

Preface

Back in 1988, Gerald M. Weinberg, in his seminal book “Understanding the Professional Programmer” [180] has formulated the following statement: “...programming computers is by far the hardest intellectual task that human beings have ever tried to do. Ever.” We can discuss this statement and argue with it but we have to admit that software development is an extremely complex task, and this is so for various reasons. Probably the most concise summary of these reasons was given by Fred Brooks in his famous essay “No Silver Bullet” [26]. He has pointed out that software is composed of two types of complexity: essential and accidental. The essential aspect is inherent for the problem at hand and cannot be reduced. This complexity has to “be there” in software so that it would be able to solve the actual problem. The accidental aspect is associated with the computer technology and includes things like programming language constructs, distributed calls, middleware, UI technologies and so on. This kind of complexity resides within the software systems due to the complexity of computer technologies themselves.

The general diagnosis might be that “it has to be complex and we cannot do anything about it”. Or maybe we can? We certainly cannot reduce the essential complexity, because the reality is complex and we need to solve complex problems in our reality. However, what we can try to do is to “tame” the accidental complexity, or more specifically—hide this complexity from the software developers. How could this be done? The first step would be to define ways of specifying precisely the essence of the solution to the problem at hand. The second step would be to build programs that would transform this essence into a working software system. The more we automate this transformation, the better we hide the accidental complexity...

It can be noted that the essential complexity of software is closely related to requirements for software. Requirements define the problem to be solved but often—at a more detailed level—they also offer essential descriptions of this problem’s solution. Let us assume that we have a notation that can describe the various aspects of the essential solution with “high precision” (whatever this means). With enough precision we would have the potential to transform these detailed requirements

descriptions automatically into executable artefacts. In other words: we elaborate requirements with fine precision, and in a “snap” we obtain working code.

This scenario has one important obstacle: we somehow need to cater for the accidental complexity. To implement the “snap” we immediately face the question of where this complexity goes. For hints on answering to this question we can refer to the bygone era of the domination of assembly language programming. Assembly language programmers needed to deal with the complexity of computer architectures which included such elements like processor registers, arithmetic logic units, memory locations and so on. This complexity was abstracted away with the advent of Third Generation Languages (3GLs) like FORTRAN, Algol, Pascal, and later—Java or C#. Compilers for such languages contain specific rules that are “injected” into the 3GL code to produce equivalent assembly and machine code. By analogy, we can thus imagine “injecting” technology-related aspects during transformation from requirements into more detailed software artifacts and finally—executable code.

The process of translating from a 3GL code to assembly/machine code (or other executable code) through a compiler, is completely automatic. What is more, we can also apply automatic translation (transformation) to higher-level artefacts, like design specifications. The main idea is to be able to create precise artefacts (models) at certain levels of abstraction (or: complexity) and transform them to artefacts (models) that are more detailed and complex, which finally includes also code. This idea was formally formulated around the year 2000 and led to the concept known as Model-Driven Architecture (MDA) [111]. Later, somewhat more general names of Model-Driven Software Development (MDSD) and Model-Driven Software Engineering (MDSE) emerged. Currently, this concept can certainly claim maturity with more and more tools supporting it.

Still, it has to be noted that practically all the tools concentrate on transformations between various design-level models and generating code from these design models. Moreover, these transformations have to be interlaced with manual interventions of software developers. For example, high-level architectural models cannot be translated directly into code. They need to be transformed into detailed design models and then adapted manually to cater for certain aspects not covered by the automatic transformation engines. Only then can they be transformed into code, which can be compiled and executed.

Can the model-driven concepts be extended onto requirements? Requirements are usually seen as much less subject to formalization and thus not really suitable for model transformation. Despite this, recently a new area of MDSD has emerged in the form of Model-Driven Requirements Engineering (MDRE). It concentrates on defining ways to formulate requirements as precise models and transforming these models into various more detailed models with technology details “injected” (design models, test models, etc.). It can be noted that the ultimate goal would be to be able to transform requirements directly and automatically to executable code. Though the question arises whether we can make requirements precise enough to reach this goal, yet retaining their comprehension by customers...

