of event and element type. For instance, a remove event is defined for the value of attributes, for list elements and references. Although there is a single model element for each of these combination in the meta-model and, thus, makes it necessary to specify the correct type in the first increment of the response language, it is possible to simplify this specification. The affected element type must not be specified in a response because the given parameter implicitly defines the element type. An appropriate logic could reduce the necessary specification to the event type.

The second category of improvements concerns the effects of a response. One possible extension is the implementation of a reuse mechanisms that allows to write code once and execute it in different responses. Furthermore, the execution of responses can potentially lead to endless loops if there are two or more responses whose effects trigger the others in each case. Due to the low coupling of responses to the context in which they can be triggered, it is harder for the developer and even for static code analyses to recognize those cases. Thus, working out a language subset that cannot lead to loops or can be easily analyzed for loops and integrating that knowledge into the language editor may improve the usability of the language.

4.2 Evaluation

The evaluation of the developed language will focus on the demonstration that responses are functionally capable of solving the underlying problem. Tests will show if they are sufficient for the definition of change-driven consistency repair routines that are needed
in practice. The evaluation will be performed empirically with existing case studies and will consist of three parts:

1. To proof the capability of reacting to each possible change event, a minimal meta-model, that contains each model element type, will be developed. Tests will be written for an instance of that meta-model provoking each change event type by an appropriate model modification. For each event type a response will be specified that transfers the modification into another equal model, so that a compare of both models after the response execution can proof the functionality.

2. The implemented language shall enable the user to define arbitrary consistency repair routines. To show that this is possible, the functionality will be evaluated using realistic examples. These examples come from three case studies: The first one is a consistency mechanism for Palladio Component Model (PCM) [44] instances and Java code. Two further ones come from the automotive domain, where the modeling languages ASCET [20] and AMALTHEA [46], as well as Franca [8] and AUTOSAR [45] are used and provide overlapping information that has to be held consistent. Examples from that case studies, that cannot be implemented with the Mappings mechanism of the MIR language, will be tested with the response language to test its capabilities.

3. The last evaluation concerns the improvements of further iterations in contrast to the initial and minimal language increment. Since it is currently not clear which improvements will be implemented, it is not possible to specify concrete metrics for this evaluation. The goal of the improvements is to make the definition of responses shorter and less redundant. Redundancy can potentially arise from the necessity of duplicating response content because it has to be executed in reaction to different events. Potential metrics for measuring the improvements are the reduction of logic code lines over all the test cases or the reduction of API calls that are necessary due to the introduction of new keywords.

4.3 Assumptions

The developed language will have several, especially technical, dependencies and requirements which are assumed to be existing and working correctly. Important assumptions will be presented in the following.

To ensure correctness in the theoretical background of this thesis, we assume a complete meta-model of potential atomic changes in models. This is necessary to ensure that each case in which an inconsistency can arise can be reacted to with a response. Thus, the change meta-model of Vitruvius, presented in Section 2.4.1 in the previous section, is assumed to be complete for Ecore-based meta-models.

For a working language implementation, all used tools, especially Xtext, Xtend and parts of Vitruvius, are assumed to operate the way they are specified. Furthermore, Xtext is supposed to provide the required functionality to implement the language that was described in the previous section. Vitruvius provides the functionality for detecting changes
in models and is supposed to produce appropriate change events in each case. It is assumed that responses and their execution can be integrated into the existing Vitruvius framework without essential modifications. The Vitruvius framework uses a correspondence mechanism to define and retrieve which model elements have equivalents in other models. This mechanism will be used to find appropriate responses when a change occurs. Thus, its implementation is supposed to be correct and working.

Since no case studies will be examined before implementing the first language increment, the usage of atomic changes as triggers and the specification of arbitrary code in the effects is supposed to be expressive enough to implement arbitrary consistency repair routines. This rests upon the fact that only a change in a model can lead to an inconsistency between that and another model and that change type can be used in a trigger to specify a repair routine. If the examination of case studies shows that this assumption is wrong against expectations, cases in which the language is not expressive enough would have to be covered with language extensions in the second increment.

