HOCHSCHULE
UNIVERSITY OF
APPLIED SCIENCES
INGOLSTADT

Enhancing Domain Modelling with Easy to Understand Business Rules

Working Papers
Arbeitsberichte

Christian Bacherler, Christian Facchi,
Hans-Michael Windisch

HOCHSCHULE
UNIVERSITY OF
APPLIED SCIENCES
INGOLSTADT

Enhancing Domain Modelling with Easy to Understand Business Rules

Christian Bacherler, Christian Facchi,
Hans-Michael Windisch

Arbeitsberichte
Working Papers

Heft Nr. 19 aus der Reihe
„Arbeitsberichte – Working Papers“
ISSN 1612-6483
Ingolstadt, im November 2010

Enhancing Domain Modelling with Easy to Understand Business Rules *

Christian Bacherler, Christian Facchi and Hans-Michael Windisch
{christian.bacherler | christian.facchi | hans-michael.windisch}@haw-ingolstadt.de

University of Applied Sciences Ingolstadt
Esplanade 10
D-85049 Ingolstadt

Abstract. The model driven software development (MDSD) paradigm is gaining momentum in developing extensive business software applications. With MDSD it can be carried out a significant contribution towards the key factors of success which are flexibility and adherence to delivery dates as well as efficient maintenance and adaptability. With AtomsPro that is introduced here we consequently integrate aspects of domain modelling and software architecture as well as clear and understandable means to enable domain experts to take an active role in the software development process. The challenging aim is to increase the overall ratio of code generated in the development process of enterprise applications by at the same time preserving abilities for efficient maintenance. Hence, we have chosen a fruitful cooperation with several involved project partners.

1 Introduction

Preserving an enterprise's key factor for competitiveness on the global markets means that business processes and business rules have to be adopted quickly into their information systems. In general, information systems are the operational manifestation of the business knowledge of an enterprise. Behavioural descriptions, business rules paired with structural schemas are a significant part of that knowledge [1] and are mostly specified using natural language descriptions supported by images. Due to market demands, a proper degree of flexibility in adopting changes of the business knowledge is obligatory in a business environment where rules are likely to change frequently and rapidly [2]. For a couple of decades enterprises face a trend towards the migration of their business knowledge to information systems which also means that the business knowledge originators, domain experts, are getting more and more dependent on IT

* Funded by the German Ministry of Economy and Technology due to a decision of the German Federal Parliament. In German: "Gefoerdert vom Bundesministerium fuer Wirtschaft und Technologie aufgrund eines Beschlusses des Deutschen Bundestages". ZIM-Project with Number KF2122301SS8

experts like software architects and developers. A counterforce to the common trend towards the unintended migration of the business knowledge into IT departments is to enable domain experts to specify requirements. This shall be achieved by enabling domain experts to specify requirements on their own using a language that is not too technical [2] by at the same time being sufficiently formal to grant machine based aid. Otherwise the latent danger of removing domain experts from their central position as domain knowledge generators and holders is imminently given. The gap between the requirements specification documents and the implementation of those requirements widens the more complex an enterprise application becomes. The constant evolution of existing systems bears the danger of an inadequate design, which tends to be normal under the common pressure of software development projects. This becomes even more critical in large and complex enterprise applications. If the complexity becomes unmanageable, the system has to be cost-intensively rebuilt from scratch. To tackle the roots of this problem was the goal of researchers and tool vendors for the last decade. However, the major contribution towards modern development paradigms like Model-Driven Software Development (MDSD) [6] we see in mechanisms to also allow formal specification of business rules next to structural schemas usually given by class diagrams [5]. In MDSD, we see the most promising candidate for overcoming the complexity of large enterprise applications at design time and later evolution. The reason is that on the one hand by using graphical modelling domain experts preserve their ability to specify significant parts of the software systems theirselves and thus remain in the role of domain knowledge generators and holders. On the other hand if being sufficiently formal in building up abstract system specifications, the degree of code generation can be raised significantly [40]. Thus, employing mechanisms to support domain experts utilizing MDSD in a structured and clear manner is the focus of this collaborative work. However, the key factors for successfully allowing domain experts to apply MDSD we see in providing modelling methods that are easy to learn and use [3,4]. These modelling methods have to be expressive enough to define an application's structure as well as its behavioural aspects. Additionally, means to define guides for the user's interaction with the application is another success factor. We achieve this with the aid of selected graphical UML model types as well as a proprietary builder for making up graphical user interfaces (GUI). A key enabler for the mentioned aspects is a proper tool support, which we bring up with the AtomsPro-Tool (see Section 4). AtomsPro is the project frame of this work and aims at developing a new method for the cost efficient development of maintainable distributed applications. However, it is necessary to enhance graphical models, especially UML class models, with additional semantics to overcome the needed specification degree for automated code generation. Hence, UML class models are used to specify structural schemas of an application which define parts of the business concepts and structure. Business concepts can be seen as the basic vocabulary for defining business rules and if being sufficiently formal the cornerstone for automatic code generation is laid. The Object Modelling Group (OMG) tries to answer this by introducing a textual general-

purpose language for business rules, the Object Constraint Language (OCL [4]). OCL is included in the UML 2.0 specification and provides a formal basis for specifying business rules upon UML class models. Work done in [41] shows that OCL is able to cover a variety of problem domains concerned with business rules and structural schemas. However, Halpin [7] et. al. argue that OCL is too technical to be used by most domain experts who are used to work with natural language specifications. From a practical viewpoint the need for a language that caters to domain experts and not only to programmers is imminently given. Thus, we propose the AtomsPro Rule Integration Language (APRIL, see Section 5) as an integral part of AtomsPro, which is a new language for formally specifying business constraints in a natural language like syntax that may be suitable for domain experts. It is enhanced by language constructs that ease the use of complex common business constraints elaborated and introduced in [1, 7, 9]. Statements made in APRIL can be compiled to OCL.

Another key factor for successfully maintaining and evolving large enterprise applications is to provide means for consistently supporting aspects of their architecture and deployment as both are depicting the glue between the application's software code and the actual system they are running on. In terms of defining an application's architecture substantial work has been done e.g. [8, 10, 11, 12] to formally specify different kinds of architectural issues of applications that are known as architecture definition languages (ADLs). Next to textual ADL approaches there exist graphical UML based ADLs that are more favourable in practice as they base on well founded and well structured modelling methods as well as a wide supporting tool landscape. With AtomsPro we support graphical architecture and deployment modelling. Especially deployment modelling is pretty much neglected in the currently existing state of the art ADLs e.g. like [8]. AtomsPro consequentially integrates graphical domain-, GUI-, architecture-, and deployment modelling as well as textual business rule specification. Hence, we enable a high ratio of automated code generation making the work of software developers more efficient. In order to achieve our goals the workload is tackled collaboratively by the project partners eMundo and the University of Applied Sciences Ingolstadt.

This paper is structured as follows. In Section 2 a brief overview of research environment of AtomsPro is given. Related work is introduced in Section 3. APRIL, presented in Section 5, is a subproject of the AtomsPro project, which is presented in Section 4. Finally, Section 6 presents some conclusions and future work. In the Appendix the concepts of the APRIL language are described. Hence, the Appendix is structured as follows: Part A contains the APRIL-syntax in EBNF. The Sections B to E handle semantic issues in a semi formal way. They depict the first steps towards an exhaustive and formal definition of APRIL.

2 Research Environment

In our project we have chosen a fruitful cooperation with several involved project partners. The main target has been to achieve a real product or at least an in-

novative prototype, which is strongly based on research results. So this can be summarized as research in applied sciences. We have involved different partners we need for reaching that goal. The whole project has been founded by the German government (BMfWi, Bundesministerium fuer Wirtschaft und Technologie) in the research program ZIM (Zentrales Innovationsprogramm Mittelstand) under the project number KF2122301SS8. This program connects smaller and midsize companies with research institutes to develop innovative products with a high business value. To achieve that goal the partners are tied together in an integrated project plan.

2.1 eMundo

eMundo is the industry partner of the project. eMundo develops distributed business software systems for their clients where standard software, e.g. standard ERP systems, cannot be applied. The applications eMundo delivers to their clients range from Web-based information systems to resource planning and scheduling systems, typically for a large number of users (10-10000). Nowadays software needs to fit into the client's IT landscape and -infrastructure. To achieve this, for every project the software development process as well as the software architecture are individually tailored to meet the client's regulations. Not only since companies are relying on outsourcing to develop software the software development market is getting more and more competitive. Thus, a major goal for companies like eMundo is to reduce software development costs while at the same time preserving software quality. From the client's perspective maintainable code is a very important requirement since most of the total expenditure for an application are spent for maintenance, i.e. bug fixing and the implementation of new features.

2.2 University of Applied Sciences Ingolstadt

The University of Applied Sciences of Ingolstadt has been founded in 1994 as a teaching and research institution to support regional companies. With a strongly increasing staff, today 3000 students are taught. In the research institute there are currently 40 researchers, which are all founded by companies or by individual government founds. Mainly the language for defining business rules (APRIL) will be driven by the University of Ingolstadt.