Purpose and Scope

There are many books that deal with issues of requirements engineering and the role of requirements in the software engineering process [5, 13, 32, 93, 130, 136, 164, 173, 183]. Most of these books concentrate on eliciting and formulating requirements of good quality (unambiguous, consistent, understandable, complete, verifiable,...). Some of the books propose to use modelling notations like use case models, class models or data flow models. Such requirements are then placed within a process in which requirements are the basis for implementing a software system.

Furthermore, there are also several books on Model-Driven Software Development/Engineering (MDSD/E) [18, 25, 66, 92, 106, 127, 167, 178]. These books concentrate on defining models and transformations between them. This includes explaining the precise meaning of models (semantics) and using this meaning to develop automatic transformations to other, more precise models. These transformations are usually performed using various model transformation languages. Some of the books present ways to define new modelling languages that can be used to formulate problems in a specific domain. This particular area of MDSD is called Software Language Engineering (SLE) [91].

This book is meant to provide the reader with a coherent approach to combine both of the above worlds [187]. It presents systematic treatment of requirements within the realm of modelling and model transformations [102], i.e. Model-Driven Requirements Engineering. What is important as the aim of the book is to treat MDRE as comprehensively as possible. The basic assumption in this comprehensive treatment is that detailed requirements models are used as first-class artefacts playing a direct role in constructing software. For this purpose, the book presents the Requirements Specification Language (RSL) that enables precision and formality, at the same time retaining end-user comprehensibility. This is important for typical requirements engineering tasks like requirements elicitation, formulation and usage.

In the book, we assume that requirements engineers use typical ways to elicit requirements from the users and from the stakeholders, and we do not provide any special guidelines in this respect. However, we provide the means to formulate these elicited requirements in the form of precise RSL models. These models facilitate assuring good quality of requirements, including coherence, unambiguity (clarity) and completeness.

Good quality requirements models, expressed in RSL can be used in a standard (“manual”) way, to produce design models and implementation. Yet, the book offers a much broader use of requirements models. The ultimate goal of the book is to give the reader the means to automate the process of turning requirements into a working system. To achieve this, the book presents techniques to write and apply model transformations and code generation to RSL. What is crucial, this is supported by a state-of-the-art tool suite that accompanies this book. The suite contains

an RSL editor with an integrated transformation engine (code generator)¹ and a transformation development environment.² Together with this set of tools, the book supplies the reader with what it promises: the means to get very quickly from requirements to code (i.e. “in a snap”).

The transformations described in this book focus on processing two main types of requirements: functional requirements and vocabulary requirements (domain definitions). The reader will notice that quality requirements (or: non-functional requirements) are left aside. We do not provide a specific notation or semantic rules for them, but we discuss their influence on the final system architecture [23]. This is definitely a very interesting topic that deserves further intensive research [55, 90]. However, currently we need to assume that the quality requirements are specified using informal natural language. Such specifications are taken into account by the transformation developers when writing the transformation programs.

Who is this Book for?

When writing this book, we concentrated on presenting many technical details of requirements modelling and model transformations for requirements. This should make the book suitable for researchers, graduate students and practitioners from the industry. Researchers will find insight into possible research directions that stem from the presented approach to MDRE. Students and practitioners will find knowledge and practical techniques in several areas, including general requirements engineering, architectural design, software language construction and model transformation.

Our main goal was to present a comprehensive approach to MDRE that leads beyond the current state-of-the-art and state-of-practice. We are convinced that the presented technologies form good grounds for a very interesting field of research and innovation. This new field could concentrate on overcoming the accidental complexity of software through moving development efforts from 3GL programs towards formalised requirements models. The results presented in this book are meant to encourage the readers to join the effort of building research fundamentals for such a next-generation development framework. This effort encompasses research on new ways to code application and domain logic at much higher levels of abstraction. The book already presents some of the solutions that involve semantically precise scenario notations and coherent development of domain models.

The research efforts can lead to important innovations in the area of software development tools. This area seems to be in stagnation in terms of innovative features and support for software developers. This particularly pertains to

¹ The ReDSeeDS tool is presented in Sect. 7.1 and can be downloaded from <http://www.redseeds.eu/>.

