A modeling environment supporting the co-evolution of user requirements and design

Citation for published version (APA):

Document status and date:
Published: 01/01/2014

Publisher Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 05. Apr. 2020
A Modeling Environment Supporting the Co-evolution of User Requirements and Design

Frank Peeters, Ion Barosan, Tao Yue and Alexander Serebrenik
A Modeling Environment Supporting the Co-evolution of User Requirements and Design

Frank Peeters¹, Ion Barosan², Tao Yue³ and Alexander Serebrenik²

¹Fontys University of Applied Sciences, Eindhoven, The Netherlands  
  f.peeters@fontys.nl  
  ²Eindhoven University of Technology, Eindhoven, The Netherlands  
  {i.barosan,a.serebrenik}@tue.nl  
  ³Simula Research Laboratory, Oslo, Norway  
  tao@simula.no

Abstract. One of the goals of Model Driven Engineering (MDE) is to enable automation. It is, however, a challenge, when pursuing this automation, to include textual requirements as a model. It is even more challenging to maintain the consistency of requirements and design models, when changes are made to one of them during an iterative process. In this paper, we propose a modeling environment, built as part of the Symbiosis framework, to enable the continuous synchronization between textual requirements and the core design model: Object Model. The co-evolution is successful due to a fact-oriented approach. We performed two case studies to evaluate Symbiosis. Results show, that it does not take much effort to follow our methodology with the Symbiosis tool support, to derive a syntactically and semantically correct and complete design model that fully conforms to textual requirements.

Keywords. Requirements, Natural Language, Fact Oriented Modeling, Model Driven Engineering, Object Model and Co-evolution.

1 Introduction

Requirements are typically written in natural language and therefore ambiguity is unavoidable. Transitioning from textual and ambiguous requirements to design (e.g., represented as a design model) is usually done manually by domain experts based on their tacit knowledge. Automated approaches (e.g., [11, 13, 14]) have been proposed in the literature to ease this process; however to obtain a correct and complete design model, most of these approaches rely on the manual refinement of an automatically generated design model from requirements. The manual refinement of the generated model requires additional effort from users having knowledge on modeling notations of the design model, and needs a round trip solution to ensure the synchronization of the design model and the requirements after the refinement is performed. These challenges hinder the practical application of these approaches.

In this paper, we present an approach, called Symbiosis, addressing the same problem by synchronizing requirements and design while constructing and generating the
design models. In other words, we enable the co-evolution of requirements and design. As shown in Figure 1, Symbiosis starts with modeling requirements, in contrast with typical model driven approaches, where requirements are the input of these approaches. We have incorporated requirements as a model (Requirement Model (RM)) within our framework. A stakeholder or a member of the software developing team (Participant) has to enter requirements manually. Every requirement should be approved by the Project Owner and every approved requirement should be realized within one of the other two models (ObjectModel (OM) and UseCaseModel (UM)). To get a picture on the progress of approval and realization it is important that the progress in the validation and realization of the RM can be monitored.

During the semi-automated construction of the OM, various design decisions are made, for instance: 'What could be a unique identification of a certain object?'. 'Do we need to consider a certain value as an object or is it satisfying to register such a value just like a string or number?'. Design decisions that require a validation by the product owner will trigger an automatic feedback to the RM.

An OM contains enough information to generate a class diagram, what represents concepts of the target domain within the scope of the RM. We claim that the complete (in the sense of implementing all the requirements in the RM) source code of all domain classes, including all needed and nice-to-have operations, can be generated and stored in the Domain Class Library (DL).

The upper part in the framework scheme concerns the creation of use cases: the refinement of action requirements into use cases within the UM and the generated source code of the user interaction with the system into a UseCase Class Library (UL). The ActionCase Class Library (AL) plays a prominent role, while refining and realizing the action requirements in use cases. Every class in the AL represents one,
so called, ActionCase, what defines the normal flow of one rather tiny use case, including all exceptions that prevent the execution of the normal flow. The design of one use case of the UM comes down to an assembly of a set of solid action cases, extended with the realization of the required quality attributes such as security, usability and reliability. The AL acts like a toolbox, which is completely generated based on the OM.

In this paper we limit our scope to RM, OM, the relations between RM and OM (i.e., breakdown, assignment and continuous synchronization) and DL. Design and realization of quality attributes and other layers, as for example the (graphical) user interface and a persistency layer, are out of the scope of this paper as well.

The rest of the paper is structured as follows. In Section 2, we discuss the related work. We discuss our approach in details in Section 3, followed by the presentation of the tool architecture (Section 4). We evaluated our approach with two case studies and results are reported in Section 5 with an overall discussion in Section 6. The paper is then concluded in Section 7.