In the case that invariant violations are exposed to be useful triggers for responses, it is assumed that the invariants implemented in Vitruvius, their evaluation and a mechanism for calculating parameters that lead to the violation of an invariant are completely and correctly implemented in Vitruvius. Furthermore, the framework is supposed to check invariants everytime a model is changed and produces the appropriate violation events.

### 4.4 Restrictions

To ensure that the goals of this thesis are reachable with proper effort, some limitations have to be made. The goal is the implementation of a prototype and proof-of-concept of the desired language within a research project and an initial evaluation of its functional capabilities.

A deep integration of responses into the Eclipse platform to ensure a convenient usability of responses by end users is not part of the thesis. The language will be integrated into the Vitruvius framework and will be used together with the rest of the MIR language. Thus, the integration of these languages, so that they can easily be used by end users, will be part of the Vitruvius project.

The thesis will not provide proofs for the language capabilities. The sufficient expressiveness of the language will be tested with the implementation of case studies. These studies will also be the basis for the improvements of the language in the second and possibly further increments. Although, the final increment of this thesis will not be evaluated in a separate case study. This may be a task for further work.

As described in Section 4.1.5 the execution of responses can lead to loops that are hard to detect by a developer or code analyses. This thesis will probably not develop mechanisms that reduce, guess or detect those situations.

In this thesis, the first increments of a new language will be developed. It focuses on the technical operability and correctness of the language and some improvements in its usability and suitability based on case studies. However, it does not primarily focus on perfect usability, which is why semantic validations in the editor will be implemented as far as possible but will be omitted when running out of time.
4.5 Risk Management

Several risks have the potential to jeopardize the successful completion of the thesis. To reduce the risks, this section summarizes plans and concepts to deal with them.

Satisfaction of Assumption

For the thesis, several assumptions, described in Section 4.3 have to be made. If one of the assumptions is or gets incorrect, the concept or technical realization of the thesis could fail or have to be essentially changed. In that case, further steps may be delayed or even impossible to realize. To reduce that risk, a sufficient buffer time is scheduled and the process is planned iteratively to have intermediate results. Wrong assumptions that are determined in an early phase can be dealt with by changing the concept since then there is no or just little implementation yet. If assumption faults are detected in later iterations, the early iterations are already completed and represent a usable state.

The assumed functionality of Vitruvius, especially the change notification mechanism and the correspondence information management, could also be mocked in case of bugs in the implementation to be almost completely independent from Vitruvius. A central assumption is that the reaction to atomic changes is sufficient to ensure arbitrary consistency constraints. If the case study exposes that this assumption is wrong, the available trigger would have to be extended.

Changes in the Vitruvius Framework

The developed language will be integrated into the Vitruvius framework. Since Vitruvius is a research project, major modification are not unusual. If the framework is essentially modified, it might affect the implementation of the response language as well. Therefore, the response language will be developed as independent from the framework as possible without duplicating concepts and code. The only true dependency to Vitruvius will be the meta-model for model changes and the mechanism for triggering the response execution in the case that an atomic change occurs. Thus, the risk that a modification in Vitruvius has a significant impact on the developed language is low.

Dependencies on Tools

The language implementation will base on several tools, especially Xtext and Xtend. In the case, that one of the tools is not available any more, the feasibility of the implementation would be questionable due to the lack of alternatives. However, since both tools are currently in a reliable and stable state, it is unlikely that they will not be supported or not working during the process of the thesis.

Iterative Process

The iterative schedule simplifies the reaction to potential problems. If an increment cannot be implemented, the previous increment is still working and can be potentially extended in another way. Especially the first iteration replaces a vertical prototype because it only aims to implement a first increment with basic functionality, which is not far beyond a simple prototype and allows to react on essential problems early.
5 Process

In this chapter the central aspects for the realization of this Master thesis are summarized. It is described who the advisors are, which artifacts are developed and which tools are used. Finally, the schedule is presented containing the iterations that are planned to design and develop the language and when milestones with executable increments are planned to be reached.