2.3 De Montfort University

To cover also research topics which might be in the area of basic research, the University of Ingolstadt has an established research cooperation with the De Montfort University in Leicester, UK. In our area, we are cooperating with the Software Technology Research Laboratory (STRL) under the head of Hussein Zedan. The STRL is a pure research institute within the De Montfort University.

3 Related Work

Work in the context of model driven software development (MDSD) is focussing towards the integration of domain experts into the softwar-development, software-maintenance and software-evolution process [6]. This is done by means that allow expressing business logic in a way domain experts are used to and by at the same time being sufficiently formal to grant automated code generation. Strong emphasis is laid upon specification methods that allow to express business logic as abstract as possible by at the same time keeping the needed detail level for a software implementation low. Another highly recommended issue is to reduce the overall amount of detail that a member of a development team has to be concerned with. Separation of concerns [20] (SoC) helps to ensure to divide concerns of different stakeholders e.g. a software architect manages the morphology of the software application itself whereas a domain expert deals with business rules. Recomposing the different views is no longer a task for human developers only but more and more for state of the art tools that comply to the MDSD paradigm. Established proprietary software development tools allow generating parts of an application. Formal models, mostly specified in UML, are required for an automated code generation. However, tools supporting MDSD can be divided into two groups. The first group is supporting the model driven architecture (MDA [29]) paradigm, which is a methodological implementation of MDSD and elaborated by the OMG. From the scope of AtomsPro the MDA paradigm requires too many steps of indirection to come to the result which shall be executable platform specific software. The second group does not obligate such intermediate model transformation steps. Members of that group claim to support all degrees of freedom of the MDSD paradigm. Hence, they are the main competitors to AtomsPro and can be subdivided furthermore regarding the kind of notation they support. On the one hand, there are tools that support graphical notation e.g. [28] and on the other hand, there are tools that support pure textual notation e.g. [26, 27]. The last group mentioned afore allows to define proprietary languages tailor-made for a domain's needs and also significantly supports the construction of a proper tool support for that language. That is why they are also called language workbenches [19]. Models, which are actually textual code in the user-defined domain specific language, can then be translated into a platform specific form e.g Java by a user defined compiler whose preparation is then again aided by the language workbench. AtomsPro does not support specifying user defined textual languages this way but makes use of a language workbench to define a business rule language that is like natural language and compliable to OCL. However, our focus is on graphical modelling and so the highly relevant related work is done amongst tools like Eclipse Open Architecture Ware [31], ObjectIF [30] and AndroMDA [32]. MDSD-tools basically allow to define additional semantics to an existing or to be defined meta model which is the basis for the desired domain specific modelling concepts. These semantics mostly appear as code templates of the target language and serve as input for a generator module that also consumes a certain domain model in order to cast its logic into a target artefact, which is often some kind of software application

code. However, the feature landscape amongst MDSD-tools that provides the needed functionality in a user friendly way can be regarded as heterogeneous. So for example some common IDE mechanisms like automatic code completion, type checking or even debugging are often not considered to a satisfying extend. The most significant shortcomings are in the area of considering architectural structures for the generated application. If there is a need to apply individual architectural schemas to a generated application, tremendous changes in the generator have to be conducted whether the tool is open enough for allowing changes at all. A focus of AtomsPro is to cater for architectural structuring of the generated application. Hence, another significant aspect is to provide means to specify the architecture of an enterprise application. Significant work has taken place with [8, 42] to specify certain architecture description languages (ADLs) to overcome the complexity of formally specifying highly modularized application components that have to be glued together. Typically, these approaches are to be divided into two groups. One group pursues the way to textually specify certain architectural issues and the other group does so by providing graphical forms of notation. Specifications made in any of the ADLs commonly serve as additional input for generating platform specific code for an enterprise application in terms of the logical structuring. Another dimension for distinguishing ADLs is the purpose for which they are designed. Thus, most ADLs additionally are to be divided into either general-purpose ADLs (e.g. UML profiles for architecture specifications [43] or ACME [8]) or domain specific ADLs (e.g. for embedded software for avionics). Nevertheless, almost all of the ADLs operate on a basic subset of modelling components, which are components, connectors, configuration blocks and constraints. The instances of those elements may than be composed utilizing the concrete syntax of the ADL for a certain architecture description. In an example scenario components may represent client and server applications, connectors can be seen as procedure calls, event broadcasts or database connections. Configuration blocks contain metadata on components and connectors. The UML can also be counted to the general purpose ADLs as it provides extension points, UML profiles [44], to tailor notation concepts for defining architecture models individually for a project's needs. Although it is often argued that the UML is too overloaded for describing architectures as it does not explicitly cater for the typical component-interface-connector focus it indeed enjoys a broad field of usage for architectural purposes. The reason for that may be sought in a widespread and mature tool support and the fact that many developers and architects are familiar with the UML. This motivated [18] to develop a transformation scheme from the textual ACME to a UML-based notation. Some research aspects extend the scope of ADLs. They are on the one hand dealing with tool based analysis and optimization of architecture specifications that concern non-functional requirements [10, 17] and on the other hand focus on supporting the architectural evolution as a part of a tool based software development process [11, 13, 14, 15, 16]. With AtomsPro we do not yet cover the issues of both groups. Despite the fact that the methodological and tool based aid around ADLs seems mature they do not claim a leading position in the

industrial practice. One reason might be that issues for describing deployment aspects have not found their way into the languages' concepts. AtomsPro closes this gap.

With APRIL (see Section 5) we translate formal business rule statements that come close to natural language into OCL. Approaches that deal with the formal specification of business rules by means of natural language focus on utilizing pure natural language e.g. [45, 46]. However, all the approaches investigated are either very extensive and by far non-trivial e.g. OMG's Semantics of Business Vocabulary and Business Rules [45] (SBVR) or have to be implemented using complicated probabilistic methods that are only able to estimate a statements formal semantics e.g. [46]. However, they are neither as easy to learn and use for business people nor as accurate as it is necessary to aid code generation. However, they roughly are to be divided into three major fields. One group providing support for easier ways to formulate OCL expressions is done in [47,48,52]. A second group focuses on paraphrasing OCL- expressions with natural language [47,51]. Thirdly [46] is a representative of a group that maps natural language to database languages (SQL[52]) that cover similar aspects in the relational world like OCL does in the object oriented world. In [47] a method is introduced to annotate rules graphically. OCL-constraints are generated with the help of predefined constraint design patterns which provide the semantics at the same time. [48] picks up this idea and tries to systematically simplify OCL constraints that have been generated based on the patterns. Their intention is to make constraint statements shorter by preserving the semantics. Doing so OCL statements shall become more readable. [52] discusses methods to specify design patterns. This group of approaches bears two major problems. First there has to be a huge variety of design patterns for all different kinds of constraints arising in practice of enterprise business that are far away from being trivial. Second, it lacks support for defining new complex constrains which is the main focus of our approach. The group that focuses on paraphrasing OCL constraints directly with natural language on the one hand introduced by [51] and on the other hand with [53] the standardized SBVR is utilized as an intermediate translation step. The reason why [51] comes into consideration for discussion is because the generated English is very good. Although from our point of view this approach does not primarily focus on domain experts but on programmers who are given an immediate semantic feedback upon the specified OCL statements. Why generated specifications (proposed in [51]) are critical has the same reason as [53] which approaches intents to give the domain expert immediate response on the status quo of the implementation. A flaw we see on this is, that it does not pull the domain expert from his peripheral position into the implementation process as it is propagated with methods where DSLs [54] contribute to MDSD. Another is a latent danger that communication between domain experts and programmers becomes unidirectional. Programmers could refer to the generated specification if they are asked to give a status report. What might be convenient for the programmers but unreasonable for domain experts who would have to read through hundreds of pages of generated text. Making all aspects of natural language suffi-

ciently formal for programming purposes is a trade-off we consider too expensive. A third group from the field of artificial intelligence pursues approaches towards machine based learning in order to provide a basis for transforming natural language into formal languages [55,56,57,58]. Another highly interesting approach in that group comes up with [46]. Giordiano and Moschitti describe a method from the field of artificial intelligence to combine natural language processing [59] and probabilistic methods based on [55,56,57,58] to transform natural language questions to SQL queries. They use different special heuristics with training examples to automatically map syntactical structures from template like natural language questions to those of SQL queries. They are measuring the effectiveness of each mapping algorithm by using them in Support Vector Machines [60] in order to select the correctly mapped structures of pairs of natural language questions and SQL queries. This shows that the role of mapping syntactical tree fragments between a natural and a synthetic language is an important step in gathering the relational semantics of two languages. The approach of Giordani and Moschitti shows that from our point of view tremendous effort has to be taken to translate natural language into an artificial language. For automatically gathering semantics they generate round about 72 wrong structural tree pairs to get one right out of it which is then useful for translating only one type of natural language question. This impressively shows that such translation tasks using syntactic structures combined with statistical methods can only be tackled with a considerable trade-off. Translation problems in this area are at least known since [61]. Another approach [47] for supporting domain experts in textual requirements engineering is to provide mechanisms for consistency checking with the help of natural language processing. Koerner et. al. propose a tool for checking textual requirements documents. The checks are based on onthologies providing the semantic fundament for reasoning about linguistic defects of the natural language requirements documents. The authors argue that the tool is able to point out to ambiguous and imprecise formulations and give propositions for improvements based on information gathered from external models (onthologies). Despite the fact that this approach is not able to generate formal business rules at the moment, it is peripherally competing with APRIL as it may provide domain experts with means to precisely (but still not yet formal) express business rules in natural language.