2 Related Work

Our approach is inspired by fact type based modeling methodologies like Object Role Modeling [6], and FCO-IM, Fully Communication Oriented Information Modeling [1]. These methodologies have proven [5] their practical value in large projects related to corporate conceptual data models or design data warehouses. While fact type based modeling originally focus on databases and data warehouses, our aim is to build software [10]. Semantic of Business Vocabulary and Business Rules (SBVR), standardized by the Object Management Group [9], introduced the use of a vocabulary, facts and rules to formalize the domain concepts and their relationships. We use a similar approach, except the way the rules are managed. SBVR prescribes a separate rules engine, which constrains the intended software system. On the contrary, we incorporate every rule into one responsible object type in the OM, as part of the 'assignment' step in Figure 1. The locality of an object type seems to offer an easier and better manner to maintain rules.

A fact type oriented information model or an entity relationship diagram can be represented with the Unified Modeling Language (UML). In literature there exist mappings of how individual fact types can be mapped onto UML constructs. Bollen [2] designed an algorithm to transform a fact type oriented information model onto a class diagram without violating the existing constraints within the information model. The transformation is restricted to the static aspects of a class diagram. Definition of constructors and methods (including getters and setters of properties) are not taken into account, which is different from our approach.

Most of the time, the software requirements are specified in Natural Language (NL); however, building models from NL requirements is difficult and time-consuming. Deetimahanti and Sanyal [4] have presented a semi-automated technique to assist developers in generating UML based analysis and design models from normalized NL requirements called UMGAR. Their approach is based on three NLP techniques for parsing sentences and performing various analyses, but currently
UMGAR still requires human interaction during the elimination of irrelevant classes and identification of aggregation/composition relationships among objects. Our OM contains enough information necessary to generate a complete domain class library; editing afterwards is not needed.

Use cases are one of the most common instruments for capturing functional requirements. There are many use case modeling approaches, defined in the context of Model Driven Architecture (MDA), in which requirements are translated into an analysis model. In general, the use cases are still specified in textual form and this can inevitably introduce ambiguity. To facilitate the transition towards analysis models and to reduce the ambiguity of use case specifications, Yue, et al. [15] has proposed 26 well defined restriction rules and a use case template. Their experiments indicate that the restriction rules are easy to apply and with appropriate tool support, error rates will mostly decrease. Also, they have shown that their approach results in a significant improvement regarding the correctness of derived class diagrams. However, there are still concerns about the completeness and redundancy of classes. Rosenberg and Stephens [12] propose ICONIX as a Use-Case driven iterative process with steps that are based on industrial empirical experience of experts in UML analysis and design. We however, construct a use case model based on the OM instead of the reverse way. This is possible because information required generating use cases are contained, structured and maintained in the OM, which is nicely synchronized with the RM. In other words, use cases are another representation of the information originally contained in the RM, later on in the OM. Considering the benefits of the continuous synchronization process of the RM and OM, we expect our approach can generate a higher quality of use cases than these related works. In addition, we obtain a higher quality of the OM.

3 The Symbiosis Methodology on Co-evolution of RM and OM

High quality software development begins often with a thorough business study, which includes an analysis of weaknesses, opportunities, challenges, risks and goals with respect to intended information system(s) [7]. Eventually, this analysis will lead to the definition of business requirements. These requirements are described from a business point of view and are reflected in the user requirements specification (URS), which we specify as a RM by following a requirements modeling methodology that will be discussed in Section 3.1. The Symbiosis methodology starts with constructing a RM; business requirements are out of scope of our research.

As shown in Figure 1, the Symbiosis methodology starts with the input of requirements from participants such as external stakeholders and requirement engineers. We recognize four requirement types: actions, facts, rules and quality attributes, which form the RM. Every requirement needs to be validated by the product owner (Section 3.1), an external stakeholder who is responsible for that requirement. After that, fact requirements approved by the product owner are broken down into the OM (Section 3.2) and the tool synchronizes the OM with the RM. Software engineers then need to configure the OM (Section 3.3) for the purpose of adding necessary constraints, permissions and facilitating responsibilities of object types. After that Symbiosis auto-
matically generate domain classes (Section 3.4), followed by the realization of manually added rules into the OM (Section 3.5).

![Figure 2: The Early Stages in the Symbiosis Process](image)

When additional requirements are added or changed during the Fact Breakdown and the Type Configuration steps, such changes are always marked as a proposal to the product owner. All changes in the RM require a new traversal through the Symbiosis process. The process flow in Figure 2 shows the linear traversal through the early stages in this process. When the state of RM changes, this traversal can be repeated partially, until eventually all fact and rule requirements are approved and realized in the OM. The round trip relationships between the last three steps will be discussed in Sections 3.4 and 3.5.