5.1 Advisors

- Reviewer: Prof. Dr. Ralf H. Reussner
- Second Reviewer: Jun.-Prof. Dr.-Ing. Anne Koziolek
- Advisor: Dipl.-Inform. Max E. Kramer
- Second advisor: M.Sc. Michael Langhammer

5.2 Artifacts

Following artifacts are planned to be developed during the processing of this thesis:

- Proposal (this document)
- Proposal presentation slides
- Source Code developed during the iterations described in Section 5.4.1:
  - Specification and validation of the developed language using Xtext
  - Generators for an execution environment realizing the specified routines and their triggers in Xtend respectively Java
  - Implementation of the case studies demonstrating the functionality and suitability of the approach
- Master thesis
- Master thesis presentation slides
5.3 Environment and Tools

This thesis is implemented in the context of the Vitruvius framework (see Section 2.4) and the contained MIR language for specifying model correspondence and consistency repair mechanisms. Eclipse is utilized as the Integrated Development Environment (IDE) since Vitruvius uses the Plug-in mechanism provided by Eclipse. It relies on the Eclipse Modeling Framework (see Section 2.2) for model-driven development in the context of the Eclipse Framework. Further extensions of the Eclipse framework and EMF that will be used are Xtext (see Section 2.2.1) for developing DSLs (see Section 2.1.2) and Xtend for developing source code that is transformed into Java code (see Section 2.2.2).

The case studies that are used to generate test cases for the developed approach depend on the Palladio Component Model (PCM), Java and the Tool JaMoPP for treating Java source code as an Ecore-based EMF model, as well as AUTOSAR, Franca, ASCET and AMALTHEA, which introduce model-driven techniques in the automotive domain.

5.4 Scheduling

According to the SPO2008 issued at the 8th of September 2008, the processing time for the master thesis is six months. The working time starts on the 14th December 2015 and ends at the latest on the 13th of June 2016. The progress will be discussed with the advisor in weekly meetings.

The thesis is split into several tasks, whose scheduling is planned using a Gantt chart in Section 5.4.2. The central artifacts of the thesis are developed in at least two iterations that are described in the following section.

5.4.1 Iterations

The language that will be developed in this thesis will be designed and implemented in an incremental fashion. At least two iterations with an interjectional examination of case studies, that are the foundation of the second iteration, are planned.

5.4.1.1 Initial language increment

The goal of the first iteration is the design and implementation of a most basic language that fulfills the functional requirements to the developed language. The language shall provide the ability to define arbitrary Xtend code blocks that are executed whenever a specific atomic change, defined in the Vitruvius change meta-model (see Section 2.4.1), of a defined property of a model occurs.

When the implementation is finished, the results of the first iteration will be written down in the thesis to reach a working and documented state of the language.

5.4.1.2 Examination of case studies

Before starting with the second iteration, the available case studies are examined. Cases, in which model consistency cannot be ensured with the declarative mapping mechanism
of Vitruvius, will be collected. The initial increment of the language will be evaluated with these cases, to demonstrate that the language is functionally usable to ensure these consistency cases.

### 5.4.1.3 Improved language increment

The collected cases and their implementations are used to work out cases in which the atomic changes are not suitable as the triggers for the execution of the model-consistency restoration routines. For these cases, new and more adequate triggers will be developed. After the new triggers are designed, they will be integrated into the already developed basic language. The goal is to develop a language increment that is not only functionally capable of fulfilling the language requirements, but also compact and expressive in recurring usage scenarios.

### 5.4.1.4 Further iterations

Depending on the fulfillment of the schedule after the improved language increment is finished, further iterations for improving the language capabilities and usability can be attached. Potential improvements to the effects of a response will also be implemented in this context.