4 AtomsPro

In this section we want to give a brief overview of the concepts of AtomsPro.We also want give a glimpse of its effective usage.

4.1 Definition of AtomsPro

The project aims at developing a new method for the cost efficient development of maintainable distributed applications. The key concept of the approach is to save development time through the use of predefined software components as

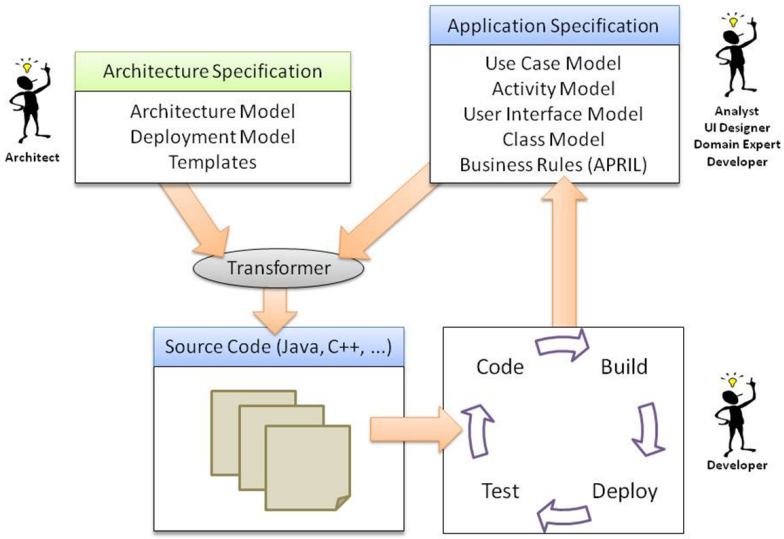

Fig. 1. AtomsPro concept

well as maximizing code generation. The starting point for the code generation is a UML[5]-based high level model of the application consisting of various views (see Figure 1). The Use Case Model shows how an application's functionality can be described in terms of use cases [38] and actors. For each use case defined in the Use Case Model an Activity Model - consisting of UML activity diagrams - can be used to specify its flow of control as well as the objects associated with it. Objects are created from classes which may be defined in the Class Model. It materializes through a number of UML class diagrams where classes may be defined for persistent and transient objects, respectively. As many applications provide some kind of user interface a User Interface (UI) Model can be used to specify the dialogs that may be used to interact with users as use cases are being performed. In order to maximize the expressiveness of UI models dialogs and dialog controls can be associated with (data) objects based on the types defined in the class model through query expressions specified in A4L. A4L (AtomsPro 4th generation language) is an implicitly typed object-based language that has been devised to support the implementation of business logic with minimal effort. This is achieved through a number of language features such as integrated application model access and navigation, implicit elicitation of data types, support for querying application data at runtime and the integration of a language to define business constraints (see Section 5). Furthermore, an activity model must be specified for each dialog to indicate how the dialog's events (e.g. button clicks) should be handled. Consequently, these event-handling models enhance a

use case's overall activity model, as dialogs are included in use case executions - typically through predefined dialog open activities. Once the specification has reached a certain maturity, it is the transformer's task to generate source code from the application specification. Before this can be done the developer needs to choose a software architecture specification to base the generation process on. The software architecture specification uses a software architecture model to specify which architecture artefacts (e.g. a Data Transfer Object) are used as building blocks for the software and how each artefact can be created from the application model using a particular template. An AtomsPro template consists of two views - an output view which is responsible for creating the output lines of the resulting file, and a logic view which uses A4L code to prepare the variables which are being referred to in the output view. Each template is parameterized through an A4L query which is formulated over the application specification to provide the specification elements (e.g. all persistent classes with 'persistent' = true) code should be generated for. Finally, a number of deployment models specify how the application should be deployed in the various development stages and environments, e.g. development, system test, production. During source code generation the existing manually written code is preserved. Once the transformer has completed, the typical code-build-deploy-test cycle can be performed by each developer resulting in changes to the manual code as well as the specification model. In the latter case an incremental generation step may be requested by the developer to reflect model changes on source code level.

4.2 Towards a methodology for the efficient use of AtomsPro

AtomsPro not only aims at providing techniques and tool support for the model-based generation of source code for a freely definable application architecture. It also supports the standard phase cycle of common software development processes sketched in Figure 1.

- During analysis phase AtomsPro supports a rapid development approach by providing an architecture model called "simulation". If chosen, this model generates code to enable the local standalone execution of the application which greatly helps to elicit requirements together with other stakeholders. Use case modelling is used to describe the application's functionality. Each use case will be further detailed by an associated activity model, which can be further decomposed to handle complex use case flows. The data needed for the use cases to be performed can be specified by defining a class model. Then, a user interface may be designed using a UI builder to specify the user interface model of the application. UI event handling and data binding can both be specified with little effort using A4L code. Based on the aforementioned models the AtomsPro IDE is capable of a simulation run showing the application in its prototype state. More functionality may be added, e.g. by implementing class model operations or queries in A4L.
- Once the application specification has reached a certain maturity, a different architecture model may be chosen to start with the actual development

(phases design and implementation). The architecture model chosen will import new model properties (e.g. persistent for classes) which have to be set accordingly, as they are required by the code generation and the templates it relies on. The A4L code developed so far may be kept (A4L is by default translated into native Java code) or be translated into the target language (currently only Java is supported). From now on, the model-transform-code-build-deploy-test cycle

5 APRIL

Authors in [1,7] argue that there is a need to substitute the pure natural language business rule notation by a formal language that is non-ambiguous and able to contribute to methods of MDSD. The other side of the medal is that this benefit is bought at the price of clarity suspending non-technical domain experts from the software creation process [23]. Major points where criticism towards formal languages like OCL is put on is firstly that its statements quickly become cryptic. This effect gets even emphasized while attempting to express inherently complex and/or extensive business rules. And secondly, handcrafting business rules in a formal textual language (OCL) is time consuming and error prone as the user has to express the underlying semantics by using the exact syntax. Relegating to the work done in [33] that last point can be fought against arguing that if a powerful integrated development environment (IDE) is provided complexity effects of the language itself do have a weaker impact. Hence, our work is mainly motivated to provide a clear formal language supported by a proper IDE enriched with state of the art features allowing to specify business rules that are understandable, usable and maintainable by non-technical people. An additional challenge may be to elaborate and combine natural language support with state of the art IDE features. Thus, we propose APRIL (AtomsPro Rule Integration Language), a formal declarative language for business rules. APRIL's syntax is based on natural language concepts that derive from predicate logic and operations on sets as well as common business constraints [25]. The aim is to enhance UML class models with additional semantics that can be understood by domain experts with weak technical background. The semantics of APRIL is underpinned by OCL. APRIL-statements are expressed in structured natural language and can be compiled into OCL expressions. Then they can be processed further by existing engines e.g. [34, 35, 36]. We seek to achieve a gain in understandability in the process of formal requirements specification with concepts that are known from other formal languages combined with new concepts based on natural language. First is the concept of decomposition of business rules into simpler entities. Means for that are package building, easy and clear definition of sub expressions (called Definitions in APRIL) and intra-rule variables. Secondly, by allowing the use of mixfix notation at certain points, possibilities to form statements that are pretty close to natural language are offered. Thirdly, we use a type system that is similar to that of OCL which supports APRIL's static typing making it easier to support semantic checks within the IDE. In the

following a brief overview of the syntactical concepts of APRIL and its project environment is given.