For a better understanding of our approach and its artifacts, we sometimes need to descend to a more concrete level. We select a multi-player game Airhockey as the running example. Three internet players are interacting on a triangular board. Every player uses a bat to defend her/his goal against the moving puck. A player can move his bat in front of the opening of his goal. The puck is moving around within the board and bouncing against the borders and bats. The example case is rather easy to understand, contains enough object oriented concepts and cannot be implemented easily as a database application.

### 3.1 Requirement Preparation

During the process of gathering requirements, every requirement is classified as one of these four categories: Action, Fact, Rule and Quality Attribute.

**Action** is defined as an intended action (task), which needs software support. Action requirements form the preparatory step towards use cases. For the Airhockey running example, we can derive action requirements including "Everybody can sign up as a registered user at the Airhockey website"; "A visitor can create a new game"; and "A player can chat during the game and a visitor can chat at the Airhockey site".

Every action is linked to specific information, sometimes as an input or sometimes as a result of the action. Information is considered as a collection of representative **facts**. For reasons of simplicity, we define facts on a concrete level, instead of on a type level, for instance: "The 4th score of Celine is 12.". Expressing in concrete example facts is easier and less error prone. There will be a smaller risk introducing wrong information or ambiguity, compared to facts on a type level [6]. The mentioned fact on a type level could be: "The \(<\text{nr} : \text{Natural}>\)th score of \(<\text{User}>\) is \(<\text{score} : \text{Natural}>\).". Expressing the fact type variant, requires extra skills of abstraction. However, the detection of the objects and other values, and the assignment of type names cause unwanted distraction and the product owner should not be bothered for that. Moreover, concrete facts can play a decisive role during the design of representative
test cases in an automated way. At last but not at least, fact requirements are rather easy to formulate because there is no strict regulation about the syntactic structure of a fact in terms of the ordering of subject, verb, nouns etc.

Every fact has to be context free. That means, one individual fact has to be clear without knowledge of any other fact. For example, Fact “David3 is spectator at this game.” is not a context free fact, because “this game” refers to a specific game mentioned in another fact, for instance a fact about "the game of William86". Context free facts can be handled and changed in a more robust way. Unique identification of mentioned objects (e.g., ‘Game’, ‘Spectator’), within one fact, plays an important role when formulating unambiguous facts. For instance, if we want to discriminate in an unambiguous manner between all chat messages of one chat room, we need an appropriate discriminator: the combination of the chat room and the, discriminating, order of arrival at that particular chat room makes a message unique ("the 34th message at the game of William86"). No two messages have the same combined values of the referenced chat room and the order of arrival.

Rules are defined as constraints and other business rules with respect to facts and actions. For example, these two rules can be specified for the Airhockey case study: “A game can be started as soon as the third player has joined the game”; and “A visitor of the Airhockey site is not allowed to play in more than one game at the same time”.

Quality attributes are a type of requirements that express non-functional attributes with respect to one or more functional requirements specified as action, fact or rule. Investigating non-functional requirements is outside the scope of this paper.

3.2 Fact Breakdown and the Object Model

The next stage after the Requirement Preparation concerns the Fact Breakdown, which maps the relations between the objects into the Object Model (OM). Before we discuss the fact breakdown in more detail, we first describe the OM.

![Figure 3: Meta-model of the OM (Simplified Version)](image)

Figure 3 presents a simplified version of our OM meta-model. An OM is composed of zero to many FactTypes, each of which can play zero to many Roles. For example, fact "The 4th score of Celine8 is 12" can be specified as a FactType about scores of users: ScoreUser: The <nr : Natural>th score of <user : User> is <score : Natural>. So we attach the name 'ScoreUser' to it. The first and third roles are played by a natural number and the second role is played by a 'User' object. Every Role has a name
and a substitution type. Included ObjectTypes all refer, in one way or another, to real objects in the specified domain.

There are fact types that are ObjectType and FactType in the same time. The next two expressions represent the same information: “There is a game with captain Celine8.” and “the game of Celine8”. Both expressions reveal the fact that a very specific game exists. Therefore, the type name of both expressions should be equal. However, the second expression does possess an extra quality: it is ready to be substituted in other facts such as "Robby participates in the game of Celine8.". The first expression we call a fact expression and the second one we call an object expression. The object expression is truly a fact expression but with the extra capability to be substituted elsewhere. Thus, an object expression could be considered as a subtype of a fact expression: talking about an object is talking about a fact. Similarly, an ObjectType is a subtype of a FactType (Figure 3). This unifying approach has been recognized firstly by Bakema et al. [1].

An ObjectType can be extended, just like a UML class, with zero to many operations like a constructor, properties and methods (Figure 3). Most of these operations can be derived from the properties of the object type (see Section 3.4.4). A BaseType represents types of values such as numbers (Natural, Integer, Real), String and Boolean. Base types are the terminal types of the system; a further breakdown is not needed. SubstitutionType generalizes ObjectType and BaseType. Only substitution types may used within the role of a fact type.