### 5.4.2 Gantt-Chart

Figure 5.4.2 shows the current scheduling of the single tasks that will be managed in this thesis. To meet potential risks that may delay the achievement of a milestone, a buffer of seven weeks is included. An architecture and code review is scheduled after the evaluation of the first language increment using the examples from the case studies because their evaluation may lead to some essential modifications of the implementation which would make a previous review obsolete. A second code review will be performed after the final implementation is complete to improve and ensure the final code quality. The exact date of the reviews will be scheduled in due time.
<table>
<thead>
<tr>
<th>ID</th>
<th>Task name</th>
<th>Start</th>
<th>End</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Perform literature research, incl. Christmas break</td>
<td>14.12.2015</td>
<td>08.01.2016</td>
<td>20D</td>
</tr>
<tr>
<td>2</td>
<td>Initial language increment (1st iteration)</td>
<td>11.01.2016</td>
<td>05.02.2016</td>
<td>20D</td>
</tr>
<tr>
<td>3</td>
<td>Design language using Xtext</td>
<td>11.01.2016</td>
<td>15.01.2016</td>
<td>5D</td>
</tr>
<tr>
<td>4</td>
<td>Implement code generation</td>
<td>18.01.2016</td>
<td>29.01.2016</td>
<td>10D</td>
</tr>
<tr>
<td>5</td>
<td>Write first draft of the thesis</td>
<td>01.02.2016</td>
<td>05.02.2016</td>
<td>5D</td>
</tr>
<tr>
<td>6</td>
<td>Examination of case studies</td>
<td>08.02.2016</td>
<td>19.02.2016</td>
<td>10D</td>
</tr>
<tr>
<td>7</td>
<td>Collect significant scenarios</td>
<td>08.02.2016</td>
<td>12.02.2016</td>
<td>5D</td>
</tr>
<tr>
<td>8</td>
<td>Evaluate initial implementation using the collected cases</td>
<td>15.02.2016</td>
<td>19.02.2016</td>
<td>5D</td>
</tr>
<tr>
<td>9</td>
<td>Perform architecture and code review</td>
<td>19.02.2016</td>
<td>19.02.2016</td>
<td>1D</td>
</tr>
<tr>
<td>10</td>
<td>Improved language increment (2nd iteration)</td>
<td>22.02.2016</td>
<td>25.03.2016</td>
<td>25D</td>
</tr>
<tr>
<td>11</td>
<td>Develop language extensions based on scenarios from case studies</td>
<td>22.02.2016</td>
<td>26.02.2016</td>
<td>5D</td>
</tr>
<tr>
<td>12</td>
<td>Revise Xtext language</td>
<td>29.02.2016</td>
<td>04.03.2016</td>
<td>5D</td>
</tr>
<tr>
<td>13</td>
<td>Extend code generation</td>
<td>07.03.2016</td>
<td>18.03.2016</td>
<td>10D</td>
</tr>
<tr>
<td>14</td>
<td>Write second draft of the thesis</td>
<td>21.03.2016</td>
<td>25.03.2016</td>
<td>5D</td>
</tr>
<tr>
<td>15</td>
<td>Perform code review</td>
<td>25.03.2016</td>
<td>25.03.2016</td>
<td>1D</td>
</tr>
<tr>
<td>16</td>
<td>Finalize thesis workout</td>
<td>28.03.2016</td>
<td>08.04.2016</td>
<td>10D</td>
</tr>
<tr>
<td>17</td>
<td>Correct workout</td>
<td>11.04.2016</td>
<td>15.04.2016</td>
<td>5D</td>
</tr>
<tr>
<td>18</td>
<td>Prepare presentation slides</td>
<td>18.04.2016</td>
<td>22.04.2016</td>
<td>5D</td>
</tr>
<tr>
<td>19</td>
<td>Buffer</td>
<td>25.04.2016</td>
<td>13.06.2016</td>
<td>36D</td>
</tr>
</tbody>
</table>

Figure 5.1: Scheduling for the thesis tasks
Bibliography


Bibliography