5.1 Basic Concepts of APRIL

Every APRIL expression is based on a UML class model. As mentioned earlier APRIL is based on predicate logic and operations on sets of objects which are instances of classes. Selecting sets is done similarly to OCL by stating a navigation along related classes in the associated UML model. For the example presented with respect to Figure 2 we assume that the current context of the rule is an object of type `Customer`, then:

```
cards.transactions
```

will collect all transaction objects that can be reached from the customer object given by navigating the associations `Customer` to `cards` and `CustomerCard` to `transactions`, respectively as a set. Using the dot-notation a flattened set of objects with type Transaction is returned and not a set of sets. APRIL also allows to abbreviate any navigation between classes by only stating the target role name. A navigation can only be abbreviated if the entire path is non-ambiguous and there are no class attributes involved. Path targets are specified with the role name of the association connected to the target class. The amount of matching path descriptions can be reduced drastically if roles get named with a model-wide unique identifier. For defining sub sets, any set within the navigation definition can be reduced by a logical expression. The next example shows a selection yielding a set of objects of type Transaction in `cards.transactions` where the value of the attribute `amount` is greater than 100. It reduces the set of Transaction-objects by assuming `Customer` as context. Note the for the introductory examples the long navigation versions are used:

```
each transaction in cards.transactions where
amount > 100
```

A valid business scenario for this explanatory rule may be to find out which transactions are profitable. Going further in the demonstration of predicate logic expressions in APRIL the universal- and existential-quantification returning a truth value in contrast to the upper examples look like the following. The business rule: "All transactions are required to have an amount greater than 100" can be translated to:

```
every transaction in cards.transactions
satisfies that
amount > 100
```

The universal-quantification yields true if for all objects of type Transaction in `cards.transactions`: the value of the attribute `amount` is greater than 100 holds. In contrast the existential quantification given below yields true if there is at least one object of type Transaction in `cards.transactions` satisfying that the value of the attribute `amount` is greater than 100.

```
at least one transaction in cards.transactions
satisfies that amount > 100
```

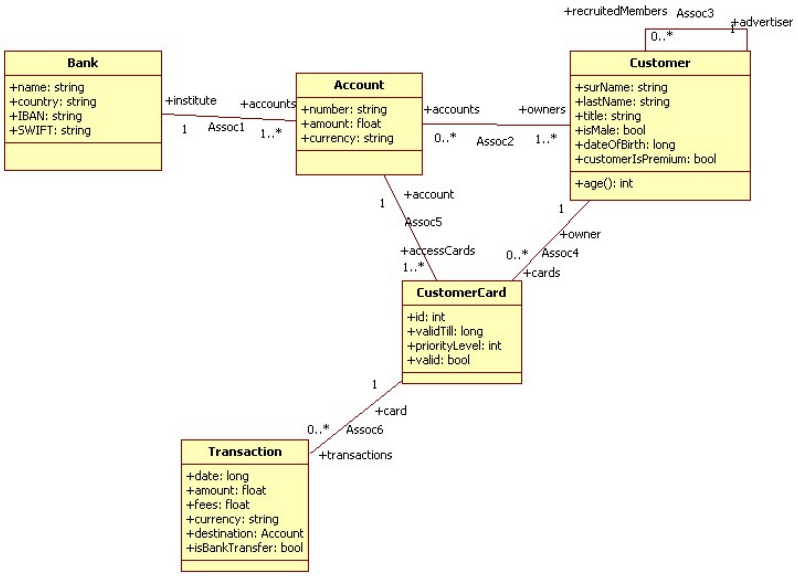

Fig. 2. Bank Customer Example

APRIL also provides means to optionally make the syntax of functions with iterators more convenient. E.g. for reducing `every transaction in transactions satisfies that <exp>` to the simpler form of `every transaction satisfies that <exp>` APRIL uses the following rules:

1. Given a role-naming convention, which says that association ends shall only be named with nouns and association ends with cardinality "many" shall be named in plural form, the syntax based inference rule is as follows:

   A function e.g. `every transaction satisfies that <exp(transaction)>` shall be expanded to `every transaction in transactions satisfies that <exp(transaction)>`.

   - Convert the name of the iterator variable into the plural form usually done by the plural-s : *plural(transaction) = transactions*. The plural-s building rule is a standard convention in English. Exceptions of that rule can be handled with a singular-to-plural mapping table.
   - Try to resolve the superset with `plural(transaction)` by utilizing the navigation abbreviation rule mentioned earlier.

2. If this algorithm is used in a Definition the parameter names are used for resolution before the role names. Generally a Definition inherits its context from the Rule it is used in.

5.2 Rule in APRIL

A rule in APRIL is the frame for the specification of business logic and is roughly divided into three sections. First is the header section where name and context are defined. The header section is followed by the actual business rule statement that always yields a truth value. The third section separated by the keyword `with` is for Local Variable Definitions (see Section 5.5). So for example the business rule: "In Germany wealthy senior customers don't have to pay transaction fees for their outgoing transactions to accounts of foreign banks." can be translated into: (see also Figure 2)

```
Invariant NoFeesForGermanSeniors concerns Bank:
every customer in WealthyMaleSeniorGermans satisfies that
the customer donesnt have to pay fees for
   transactions to accounts of foreign banks
with
WealthyMaleSeniorGermans is defined as
wealthy male senior owners in Germany .
```

Starting from the explicitly stated context Bank (indicated by the concerns keyword) the universal quantification function `every <element> in <set> satisfies that <statement>` is used to check if the embodied statement holds for every single customer element. It is mandatory that statements made in the body of the Rule itself yield a truth-value. Note that the parameter `transactions` depicts the optional short form of `customer.cards.transactions`. However, it may be noticed that the statement defined is untypical for a programming language as it is read pretty much like natural language. The key enabler here is the use of mixfix operation naming (see also Section 5.4) which allows composing name parts and parameters in an operator descriptor arbitrarily. Moreover, the definition calls that are defined in the next code section are nested. The concept of nesting mixfix named operations is explained for the following local variable definition as it is considered more obvious. In the local variable definition block (after `with`) a nested definition call utilizing a mixfix named operator is used to apply a filter on the Set of Customer objects addressed by the `owners` navigation-statement which is the abbreviated form of the navigation `accounts.owners`. For supporting the explanatory intent for the nested operator calls used, the thought and more common prefix-notation (like in Java) may state the body of the local variable definition named `WealthyMaleSeniorGermans` like the following: wealthyMaleSenior(inGermany(accounts.owners)). In order to be able to make up all the operator statements used in the Rule above their semantics have to be defined properly. This is done in the following supporting examples on APRIL Definitions tailor-made for the Rule they are used in. A description on the motivation and usage of APRIL Definitions is done in Section 5.3

```
Filter (customers as Collection of type Customer) in Germany
yielding Collection of type Customer
is defined as
each customer where
  at least one account satisfies that
     institute.country = 'Germany' .

Filter wealthy male senior (customers as Collection of type Customer)
yielding Collection of type Customer is defined as
```

```
each customer where
age >=55 and isMale and sum of accounts.amount >= 1000000 .

Value the (customer as type Customer) donesnt have to pay
       fees for (transactions as Collection of type Transaction)
  yielding Boolean
is defined as
every transaction in CustomersTransactions satisfies that
  fees = 0
with
CustomersTransactions is defined as
 each transaction where
  owner = customer .

Filter (transactions as Collection of type Transaction) to accounts
    of foreign banks yielding Collection of type Transaction
is defined as
each transaction where
  destination.institute.country <> 'Germany' .
```

The identifiers of the Value and Filter Definitions are defined in mixfix notation which is declared as a space separated list of name parts surrounding place holders for parameters that are explicitly typed. In the examples above the first Filter definition is a generic collection type specified by the user defined model type Customer. The entire example (Rule plus its Definitions) impressively shows how APRIL encourages decomposition that in the end leads to a better readability of an extensive core rule while in fact the total count of the lines of code may increase. This in fact can be examined in the following semantically equal OCL statement.

```
context Bank inv NoFeesForGermanSeniors:
let wealthy_male_senior_customers : Collection(Customer) =
accounts.owners->select(customer | customer.age >= 55 and
customer.isMale and customer.accounts.amount->sum() >=
1000000) in
let WealthyMaleSeniorGermans : Collection(Customer) =
wealthy_male_senior_customers->select(customer |
customer.accounts->exists(account |
account.institute.country='Germany') )
in
WealthyMaleSeniorGermans->forAll( customer |
customer.cards.transactions->select(transaction |
transaction.destination.institute.country <> 'Germany')->
 select(transaction |
transaction.card.owner = customer)->forAll(transaction |
transaction.fees=0))
```

We assert that in contrast to the APRIL example the OCL example is only understandable to a programmer familiar with OCL. In the following sub sections the basic concepts of the upper example and the related concepts of APRIL are briefly discussed.

5.3 Definitions

In APRIL a definition is a named expression which may be used to outsource expressions from the rules. One particular feature that can be used for Definitions are names in mixfix notation (See Section 5.4). Definitions can be accessed by Rules and other Definitions by stating the name and parameters that match in number, order and type, respectively. The body statements of Definitions are

similar to those of Rule statements. We distinguish three types of definitions as mentioned in the following sections.

Filter and Value Definitions An often faced scenario specifying business rules is to filter subsets by applying set comprehension on values of elements in the base set. Thus, Filter definitions can only return sets. This type of Definition is introduced with the keyword `Filter`. As filters are very common in entity modelling we have introduced an extra construct. A more versatile kind of definition is the Value Definition. Its return value can be of any type and can be gathered implicitly from its expression at compile time if not explicitly stated. Its notation is similar to a Filter Definition introduced by the keyword `Value`.

Model Extension Definitions In APRIL it is possible to annotate extensions for elements in UML class models corresponding to the abilities given by OCL that enables the user to textually define attributes and operations. In APRIL, this ability is restricted to class attributes (introduced with `Class attribute`) and side effect free methods (introduced with `Class operation`). This kind of Definition is defined similarly to that of the Filter Definition by substituting the introductory `Filter` key word with one of the afore mentioned. The motivation behind this is the integration of object oriented features in APRIL.