There are two kinds of constraints. A StaticConstraint is a restriction on the allowed substitution values in a role or a fact type as a whole. A DynamicConstraint concerns allowed state transitions or permissions to update the state of the system.

An OM can be transformed into a UML class diagram. At this moment we use nine transformation rules (implemented as part of the Symbiosis framework) to map an OM into a class structure. Reporting this transformation in details is however out of the scope of this paper.

It is though worth emphasizing that a class diagram does not reveal all the information of the OM. Due to the more fine granulated structure with different sized fact types, determined by the number of roles contained, and related constraints, it is preferable to express the conceptual design in fact types instead of classes with their relations. The holy grail of our system is located in the explicit introduction of the Role construct. Restrictions and permissions can be assigned to one specific role. UML doesn't offer role assignments in such a formal way.

The process of fact breakdown aims in the first place to identify the roles that object and base types play. The fact breakdown is supported by the Symbiosis tool and user input is required during the process: Every role must be mapped to a substitution type, and, if required or desired, a role name. A base type substitution is always accompanied with a role name, but in case of an object type substitution there is sometimes no need to clarify the specific role that an object type plays. For example, the result of a fact breakdown can be: Participation: "<Player> participates in <Game>." and Game: "the game of <captain : Player>".

There are no explicit role names mentioned in the two roles of the fact type 'Participation'. In contrary, the 'Player'-role in the 'Game' object type is made clearer with the
introduction of the 'captain' role name. An absent role name ends up in a role name that is equal to the name of the object type.

The baptizing of fact types, object types and roles (e.g., 'Participation', 'Game', 'captain') is handwork. Object types that are already part of the OM can be traced by name during the fact breakdown.

It is possible to automate the breakdown process in a more complete way; however, a fully automatically generated object model would not be fully complete and correct, as discovered by Yue, et al. [14] in a slightly different context: generating class diagrams from use case models. We therefore, take a different approach by involving users during the fact breakdown process instead of refactoring incorrect and incomplete models afterwards. Our future work will compare these two different processes via empirical studies.

While executing the fact breakdown, the OM grows with fact types and related object types. Every fact requirement is mapped into a fact type. In case of a new fact type, the OM is extended with this fact type. Often such a fact type leads to the discovering of new object types. In that case, the OM will grow further with all new object types that play a role within this fact type. We want to emphasize that the object model is growing by adding fact types instead of object types. For example, during the breakdown process of “The 4th message at the game of LittleDevil is sent by Irene.”, one fact type (SenderMessage: <Message> is sent by <sender: Visitor>) and five object types (Message: the <nr: Natural>th message at <Game>; Game: the game of <captain: Player>; Player: <Visitor>; Visitor: <User>; and User: <name: String>) will be revealed.

We want to mention that the process of fact breakdown is not deterministic. Sometimes, design decisions, like inheritance relations between two object types, made during a breakdown, can be reconsidered. In general, refactoring of the OM should be an essential part of an object modeling methodology, but always in such a way that the complete state of the OM remains synchronized with the RM.

The structure of the object types is constantly a rather coherent graph: every object type is from the very beginning related to the roles they play in other fact types. The only way to extend an OM with a new object type, is by executing a fact breakdown which reveals an object of a type which is still not present in the OM. On the contrary, most of general object modeling methodologies start with the detection of classes and try to complete the class structure with necessary associations. Due to the missing link to the requirements, there is less evidence on completeness or redundancy of UML associations. After the fact breakdown process is finished, we continue imposing constraints on the current type set with fact and object types.

3.3 Type Configuration

The fact breakdown results in a fact type related network with default constraints. Unique identification of object types belongs to the default constraints, but there are more. During the type configuration we are concerned about necessary extra constraints such as unique role combinations within fact types, mandatorial roles, value constraints, qualifiers, default values, derivability etc. Possible communication be-
tween objects, permissions, responsibilities that restrict the dynamic behavior are of interest too; for instance, do we need facilities to add, insert, update or remove facts? To illustrate the constraints configuration, we consider the fact type example ScoreUser: "The \(<\text{nr} : \text{ScoreNr}\>\)th score of \(<\text{User}\>\) is \(<\text{score} : \text{Natural}\>\).", where role and object type constraints are specified between curly braces:

- role 0: \(<\text{nr} : \text{ScoreNr}\>\) \{\text{u1, seq}\}
- role 1: \(<\text{User}\>\) \{\text{u1, f:5, com}\}
- role 2: \(<\text{score} : \text{Natural}\>\) \{\text{d:3}\}
- \text{ScoreNr:} \(<\text{value} : \text{Natural}\>\) \{\text{v:1..5}\}