² The MOLA Tool is presented in Sect. 6.1 and can be downloaded from <http://mola.mii.lv/>.

requirements engineering tools [50]. Evaluations of such tools made more than 15 years ago [182] and recently [30] show little (or: practically no) progress in terms of automatic handling of requirements and assuring their formal precision. In general, it can be noted that software development tools generally do not form a coherent framework that integrates requirements directly with implementation activities [71]. This is mainly because of clearly visible mismatch between representations for user requirements and software requirements versus representations for architectural and design models and code. This book aims at changing this situation, by proposing a tooling environment that offers the means to bridge this gap through automatic transformations.

Our second goal was to show how the techniques of MDRE could already now support and increase productivity and quality in a typical software engineering project. What is important, the presented techniques can be used in various contexts, for different purposes. Requirements engineers will certainly benefit from the presented systematic approach to formulating requirements. RSL, as a notation, can be used even without the tooling environment to keep high quality of requirements, which includes precision, coherence and unambiguity. When the appropriate editor is applied, the RSL notation gains additional support through syntax checking and automatic domain model synchronisation. The editor can also support using requirements-based patterns, thus increasing productivity in creating high quality requirements models.

The presented techniques will also benefit software developers (architects, designers, programmers). They will be able to work out their models and code at a significantly higher level of abstraction—close to the problem domain. The developers will be able to work on semantically precise (code-like) requirements models in close cooperation with requirements engineers and end-users. This book shows how they can be then relieved from caring about many of the “accidental” aspects of software development which are encapsulated in the automatic transformations from requirements to code. What is important is that the transformations always generate high quality code with uniform architecture, well documented with the generated UML models.

Apart from supporting these traditional roles in software engineering, the book promotes the role of transformation engineer. This book provides the grounds for readers interested in constructing and evolving model transformations, specifically those operating on requirements. These skills are becoming very important in the current world of changing software technologies. Having fast changing targets of the transformations and code generation, we need skilled transformation developers. The book concentrates on applying MDSD to requirements models but the presented techniques can be used for any kind of model transformation task.

Finally, the book can benefit project managers through defining a clear path from requirements to code. The managers receive guidelines on how to efficiently organise software development effort around automatic model transformations from requirements to code. This includes iterative development of software applications with evolving functionality and with evolving implementation technology.

Recommended Prerequisites

This book assumes at least some basic to intermediate knowledge of various aspects of software engineering. The overall goal was to maintain it accessible for those not yet familiar with MDSD, SLE or requirements modelling. At the same time, the book goes far beyond the basics and covers several research-level topics and shows possible research directions.

Generally, it is assumed that the reader is familiar with the software development process and its phases like requirements specification, design and implementation. Thus, it is recommended that student readers have already taken a course in Software Engineering Fundamentals or similar, or alternatively—study a good book on that topic [132, 159]. The book frequently presents UML and UML-like diagrams. The reader is thus expected to understand some of the commonly used UML notations: class and object models, interaction models, activity models and use case models. Good understanding of UML syntax and semantics of these five model types is highly recommended. This is part of any good course or book on UML [24, 52, 128].

The book includes many examples and specifications that use Java or Java-like code. It is thus recommended that the reader knows at least fundamentals of Java programming, accessible through various courses and books like the widely known “Thinking in Java” [42]. Readers familiar with other similar languages like C# or C++ should also find the code parts easy to understand. Knowledge of imperative programming (like in Java or C#) will be also needed to understand model transformation programs. These programs use graphical notation similar to UML’s activity diagrams.

The book treats the topics of use case development, software language development and metamodeling. Readers familiar with them should find certain parts of the book easier to understand. However, the book aims at explaining these topics also to unfamiliar readers.

Structure of this Book

The book is divided into eight chapters with two appendices. The first two chapters present the main concepts and give an introductory guide to requirements modelling in RSL. The next two chapters concentrate on presenting RSL in a formal way, suitable for automated processing. Chapters 5 and 6 concentrate on model transformations with emphasis on those involving RSL and UML. The transformations are presented using the model transformation language called MOLA. Chapters 7 and 8 provide a summary in the form of a systematic methodology with a comprehensive case study. The book is supplemented with two appendices containing short summaries of RSL and MOLA notation.