5.4 Mixfix Operators

In APRIL a definition header may consist of a user defined mixture of name parts and parameter definitions which is similar to what is called mixfix operator in some functional languages. In the older literature mixfix may also be referred to as distfix [50]. Danielsson et.al. [49] show how to parse mix fix operators. The benefit of mixfix naming is the gain in flexibility when it comes to achieving the best readability possible for a business rule. Any definition can be seen as an extension of the business vocabulary [21]. The syntactic structuring [22] by means of natural language is delegated to the user, as we think that the human ability of applying natural language is superior to that of a machine. Thus, we do not utilize natural language processing. Mixfix naming supports the notation of syntactic structures closely to natural language [23] which then can be used to combine business entities. This kind of naming allows APRIL to stay light weight in terms of its syntax.

5.5 Local Variable Definition

A local variable definition is a method for decomposing business rule expressions. It is defined inside an APRIL business rule and encapsulates an expression that has a type and an identifier. These definitions are to be used in other local variable definitions or in the statements of the actual Rule or Definition. The typing is done implicitly by deducing the type of the expression it encapsulates.

5.6 Common Constraints

A central aspect that tremendously contributes to the expressiveness of APRIL is the inclusion of language constructs that allow to shortly specify constraints that occur very frequently in practical modelling tasks. They are also often referred to as common constraints. In this paper, we also want to stick to this nomenclature. In [7, 9, 23, 24, 25] research has been done to elaborate, identify and group the different types of common constraints. Costal et.al. [1] show that these types of business rules can cover round about 50% of the over all constraints occurring in a real life scenario. However, this paper exclusively deals with the identifier-common-constraints incorporated into the taxonomy depicted in Figure 3 showing a hierarchy of the most important common constraints.

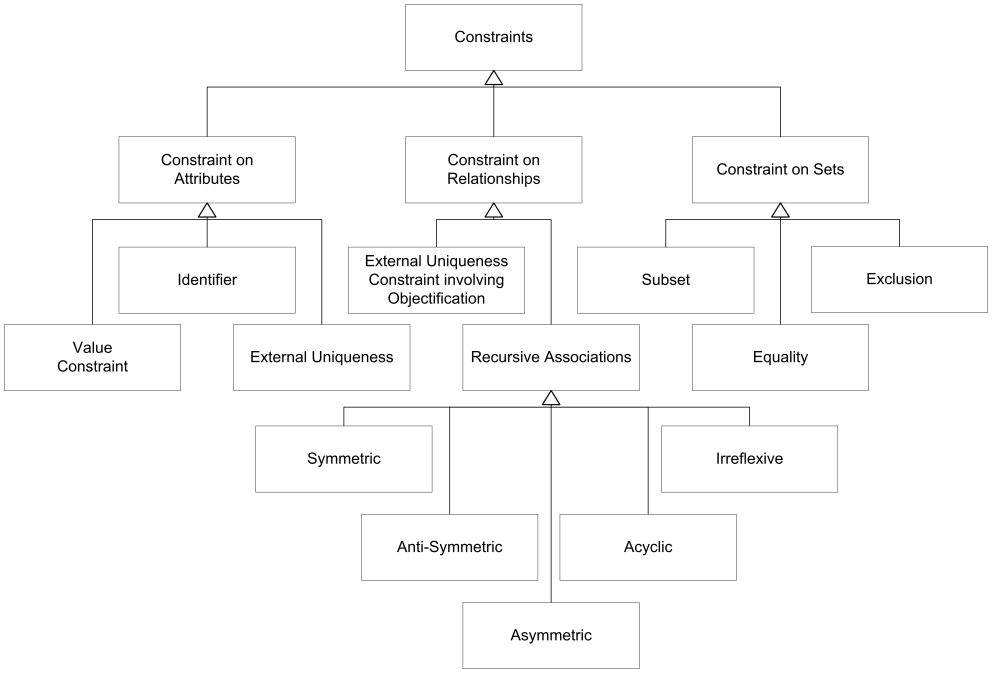

Fig. 3. Taxonomy of Statc Common Constraints

The identifier is a constraint that is known to many modelling methods that deal with entities. UML's class attributes are predestined for holding a primary identification rule stated in OCL or APRIL as UML class diagrams by default lack such means. A real world scenario with respect to Figure 2 may be that a Bank has a unique IBAN-number, which is stated in APRIL as well as OCL as follows :

```
Invariant BankIdentification concerns Bank:
IBAN is unique .
```

```
context Bank inv BankIdentification:
Bank.allInstances->isUnique(IBAN)
```

A more general version of the primary identifier constraint is the internal uniqueness constraint or composed identifier. It says that value combinations of two or more attributes of a class are unique [23, 25]. E.g. in Figure 2 an object of type Customer is identified by `name` and `dateOfBirth`. The APRIL and OCL statements look like:

```
Invariant CustomerIdentification concerns Customer:
each name, dateOfBirth combination is unique .
```

```
context Customer inv CustomerIdentification:
Customer.allInstances->forAll(c1,c2 | c1<>c2 implies
not((c1.name = c2.name and
c1.dateOfBirth = c2.dateOfBirth)))
```

5.7 Syntax of APRIL

The syntax of APRIL is specified as Grammar in Extended Backus-Naur Form (EBNF, see Appendix A). This specification is the basis for building the APRIL-parser, which integrates into the APRIL-tool currently under construction. We have realized the parser using the eclipse based xtext-tool [26]. In the following we give a short description of the APRIL syntax using two example rules.

For addressing all instances instantiated from a class the following expression is used:

```
"all␣instances␣of" <ID>
```

The APRIL all instances function is introduced by the "all instances of" keyword followed by an `<className>` that is a class name (e.g. `Customer`).

In APRIL, one of the core concepts is the possibility to decompose large rules into smaller fragments ((see 5.3)). For that reason we want to describe the syntax of the Filter Definition. For a more detailed insight into the syntax of APRIL the reader may be referred to Appendix A.

```
<FilterDefinition> ::=
"Filter" <MixFixName> ["yielding" <ID>] "is␣defined␣as" <Rule>
```

A concrete example following this rule is the expression (adopted from Section 5.2):

```
Filter (transactions as Collection of type Transaction) to accounts
    of foreign banks yielding Collection of type Transaction
is defined as
each transaction where
  destination.institute.country <> 'Germany' .
```

The non-terminal symbol `<FilterDefinition>` is the identifier for the filter rule. The filter rule is introduced by the keyword `Filter` followed by a user defined mixfix name. The mixfix name may consist of several white-space-separated name constants and parameter descriptors in an arbitrary order. The keyword `yielding` indicates the type specified by a descriptor e.g. `Collection of type Customer`. Although this expression is marked optional, indicated by

square brackets, we strongly recommend to make use of it. As if not, the derived type from the body expression might differ from that which the user intended. The header of the filter definition is separated from its body by the keyword `is defined as`. The body of a filter definition in APRIL is indicated by the non-terminal symbol `<Rule>`.

The complete syntax of APRIL is in Appendix A.

5.8 First Steps Towards a Semantics of APRIL

The semantics of APRIL is defined using the Object Constraint Language (OCL) 2.0 [4]. We chose a denotational semantics for mapping APRIL expressions into OCL-expressions. In order to indicate the meaning of an expression we use $[\![.]\!]$ as interpretation function as follows:
$[\![.]\!]$:`APRIL_Expression` $\rightarrow$ `OCL_Expression`.
Whereas `APRIL_Expression` denotes a set of character sequences that have to be built obeying the `<Rules>` production rule according to the syntax specification (see Appendix A.1). Any character sequence of sort `APRIL_Expression` can be translated to a character sequence of sort `OCL_Expression`, which obeys the production rules of the OCL syntax [4].

The semantic foundation of OCL is realized by several tool manufacturers according to the OCL specification. Moreover, a lot of effort of the scientific community has been taken to provide semantic underpinnings for OCL e.g. [4, 35, 47]. The presented semantics is therefore based on these results. Even if that work might not be finished or discussed, due to the existing implementations of OCL-tools, OCL can be used as foundation of the semantics. Additionally, the use of a slightly discussed OCL semantics is better than using natural language for describing business rules.

Most of the basic functions of APRIL (e.g. select-function see Appendix C) can be used directly to OCL and do not need any further processing with respect to the APRIL definitions (see 5.3).

Here the translation of the APRIL AllInstances-function stands as an example for the translation of basic functions from APRIL to OCL.

$$[\![\text{"}all\ instances\ of\text{"} < className >]\!] := [\![< className >]\!] -> allInstances()$$

In this example, the semantics of the APRIL AllInstances-function is mapped onto its corresponding OCL-function. The concrete usage in a stepwise translation of an APRIL-select function may look like:

1. $[\![$`each transaction in all instances of Transaction where destination.institute.country <> 'Germany'` $]\!]$
   In this step, we choose the translatation into
   `<SelectFunction> ::= <source>->select(<iterator>|<body>)` (see Appendix C, first table, row nine).