Uniqueness constraint u1 claims a unique value combination on role 0 and 1. In this case, it is impossible for two different facts to have the same combined value with respect to the first two roles. Assume, we have two facts: "The 3th score of Chompy is 7." (f1) and "The 3th score of Chompy is 6." (f2). The uniqueness constraint u1 does not allow the existence of f1 and f2 at one moment. Frequency constraint f:5 demands five 'ScoreUser'-facts per 'User'. The value constraint of 'ScoreNr', defines the range of the index of the score (1..5). The \{\text{d:3}\} at the last role of 'ScoreUser' specifies that a score has a default value of 3. This value can be used when the user still does not finish enough games. Finally, the user acts as composition on behalf of the 'ScoreUser'-facts (see 'com' at role 1).

During the stage of the fact breakdown a large part of the type configuration gets an automated default setting. A lot of constraints are more or less predictable, which saves a respectable amount of manually configuration time. For example, during the fact breakdown, the uniqueness constraint for the roles of an object type is added automatically. This is because object expressions needs to be unambiguous. However, a redundant identification should be avoided. For example, the next object expression does not align with this rule of efficiency: "the 3th message at the game of Doreen written by William86". In this case it is over identified; it is not needed to identify the message by specifying the number, the game and the author of the message. That’s why object types are always go along with a uniqueness constraint spanning all roles.

A software team member can correct the default settings and add other constraints and permissions. The creation or change of a default or manually added constraint in the OM will result automatically in an update of a specific rule requirement. Every default or manually entered permission in the OM causes the creation of a new action requirement. For instance the 'u1' of 'ScoreUser' results subsequently in one generated rule requirement: "Two different facts about "The \(<\text{nr} : \text{Natural}\>\)th score of \(<\text{User}\>\) is \(<\text{score} : \text{Natural}\>\)," with an equal combined value on \(<\text{User}, \text{nr} : \text{Natural}\>\) is not allowed."

Some of the design decisions during fact breakdown or type configuration do not need an external approval. Decisions related to inheritance, objectifying, communication, responsibility and object registries are concerns with the interaction between the software artifacts. The product owner should not be bothered about these design concerns. All other changes are reflected in the RM and need validation by the responsible product owner.

Type configuring is an iterative process that is triggered by three inputs. The first validation is based on the view on the fact types with its roles and constraints. The
second validation is the report of the quality scan of the OM (Section 3.4.1). The report contains errors that need to be fixed and warnings that need to be considered. Normally, the scan asks for a type refactoring of the OM. The generated operations of the classes act as a third input to the type configuration (see feedback arrow in Figure 2). The software engineer needs to validate the classes and its operations. This validation could demand for type refactoring too.

3.4 Automated Construction of Domain Class Libraries (DL)

To generate the DL the following steps, presented in Figure 4, are executed. These steps will be briefly introduced in the next subsections.

![Figure 4 Automated Construction of Domain Classes](image)

3.4.1 Scan on Meta Rules

An OM contains enough information that can be used to generate a static class structure. However, not every OM is ready for such a transformation. There are some preconditions that a well-formed OM needs to fulfill. We call them meta rules. Until now, practical experience has revealed six meta rules. However, future research may yield new findings about extra meta rules. To illustrate the meta rules, we present two of them.

The first meta rule is that an OM needs to be elementary, which demands that all its components (fact types), are elementary. The uniqueness constraint (uc) determines the elementariness of a fact type. Every object type possesses a uc which spans all its roles. Thus, to determine whether an OM is elementary, a check of only the non-objectified fact types is required. We define the elementariness of a fact type as follows: A fact type, with \( n (n>1) \) roles, is elementary if the smallest uniqueness constraint spans at least \( n-1 \) roles [6].

An example of a non-elementary fact type, with in this case three roles, is: `<Message> reads "<text : String> is sent by <sender : Visitor>.' Due to the functional dependency between 'Message' at the one hand side and 'text' and 'sender' at the other hand side, the single 'Message'-role is restricted by a uc. This uc spans just one role of a fact type with three roles: the \( n-1 \) rule is violated. In this case, we can get rid of this violation by splitting up the fact type into two separate fact types: "<Message> reads "<text : String>.", and "<Message> is sent by <sender : Visitor>"."

A second meta rule concerns about the responsible roles of a fact type. A responsible role is played by an object type that is able to create, add or update a fact of a certain fact type. To prevent responsibility conflicts we introduce the second meta rule: Every fact type contains at most one responsible role.

The other four remaining meta rules are mentioned shortly: an object type with two parent object types is not allowed, compositional cycles are forbidden, every fact type must have a navigable role and finally, prevention of dangling objects is needed.
3.4.2 Creating Registries
We aim to design a domain model with enough access points to the relevant domain objects. Every object type that is not accessible gets a composition relation with a new singleton object type. An object of an object type is accessible if there always exists another object that is aware of the presence of the first mentioned object. Otherwise, such an object could not be retrieved from outside the domain layer. This problem can be resolved with the help of additional registry object types.