Chapter 1 introduces the main concepts of the book. It presents rationale for formulating requirements as precise models and explains approaches to use such models as “first-class citizens” in the software development process. This chapter also gives an introduction to MDRE as a new but already established and promising branch of Model-Driven Software Development.

Chapter 2 explains how to formulate precise requirements using RSL. It presents all the relevant RSL constructs using an example specification. All the important details of the RSL concrete syntax are outlined and its usage shown in practice. Also, good practices in formulating requirements using RSL are given. The main purpose of this chapter is to provide an RSL tutorial for requirements engineers and software developers. It should also be read as an introduction to reading the next chapter on the RSL metamodel.

Chapter 3 gives a more formal definition of RSL forming the basis for performing transformations from RSL to other modelling languages (mostly UML) and code. An important purpose of this chapter is to give the reader practical introduction to metamodeling. Consecutive sections present the language’s abstract syntax in the form of a metamodel. For each of the syntactic elements, concrete (visual and/or textual) syntax is presented through examples. The explanation of the syntax is supplemented by informally presented semantics (meaning) of the various language elements. At this level, semantics is given in reference to the concepts of business modelling and requirements engineering.

Chapter 4 presents the translational semantics of RSL using Java. This consists in a set of rules that translate RSL constructs into equivalent Java constructs. In order to construct the rules, a specific architectural framework (pattern) with specific technological assumptions (UI framework, data passing model, etc.) is chosen. This framework is expressed in plain Java for all of its components. The selected target code structure is used to explain RSL runtime semantics (semantics for executing RSL specifications). Based on this, the translation rules are generalized in order to be applicable to various technological contexts. This chapter should be read by RSL developers to understand precisely RSL semantics for the working system. Moreover, this chapter is important for transformation engineers. It shows the initial steps in designing a transformation from RSL to a specific technology.

Chapter 5 presents an introduction to model transformations. It explains intricacies of operating on models which are nonlinear graphs in contrast to text (cf. programs) which is linear. The whole presentation in this chapter is accompanied by various examples from the simple “Hello world” to more advanced transformations using the MOLA model transformation language. This chapter should give the reader the basis to understand model transformation and develop non-trivial transformation programs.

Chapter 6 extends the previous introductory chapter with guidelines for developing complex transformations based on complex metamodels. The basis for this is the RSL metamodel, the UML metamodel and the syntax of Java. This chapter thus also gives some more details of the UML formal specification and the Java syntax. For the chapter to be practical, the presentation is backed by a short introduction to a MOLA development environment. After reading this chapter, the reader should be

able to understand the rules for developing complex transformations originating in requirements models, and be able to tailor them to specific target technologies (including those emerging in the future).

Chapter 7 summarizes the book by offering a methodology and tools to apply its contents in practice. It gathers all the presented elements and places them in a coherent software development framework. This framework complies with modern iterative approaches, including agile software development. Software project managers and software developers will benefit from this chapter by organizing MDRE-based software projects according to its guidance.

Chapter 8 presents a comprehensive case study. Within the study, a specific target platform with detailed technology solutions is applied. The case study involves an example requirements model in RSL. The chapter presents some details of the model and the system (UML, Java code, UI layouts) generated from this model. It also discusses important details of the transformations that lead to generating this system.

Finally, there are two appendices. Appendix A offers a short reference of the RSL concrete syntax and Appendix B does the same for MOLA. Each of the syntactic elements is shortly summarized and an example given. The aim of the appendices is to provide the language users (requirements engineers, software developers) with an easily accessible and complete reference of the syntax.

Acknowledgments

The tool suite that accompanies this book has been developed within two projects partially funded by the European Union: ReDSeeDS³ and REMICS.⁴ We would like to thank all the Partners that have cooperated with us within these projects, especially in developing the RSL language and the ReDSeeDS tool.

Warsaw, August 2014

Michał Śmiałek
Wiktor Nowakowski

³ <http://www.redseeds.eu/>.

⁴ <http://www.remics.eu/>.

From Requirements to Java in a Snap
Model-Driven Requirements Engineering in Practice
Smialek, M.; Nowakowski, W.
2015, XXIII, 352 p. 295 illus., Hardcover
ISBN: 978-3-319-12837-5