2. $[\![$`all instances of Transaction`$]\!]$`->select(`$[\![$`transaction`$]\!]$`|`
   $[\![$`destination.institute.country`$]\!][\![$`<>`$]\!][\![$`'Germany'`$]\!]$`)`

The second step maps the parameters of the APRIL-function to the parameters of the OCL-function. Whereas the `<body>` part gets translated into an OCL-expression using a relational infix operator with two parameters.

3. `Transaction.allInstances()->select(`$[\![$`transaction`$]\!]$`|`
   $[\![$`destination.institute.country`$]\!][\![$`<>`$]\!][\![$`'`$Germany$`'`$]\!]$`)`
   In this step the AllInstances-function gets translated. In this context the meaning of class name ($[\![< className >]\!]_{CM}$) is defined as element of the set of class names defined in the corresponding UML-class model. The respective class model (CM) spans the graph upon which expressions are specified. Thus, for referring single objects or sets of objects class names, in combination with certain functions, or role names, attached to association ends, are used. Each name of a UML-class is a sequence of alphanumeric characters with special conventions of ordering described in the UML-specification [4]. In this example the parameter `<className>` is occupied by the value `Transaction`.

4. `Transaction.allInstances()->select(transaction |`
   `destination.institute.country <> 'Germany')`
   At last the low layer tokens get translated. Thus, `<iterator>` (here occupied by value `transaction`) gets translated into `transaction` as user defined iterator variables are adopted unchanged. The path name `destination.institute.country` also gets adopted unchanged. However, note that e.g. APRIL allows to abbreviate navigation paths in some special cases (see Section 5.1), which has to be resolved in a preceding step. The basic operator `<>` (see Appendix B) as well as basic types like integers and strings (here `'Germany'`) also get adopted as they are. Hence, the final translation step results into the OCL expression.

For a more detailed insight into the translation examples of APRIL to OCL the reader may be referred to Appendix C.

5.9 Short summary of the core concepts of APRIL

Our main aim is to enable the user to specify formal business rules in a structured, clear and natural language like way. By now we have elaborated the syntax and semantically OCL-based foundation to specify static constraints based on UML class models. A critical issue is also to provide mechanisms for decomposing rules e.g. via Definitions (see Section 5.3) that encourages the employment of methods of separation of concerns well known to existing programming paradigms. However, a big difference between APRIL and its target language OCL is that constraints are defined component centric in OCL. That means every invariant upon a class is defined in the same OCL constraint. Molina [15] shows that a separation of concerns at subcomponent level like UML classes is useful to reduce complexity at design time. Predicate logic is the basis for APRIL's expressive power. Another feature for forming user-defined natural language like

expressions to declare and refer to sub expressions is that APRIL makes use of definition names with a mixfix notation. We believe that allowing the user to intelligently compose mixfix named operations helps to make natural language statements. Halpin [23] also supports this attitude towards mixfix naming. Another yet important point that significantly contributes to the expressive power of APRIL is to involve language concepts that simplify the specification of frequently used constraints in practical modelling. The importance of common constraints in practical modelling is underlined by the work done in [1, 7, 9, 23, 25]. They elaborated many of the common constraints used in real world scenarios and grouped them into a taxonomy depicted in Figure 3. APRIL uses a static type system that is equal to that of OCL.

5.10 Towards a methodology for defining APRIL Statements

Section 5.2 shows an APRIL-Statement that is very near to its natural language business rule. Being able to formulate statements like this, does not come for free simply because of the use of APRIL. The intelligence of making use of natural language is delegated to the user while APRIL only takes the helper role of annotating the formal semantics behind the user-defined expressions. Thus, a first step towards formulating a rule statement that is close to a natural language sentence is to make aware which entities are concerned by that rule. In our example from Section 5.2 these entities of the types `Customer`, `Transaction`, `Account` and `Bank` as well as the attribute `fees` of the class Transaction. At least these concepts have to be in the sentence to make the natural language semantics and the APRIL-semantics go congruent. The second step is to consider a natural language refinement of these concepts for making them reusable e.g. in an APRIL-Definition construct. E.g. "Customers in Germany" filters all Customers having a bank-account at a German bank. The next step is to think about the natural language composition of these sentence fragments that are to be cast into an operation name of an APRIL-Definition. It is important to bear in mind that formal semantics aspects have to be met also. E.g. if an operation `wealthy male senior (customers as Collection of type Customer)` takes a generic collection specified by type Customer as parameter it is obligatory that the nested operation `(customers as Collection of type Customer) in Germany` yields this type. The last step is to specify the APRIL semantics behind the concepts as exemplified in the other APRIL-Definition expressions in Section 5.2. Our experience in sticking to this methodology is that it can sometimes be time consuming especially for very extensive statements. The reason is that more than one iteration loop have to be conducted before a good result is generated. Thus, future work will also focus on a tool based aid for composing typed APRIL-Definitions to a sentence.

6 Conclusion and Future Work

With AtomsPro we aim to achieve more effectiveness in the software development process. This is done by means of the model driven software development

paradigm to support the automatic generation of significant parts of the application based on domain- and architecture-models as well as an innovative language for the notion of business constraints. The domain models bear the actual business logic and are on a higher level of abstraction than the code, which is generated from them. A significant enhancement towards an increase in maintenance and reusability is the use of the architecture and deployment model, which structure the domain logic. Another yet significant issue is to contribute to the needs of domain experts who shall be enabled to formally express business rules upon parts of the domain model by means that are almost like natural language. Therefore, we introduce APRIL. However, different tasks are still open especially in terms of a tool-support for APRIL:

- Evaluation of compiler technologies on how to handle mixfix definitions effectively.
- Investigations on appropriate tool support for specifying business rules are targeted. It might show up that innovative IDE-concepts are necessary.
- Based on the previously defined semantics and the already available parser a compiler which translates APRIL expressions to OCL expressions has to be realized.
- Also methods for recomposing APRIL statements into consistent and performance oriented OCL-constraints have to be evaluated as they can have a strong impact for further processes in a tool chain.
- In parallel some industrial case studies have to be carried out to determine whether APRIL expressions can be easily defined and understood. So it has to be checked on more real life examples how realistic business rules can be expressed by APRIL and if this can be done in a more understandable way than OCL.
- Comprehensive case studies for validating the methodologies for the efficient use of AtomsPro as well as APRIL

References

1. D. Costal, C. Gómez, A. Queralt, R. Raventós, E. Teniente: Facilityting the Definition of the General Constraints in UML, Springer, Berlin Heidelberg, 2006
2. Markus Schacher, Patrick Graessle: Agile Unternehmen durch Business Rules, Springer Verlag Berlin Heidelberg 2006
3. Peter Coad, Edward Yourdon: Object Oriented Analysis, Prentice Hall; 2 edition (1 Nov 1990), ISBN-13: 978-0136299813
4. OMG: UML, http://www.uml.org/; as integral part: OCL, http://www.omg.org/technology/documents/formal/ocl.htm
5. J. Rumbaugh, I.Jacobson, G.Booch: The Unified Modeling Language Reference Manual, 2. Edition, Addison-Wesley, 2005
6. Markus Voelter, Thomas Stahl et.al. : Model-Driven Software Development, John Wiley & Sons, 2006
7. Terry Halpin, Business Rule Verbalization, Information Systems Technology and its Applications, 3rd International Conference ISTA 2004 pp 39-52, Salt Lake City, Utah
8. ACME Team, Acme-Project web site: http://www.cs.cmu.edu/~acme/

9. Elita Miliausikaite, Lina Nemuraite: Taxonomy of Integrity Constraints in Conceptual Models. IADIS Virtual Multi Conference on Computer Science and Information Systems 2005.
10. Mugurel T. Ionita, Dieter K. Hammer, Henk Obbink: Scenario-Based Software Architecture Evaluation Methods: An Overview http://wwbiw.win.tue.nl/oas/architecting/aimes/papers/ Scenario-Based SWA Evaluation Methods.pdf
11. André van der Hoek, Marija Rakic, Roshanak Roshandel, Nenad Medvidovic : Taming Architectural Evolution, ACM SIGSOFT Software Engineering Notes, Vol 26, pp 1-10, 2001
12. D. Garlan, R. Allen, J.Ockerbloom. "Exploiting Style in Architectural Design Environments" In Proceedings of SIGSOFT 1994: Foundations of Software Engineering, pp. 175-188, New Orleans, Luisiana, USA, 1994.
13. Lei Ding, Nenad Medvidovic: Focus: A Light-Weight, Incremental Approach to Software Architecture Recovery and Evolution
14. David Garlan and Bradley Schmerl. Ævol: A tool for defining and planning architecture evolution. In 2009 International Conference on Software Engineering, May 2009. Accepted for Publication
15. Sagar Chaki, Andres Diaz-Pace, David Garlan, Arie Garfunkel and Ipek Ozkaya. Towards Engineered Architecture Evolution. In Workshop on Modeling in Software Engineering 2009, May 2009. Accepted for Publication.
16. David Garlan. Evolution Styles: Formal foundations and tool support for software architecture evolution. Technical report, CMU-CS-08-142, School of Computer Science, Carnegie Mellon University, June 2008.
17. George Edwards, Chiyoung Seo, Nenad Medvidovic: Model Interpreter Frameworks: A Foundation for the Analysis of Domain-Specific Software Architectures http://csse.usc.edu/~cseo/publication/MIFs.pdf
18. Shang-Wen Cheng, and David Garlan: "Mapping Architectural Concepts to UML-RT" in 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001), Monte Carlo Resort, Las Vegas, Nevada, USA, June, 2001.
19. Martin Fowler, Martin Fowlers Blog web-site:" Language Workbenches: The Killer-App for Domain Specific Languages?", http://martinfowler.com/articles/ language-Workbench.html3
20. E.W. Dijkstra, A Discipline of programming, Prentice Hall, Englewood Cliffs, NJ, 1976
21. Markus Schacher, Patrick Graessle: Agile Unternehmen durch Business Rules, Springer Verlag Berlin Heidelberg 2006
22. F. Aarts, J.Aarts: English Syntactic Structures, Prentice Hall, 1982; ISBN: 978-0080286341
23. Terry Halpin, Verbalizing Business Rules, Business Rules Journal, Part 1-16, 2005
24. Constructing the infrastructure for the knowledge economy : methods and tools, theory and practice, ed. by Henry Linger et.al., New York [u.a.] : Kluwer Acad./Plenum Publ., 2004. - XIV, ISBN 0-306-48554-0
25. Elita Miliausikaite, Lina Nemuraite: Representation of Integrity Constraints in Conceptual Models, Information Technology and Control, 2005, Vol.34, No.4
26. itemis, xtext- tool : http://xtext.itemis.de/
27. JetBrains, Meta Programming System MPS: http://www.jetbrains.com/mps/index.html
28. MetaCase, MetaEdit+: http://www.metacase.com/
29. OMG, Model Driven Architecture: http://www.omg.org/mda/