3.4.3 Creating Classes
A well-formed OM should satisfy the current six meta rules, otherwise a transformation towards a class structure could be impossible or it will end up with an incorrect result.

The mapping of an OM to a class structure is driven by eight transformation rules, which yield classes, attributes, associations, multiplicity and inheritance relations. To illustrate the transformation rules we mention only two of the nine transformation rules. The first transformation rule is obvious: 'Every object type is converted into a class.' The second one is about fact types with one role: 'A non objectified fact type with one role, results in a Boolean attribute of the class referred by the role type; the name of the attribute is derived from the name of this fact type.'

The other remaining transformation rules are related to the creation of properties, multiplicity, qualifiers and compulsory objectified fact types.

3.4.4 Creating Operations
One of the goals of the methodology is to produce the source code for all domain classes. In the previous sections we only looked at the structural dimension of an object. Now we focus on the behavioral dimension of an object: constructor, properties and methods. Larman [8] made a list of relevant patterns used when designing object oriented software. One of his General Responsibility Assignment Software Patterns, abbreviated GRASP, is about design of the behavioral aspects of an object. It concerns the Information Expert principle. We applied the Information Expert principle to allocate the behavior based on the properties of an object. The properties of an object are defined through their involvement within the navigable roles of fact types.

3.4.5 Completing Source Code
The OM contains enough information to generate the complete source code of a class: data fields, bodies of all operations and import instructions. A library can be easily generated. It is even possible to inject Object Relational Mapping instructions into the source code. There are situations where algorithmic specification must be included manually into the OM. The implementation is explained in the next section.

3.5 Manually Added Rules into the OM
There are two types of rule requirements. The first type is rule requirements that are generated during fact breakdown or type configuration. The other rule requirements are manually added. However, some of the generated rules coincide with the manually added ones. The redundant manually rule requirements should be removed. After
this removal we consider three categories of rules: condition rules, derivability rules and event rules. A condition rule restricts when an action requirement is applicable. A derivability rule describes a computational process. An event rule specifies an event, event source and event handler. Sometimes, these rules need to be formalized with the help of manually added algorithms within an IDE.

The generated classes should be considered as a volatile snapshot on the current OM. Every time we generate these classes, the old ones are overwritten. Therefore, in case of manually added algorithms we save the algorithms separately in the OM (see feedback arrow in Figure 4).

4 Automation

The development of the Symbiosis tool\(^1\) started in 2011 with the design of the meta-model of an OM. Originally, this meta model has been designed conform the Symbiosis methodology itself, as a proof of concept. The entire tool is realized with the platform independent Java technology. The realization of a Project Layer, as a backbone for the RM and OM, became the next step (Figure 5; the numbers are aligned to the numbers within Figure 2). Afterwards we constructed a Graphical User Interface (GUI) prototype. It is the front end for users to complete user-interactive functionalities: Requirement Viewer, Fact Breakdown and Type Configurator. The Domain Class Generator (step 4) is part of the OM-engine.

Figure 5: Architecture of Tool Support

The Symbiosis tool interacts via the file system with a text editor and an IDE. The text editor is used, as a first step, for preparing and organizing the requirements. The IDE makes it possible to transform algorithms of manually added rules into computer source code, based on the Domain Class Library. Eventually, the manually added source code is fed back into the OM. The development of the ActionCase Classes (step 6) is planned to be yielded in 2014.

\(^1\) www.equaproject.nl
5 Evaluation

In order to verify the proposed methodology, two case studies were performed in this work. In the rest of the section, we first provide a brief description of each case study, followed by the execution of the case studies; the handling of manually added rules into the OM is not included. The results are presented and discussed.

Airhockey: Three internet players are interacting on a triangular board. Every player uses a bat to defend her/his goal against the moving puck. A player can move his bat in front of the opening of his goal. The puck is moving around within the triangle board and bouncing against the borders and bats.

Crisis Management System – CMS: CMS [3] was designed to help in identifying and handling a crisis such as earthquakes, tsunami, floods, terrorist attacks, and accidents. CMS provides ways for communication and coordination among different coordinators. CMS supports its users in allocating necessary resources to handle a crisis and provide access to relevant crisis information.

Table 1 Required Effort (in hours) to Perform the Key Steps of the Case Studies

<table>
<thead>
<tr>
<th>Requirement Preparation</th>
<th>Fact Breakdown</th>
<th>TypeConfig Initial</th>
<th>TypeConfig Refactoring</th>
<th>Domain Classes</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airhockey</td>
<td>1.0</td>
<td>0.5</td>
<td>0.5</td>
<td>0.0</td>
<td>2.5</td>
</tr>
<tr>
<td>CMS</td>
<td>3.0</td>
<td>1.0</td>
<td>1.0</td>
<td>2.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Both case studies were executed by two of the authors, who played the role of being the software engineer and product owner. The required effort, to perform the key steps of the methodology for the two case studies, is reported in Table 1. Table 2 shows that CMS has a larger scale than Airhockey in terms of the initial number of requirements (RM:1,2,4,5,6). More effort was required to prepare the RM and to derive the OM for CMS than Airhockey.

During the requirement preparation phase, the description of each case study system was analyzed by the methodology experts. The elicited requirements were categorized according to the methodology described in Section 3.1 as the initial version of the RM. Since the original requirements specification of CMS does not contain fact requirements, we introduced them to form a complete RM. The entire process took one hour for Airhockey and three hours for CMS.

The fact breakdown process took half an hour and one hour for Airhockey and CMS case studies, respectively. As discussed at the end of Section 3.3, the type configuration process is divided into two phases: the initial phase and the refactoring phase. The refactoring is triggered by the results of an automated scan and a manual inspection of the generated classes. Significantly more time was spent on the type refactoring than the initial type configuration for CMS (two hours vs. one hour): too many artificial registries were generated.

Table 2 summaries the RM and OM artifacts, applied to the two case studies. Also, the table presents the characteristics of the generated classes in terms of operations, attributes, associations and inheritance relationships. The RM model contains manual-
ly added and generated action and rule requirements. Many manually added requirements are duplicated in the RM; they need to be removed by hand (RM2+RM6). Analyzing the data from Table 2, we observed that generated requirements (RM3+RM7) represent by far the majority of the requirements (RM1+RM3+RM4+RM5+RM7). In the Airhockey case 38 from 48 requirements were generated and in the CSM case 83 from 111. The normal ObjectTypes are FactTypes that are objectified, but do not belong to the Registries, abstract ObjectTypes or ValueTypes.

Table 2 Descriptive Statistics of Key Elements of the RM, OM and Classes

<table>
<thead>
<tr>
<th></th>
<th>Airhockey</th>
<th>CSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Action requirement manually</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2 Action requirement manually redundant</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>3 Action requirement generated</td>
<td>12</td>
<td>31</td>
</tr>
<tr>
<td>4 Fact requirement manually</td>
<td>7</td>
<td>21</td>
</tr>
<tr>
<td>5 Rule requirement manually</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6 Rule requirement manually redundant</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7 Rule requirement generated</td>
<td>26</td>
<td>52</td>
</tr>
<tr>
<td>OM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Fact Types</td>
<td>7</td>
<td>20</td>
</tr>
<tr>
<td>2 Normal ObjectTypes</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>3 Artifical Registries</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4 Abstract ObjectTypes</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>5 Value Types (including enums)</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>DC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Classes</td>
<td>12</td>
<td>19</td>
</tr>
<tr>
<td>2 Total number of associations</td>
<td>17</td>
<td>20</td>
</tr>
<tr>
<td>3 Total number of BaseType/ValueType attributes</td>
<td>8</td>
<td>30</td>
</tr>
<tr>
<td>4 Total number of inheritance relations</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>5 Total number of generated operations</td>
<td>133</td>
<td>203</td>
</tr>
</tbody>
</table>

6 Overall Discussion

One of the important aspects of our approach is the incorporation of the RM, and especially fact requirements, into the modeling environment. Fact requirements are the source of the artifacts in the OM, which facilitate the co-evolution of requirements with design. Another advantage of our approach is the way we elicit requirements using the natural language in an unrestricted way.

Traditionally, the design of a class diagram starts with the gathering of relevant classes, then establishing the relations between the classes and finally populating the classes with attributes and operations. On the contrary, we start with the gathering of relations. Object types appear when they play a role within the gathered relations. In our approach, a role is semantically richer than the UML counterpart. The UML role has a type, a name and a multiplicity. Roles, we consider, can be extended with extra constraints, for instance: frequency constraint, value constraint and action permissions (add, remove, insert, set, adjust), which makes it possible to generate operations including their complete parameter list and source code. Roles are often the link between constraints and requirements, making co-evolution possible. It is important to emphasize that a UML association is a binary fact type (fact type with two roles). Fact types with zero, one or three or more roles have no counterpart in UML, making the modeling process easier and more versatile.
It is rather normal in the software development to start with capturing the behavior of the system. Mostly, this is done with the help of use cases. The domain objects are discovered later on during later conceptual design activities. The quality of the domain model can be improved through an iterative approach, but there is no certainty that the final model is correct or complete. In our approach we have control over the required elements in the OM, due to the strict coupling to the requirements. However, on the other hand, we have enough information to generate reliable and sufficient ActionCases, which are the base components for the assembly of use cases into the UM.