30. microTool, ObjectIF: http://www.microtool.de
31. Eclipse open plattform, Open Architecture Ware: http://www.openarchitectureware.org/
32. AndroMDA: http://www.andromda.org
33. Joanna Chimiak-Opoka, Birgit Demuth, Darius Silingas, et. al. : Requirements Analysis for an Integrated OCL Development Environment, ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems, OCL Workshop 2009, Denver, Colorado, USA, 2009
34. Dresden OCL Toolkit http://dresden-ocl.sourceforge.net/
35. Together von Borland, http://www.borland.com
36. Use Tool, state 2008, http://www.db.informatik.unibremen. de/projects/USE/
37. J. Ackermann. Formal Description of OCL Specification Patterns for Behavioral Specification of Software Components. T. Baar, editor, Workshop on Tool Support for OCL and Related Formalisms, Technical Report LGL-REPORT-2005-001, pages 15-29. EPFL, 2005
38. I. Jacobson, M. Christerson, P. Jonsson: Object-Oriented Software Engineering - A Use Case Driven Approach, Addison-Wesley, 1992, ISBN 0-2015-4435-0
39. HQL: http://docs.jboss.org/hibernate/core/3.3/reference/en/ html/query-hql.html
40. Steven Kelly, Juha-Pekka Tolvanen, Domain-Specific Modeling, ISBN: 978-0-470-03666-2, March 2008, Wiley-IEEE Computer Society Press
41. Martin Fowler, Kendall Scott: UML distilled (2nd ed.): a brief guide to the standard object modeling language, Addison-Wesley, 2000
42. Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. "Modeling Software Architectures in the Unified Modeling Language." ACM Transactions on Software Engineering and Methodology, vol. 11, no. 1, pages 2-57 (January 2002).
43. Mohamed Mancona Kandé, Alfred Strohmeier: Towards a UML profile for software architecture descriptions, Proceedings of the 3rd international conference on The unified modeling language: advancing the standard, 2000
44. OMG, Catalog of UML Profile Specifications: http://www.omg.org/technology/documents/ profile_catalog.htm
45. OMG, SBVR 1.0 Specification, http://www.omg.org/spec/SBVR/1.0/
46. Alessandra Giordiani, Alessandro Moschitti: Syntactic Structural Kernels for Natural Language Interfaces to Databases, W. Buntine, M. Grobelnik et. al., ECML PKDD 2009, LNAI 5781, pp. 391- 406, Springer Berlin Heidelberg, 2009
47. M. Wahler, J. Koehler, A. D. Brucker: Model-Driven Constraint Engineering , Proceedings of the Sixth OCL Workshop, OCL for (Meta-)Models in Multiple Application Domains (OCLApps 2006)
48. M. Giese, R. Haehnle, D. Larsson: Rule-Based Simplification of OCL Constraints, Chalmers University of Technology, School of Computer Science and Engineering
49. Nils Anders Danielsson, Ulf Norell: Parsing Mixfix Operators, Accepted for publication in the proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages (IFL 2008)
50. Simon L. Peyton Jones: Parsing distfix operators, Communications of the ACM Volume 29, pp 118-122, 1986, ACM, New York
51. Bernhard Beckert, Reiner Haehnle, Peter H. Schmitt: Verification of Object-Oriented Software The KeY Approach. Lecture Notes in Artificial Intelligence, Springer Berlin Heidelberg, No 4334, 2007, pp 317 333

52. J. Ackermann. Formal Description of OCL Specification Patterns for Behavioral Specification of Software Components. T. Baar, editor, Workshop on Tool Support for OCL and Related Formalisms, Technical Report LGL-REPORT-2005-001, pages 1529. EPFL, 2005.
52. C. J. Date, Hugh Darwen: A Guide to SQL Standard, Addison-Wesley Professional; 4 edition 1996.
53. Jordi Cabot, Raquel Pau, Ruth Raventós: From UML/OCL to SBVR specifications: A challenging transformation, ELSEVIER 2009
54. Marjan Mernik, Jan Heering, Anthony M. Sloane: When and how to develop domain-specific languages, ACM Computing Surveys (CSUR) archive Vol. 37, ACM New York, NY, USA 2005
55. . R.J. Kate, R.J. Mooney: Using string-kernels for learning semanitc parsers, Proceedings of the 21st ICCL and 44th Annual Meeting of the ACL, Sydney, Australia, July 2006, pp. 913-930. Association for Computational Linguistics (2006)
56. . L.S. Zettlemoyer, M. Collins: Learning to map senteces to logical form: Structured classification with probablistic categorial grammars. UAI, pp 658 666 (2005)
57. Y. W. Wong, R. Mooney: Learning for semantic parsing with statistical machine translation. Proceedings of the Human Language Technology Conference of the NAACL, Main Conference, New York City, USA, June 2006, pp 439 446. Association for Computational Linguistics 2006
58. R. Ge, R. Mooney: A statistical semantic parser that integrates syntax and semantics. Proceedings in the Ninth Conference on Computational Natural Language Learning (CoNLL-2005), Ann Arbor, Michigan, June 2005, pp. 9-16. Association for Computational Linguistics 2005
59. Daniel Jurafsky, James H. Martin: Speech and Language Processing, Prentice Hall; 2 edition, 2008, ISBN: 978-0131873216
60. Nello Cristianini, John Shawe-Taylor: An Introduction to Support Vector Machines and Other Kernel-based Learning Methods, Cambridge University Press 2000, ISBN: 0 521 78019 5
61. Erik Sandewall: Representing Natural-Language Information in Predicate Calculus, Stanford Univ Calif Dept of Computer Science., Stanford Artificial Intelligence Laboratory 1970

Christian Bacherler is a project engineer at the Institute of Applied Research which is attached to the University of Applied Sciences Ingolstadt, since October 2007. His major research interests are in the area of automated generation of software code for automation systems, Model Driven Software Development and the formal modelling of domain knowledge utilizing concepts of natural language. Christian Bacherler was born in 1981 in Ingolstadt, Germany. He holds a diploma degree in engineering and business from the University of Applied Sciences Ingolstadt and takes part in an educational programme for postgraduate Students to take their MPhil/PhD thesis at the De Montfort University Leicester, England.

Christian Facchi is Professor for SW Engineering, Distributed Systems and Mathematics at the University of Applied Sciences Ingolstadt, Germany since 2004. His major research interests are in the area of performance analysis and modelling, Car2X Communication, testing of Software and Software Development Process. In those areas he is leading several industrial and public founded research projects. Since 2006 he is a member of the scientific consulting committee of SCOPAR(www.scopar.de).

Dr. Facchi was born in 1964 at Munich, Germany. He holds a doctoral degree and a diploma degree from the Technische Universitaet Muenchen, Germany both for Computer Science. Before he changed to the Fachhochschule Ingolstadt he has been employed at Siemens in the Mobile Phones development department for 9 years. During his employment at Siemens Mobile Phones he was the head of worldwide strategy for SW development environments and the leader of several projects.

Hans-Michael Windisch is Professor for Software Engineering and Technical Computer Science at the University of Applied Sciences Ingolstadt (HI), Germany since 2002. His research interests are in the area of Model Driven Software Development (MDSD) and Software Testing where he is currently taking part in several research projects. Dr. Windisch was born in 1964 in Ingolstadt, Germany. He holds a doctoral degree and a diploma degree in Computer Science from Technische Universitaet Muenchen, Germany. Before joining HI he worked on big software development projects as Software Engineer, Consultant and Project Manager. In 2000 he co-founded eMundo GmbH in Munich, a Software and IT Consulting company specializing on developing Enterprise IT Systems based on Java. He is one of eMundo's managing directors ever since.