7 Conclusion and Future Work

In this paper, we presented the Symbiosis modeling environment, addressing the problem of synchronization between requirements and design while constructing and generating an OM and DL.

The main execution process in Symbiosis is as follows: 1) preparation and import of the requirements into RM, 2) fact breakdown, 3) initial type configuration, 4) generating class structure, 5) type refactoring, 6) assignment of manually added rules, 7) generating a draft DL, 8) editing algorithms, 9) generating AL (+DL), 10) design of UM and 11) generating UL. However, every time the RM changes it requires a validation by the product owner and a new iteration of the execution process. Currently, the most important part of the framework has been implemented: the construction of the OM, the synchronization between the RM and the OM, and the creation of a class structure. Our future research, will be focused on the following issues: the integration of the manually added rules (6 and 8), the generation of the ActionCase Library (9), the design of a UM (10) and its realization in a UL (11).

We conducted two case studies to evaluate our approach and results are very encouraging. To evaluate the efficiency and effectiveness of the Symbiosis methodology, we have planned for this year more case studies and controlled experiments.

Bibliography


2 Classes with operations of both case studies can be inspected on www.equaproject.nl
In this series appeared (from 2012):

12/01 S. Cranen Model checking the FlexRay startup phase

12/02 U. Khadim and P.J.L. Cuijpers Appendix C / G of the paper: Repairing Time-Determinism in the Process Algebra for Hybrid Systems ACP

12/03 M.M.H.P. van den Heuvel, P.J.L. Cuijpers, J.J. Lukkien and N.W. Fisher Revised budget allocations for fixed-priority-scheduled periodic resources

12/04 Ammar Osaiweran, Tom Fransen, Jan Friso Groote and Bart van Rijnsoever Experience Report on Designing and Developing Control Components using Formal Methods

12/05 Sjoerd Cranen, Jeroen J.A. Keiren and Tim A.C. Willemse A cure for stuttering parity games

12/06 A.P. van der Meer CIF MSOS type system

12/07 Dirk Fahland and Robert Prüfer Data and Abstraction for Scenario-Based Modeling with Petri Nets

12/08 Luc Engelen and Anton Wijs Checking Property Preservation of Refining Transformations for Model-Driven Development

12/09 M.M.H.P. van den Heuvel, M. Behnam, R.J. Bril, J.J. Lukkien and T. Nolte Opaque analysis for resource-sharing components in hierarchical real-time systems - extended version –

12/10 Milosh Stoljk, Pieter J. L. Cuijpers and Johan J. Lukkien Efficient reprogramming of sensor networks using incremental updates and data compression

12/11 John Businge, Alexander Serebrenik and Mark van den Brand Survival of Eclipse Third-party Plug-ins

12/12 Jeroen J.A. Keiren and Martijn D. Klabbiers Modelling and verifying IEEE Std 11073-20601 session setup using mCRL2

12/13 Ammar Osaiweran, Jan Friso Groote, Mathijs Schuts, Jozef Hooman and Bart van Rijnsoever Evaluating the Effect of Formal Techniques in Industry

12/14 Ammar Osaiweran, Mathijs Schuts, and Jozef Hooman Incorporating Formal Techniques into Industrial Practice

13/01 S. Cranen, M.W. Gazda, J.W. Wesselink and T.A.C. Willemse Abstraction in Parameterised Boolean Equation Systems

13/02 Neda Noroozi, Mohammad Reza Mousavi and Tim A.C. Willemse Decomposability in Formal Conformance Testing

13/03 D. Bera, K.M. van Hee and N. Sidorova Discrete Timed Petri nets

13/04 A. Kota Gopalakrishna, T. Ozelebi, A. Liotta and J.J. Lukkien Relevance as a Metric for Evaluating Machine Learning Algorithms

13/05 T. Ozcelebi, A. Weffers-Albu and J.J. Lukkien Proceedings of the 2012 Workshop on Ambient Intelligence Infrastructures (WAmI)

13/06 Lotfi ben Othmane, Pelin Angin, Harold Weffers and Bharat Bhargava Extending the Agile Development Process to Develop Acceptably Secure Software


13/08 Mark van den Brand and Jan Friso Groote Software Engineering: Redundancy is Key

13/09 P.J.L. Cuijpers Prefix Orders as a General Model of Dynamics
14/01  Jan Friso Groote, Remco van der Hofstad and Matthias Raffelsieper  On the Random Structure of Behavioural Transition Systems

14/02  Maurice H. ter Beek and Erik P. de Vink  Using mCRL2 for the analysis of software product lines

14/03  Frank Peeters, Ion Barosan, Tao Yue and Alexander Serebrenik  A Modeling Environment Supporting the Co-evolution of User Requirements and Design