A Syntax of APRIL in EBNF

Grammar excerpt of the xtext-Implementation of APRIL in the version 0.3 .

```
"StartSymbol" = <Model>

<ID>::= ("a".."z" | "A" .. "Z" | "_") {"a".."z" | "A" .. "Z" | "_" | "0".."9" }
<Number> ::= <Integer> | <Float>
<Integer> ::= ("0".."9") {"0".."9"}
<Float> ::= <Integer>.<Integer>
<Boolean> ::= "true" | "false"
<MixFixName> ::= (<ID> | <MNParamDefinition>) [ "␣" <MixFixName>]
<MNParamDefinition>::= "(" <ID> "as" <ID> ")"
<OperationName> ::= <ID> "(" [<ID> ["as" <ID>] {"," <ID> ["as" <ID>"]}]␣")"
<ClassName>␣::=␣<ID>
<PathNameOrDefinionCall>␣::=␣"this"␣|␣[<preExpression>]␣<NavigationPath>␣|␣<MixFixName>
<NavigationPath>␣::=␣<ID>␣{"."␣<ID>}
<AttributeOrClassNameList>␣::=␣<NavigationPath>␣\{","␣<NavigationPath>\}
<preExpression>␣::=␣"former value of"␣<ID>
```

A.1 Rules

```
<Model> ::= (<Definitions> | <Rules>) {<Definitions> | <Rules>}

<Rules> ::= <Invariant> | <PreOrPostCondition>

<PreOrPostCondition> ::=
("Precondition" | "Postcoindition") <ID> "concerns" <OperationName> ":" <Rule>

<Invariant> ::= "Invariant" <ID> "concerns" <ClassName> ":" <Rule>

<Definitions> ::= <FilterDefinition> | <ExtensionDefinition> | <ValueDefinition>

<FilterDefinition> ::=
"Filter" <MixFixName> ["yielding" <ID>] "is␣defined␣as" <Rule>

<ExtensionDefinition> ::=
"Class␣attribute" <pathName> ["yielding" <ID>] "is␣defined␣as" <Rule> |
"Class␣operation" <OperationName> ["yielding" <ID>] "is␣defined␣as" <Rule>

<ValueDefinition> ::= "Value" <MixFixName> ["yielding" <ID>] "is␣defined␣as" <Rule>

<Rule> ::= <BooleanExpression> <RuleEnd>

<RuleEnd> ::= ("." | <LocalVariableDefinitionBlock>)

<LocalVariableDefinitionBlock> ::=
"with" <LocalVariableDefinition> {"," <LocalVariableDefinition>} "."

<LocalVariableDefinition> ::= <ID> "is␣defined␣as" <BooleanExpression>

<BooleanExpression> ::= <AndExpression> { ("or" | "implies␣that") <BooleanExpression>}
<AndExpression> ::= <RelationalExpression> {"and" <AndExpression>}

<RelationalExpression> ::=
<AdditionExpression> {(">" | "<" | ">=" | "<=" | "=" | "<>") <RelationalExpression>}

<AdditionExpression> ::=
<MultiplicationExpression> {"+" <AdditionExpression>}

<MultiplicationExpression> ::=
<NagationExpression> {("*" | "/") <MultiplicationExpression>}
```

```
<NegationExpression> ::= ["-" | "not" | "it␣is␣not␣the␣case␣that"] <Value>

<Value> ::= <Boolean> |
<ID> |
<Number> |
"(" <BooleanExpression> ")" |
<PathNameOrDefinitionCall> |
<Functions>

<Functions> ::= <SetFunctions> |
<BoolFunctions> |
<ValueFunctions>
```

A.2 Function Lexicon

```
<BoolFunctions> := <ForAll> |
<Exists> |
<IsEmpty> |
<IsNotEmpty> |
<Unique> |
<Includes> |
<Excludes> |
<Equivalence>

<ForAll> ::=
"every" <ID> [, <ID>] ["in" <PathNameOrDefinitionCall>] "satisfies␣that" <BooleanExpression>

<Exists> ::=
"at␣least␣one" <ID> ["in" <PathNameOrDefinitionCall>] "satisfies␣that" <BooleanExpression>

<IsEmpty> ::= <PathNameOrDefinitionCall> "is␣empty"
<IsNotEmpty> ::= <PathNameOrDefinitionCall> "is␣not␣empty"

<Unique> ::= <PathNameOrDefinitionCall> "is␣unique" |
<PathNameOrDefinitionCall> "is␣unique␣in" <Value>

<Includes> ::= <PathNameOrDefinitionCall> "is␣in" <Value>
<Excludes> ::= <PathNameOrDefinitionCall> "is␣not␣in" <Value>

<Equivalence> ::=
"for␣every" [<ID> "in"] <PathNameOrDefinitionCall>
"all␣or␣none␣of␣the␣following␣holds␣:" <BooleanExpression> {"," <BooleanExpression>}


<ValueFunctions> ::= <CountFunction> |
<SumFunction> |
<AtFunction> |
<LastFunction>

<CountFunction> ::= "number␣of" <ID> "in" <PathNameOrDefinitionCall>
<SumFunction> ::= "sum␣of" <ID> "from" <PathNameOrDefinitionCall>

<AtFunction> ::=
"element␣at␣position" <Integer> "in" (<PathNameOrDefinitionCall> | <SetFunctions> )

<LastFunction> ::=
"element␣at␣last␣position␣in" (<PathNameOrDefinitionCall> | <SetFunctions> )

<SetFunctions> ::= <SelectFunction> |
<AllInstancesFunction> |
<ReachableObjectsFunction> |
<UnionFunction> |
<IntersectionFunction> |
<EachCombinationFunction> |
<WithoutFunction> |
```

```
<CollectionConversionFunctions>

<SelectFunction> ::= "each" <ID> ["in" <PathNameOrDefinitionCall>] "where" <BooleanExpression>
<AllInstancesFunction> ::= "all␣instances␣of" <ID>
<ReachableObjectsFunction> ::= "reachable␣objects␣along" <ID>

<UnionFunction> ::=
"union␣of" <PathNameOrDefinitionCall> "with" ( <SetFunctions> | <PathNameOrDefinitionCall> )

<IntersectionOf> ::=
"intersection␣of" <PathNameOrDefinitionCall> "with" ( <SetFunctions> | <PathNameOrDefinitionCall> )

<EachCombinationFunction> ::= "each" <AttributeOrClassNameList> "combination"

<WithoutFunction> ::=
<PathNameOrDefinitionCall> "without" ( <PathNameOrDefinitionCall> | <SetFunctions> )

<CollectionConversionFunctions> ::= <Set> |
<Bag> |
<Sequence> |
<OrderedSet>

<Set> ::= "collection" <PathNameOrDefinitionCall> "as␣set"
<Bag> ::= "collection" <PathNameOrDefinitionCall> "as␣bag"
<Sequence> ::= "collection" <PathNameOrDefinitionCall> "as␣sequence"
<OrderedSet> ::= "collection" <PathNameOrDefinitionCall> "as␣ordered␣set"
```

B Operations on Basic Types

The following table depicts functions on basic types in APRIL and their translation into OCL. Although APRIL strives to utilize natural language the functions are kept in a simple mathematical style. The reason for that is, that it would be an unacceptable trade off concerning understandability compared to clearness. As almost everyone is aware of the meaning of the commonly accepted mathematical and logical symbols there is no need for re-writing them in natural language.

APRIL	OCL	Signature
`and`	`and`	`Boolean X Boolean -> Boolean`
`or`	`or`	`Boolean X Boolean -> Boolean`
`implies that`	`implies`	`Boolean X Boolean -> Boolean`
`not |` `it is not the case`	`not`	`Boolean -> Boolean`
`-`	`-`	`Integer->Integer ;` `Real->Real`
`+`	`+`	`Integer X Integer -> Integer;` `Real X Real -> Real`
`-`	`-`	`Integer X Integer -> Integer;` `Real X Real -> Real`
`*`	`*`	`Integer X Integer -> Integer;` `Real X Real -> Real`
`/`	`/`	`Integer X Integer -> Integer;` `Real X Real -> Real`
`mod`	`mod`	`Integer X Integer -> Integer;` `Real X Real -> Integer`
`<`	`<`	`Integer X integer -> Boolean;` `Real X Real -> Boolean`
`<=`	`<=`	`Integer X integer -> Boolean;` `Real X Real -> Boolean`
`>=`	`>=`	`Integer X integer -> Boolean;` `Real X Real -> Boolean`
`>`	`>`	`Integer X integer -> Boolean;` `Real X Real -> Boolean`
`=`	`=`	`Integer X integer -> Boolean;` `Real X Real -> Boolean`
`<>`	`<>`	`Integer X integer -> Boolean;` `Real X Real -> Boolean`
`floor`	`floor`	`Real -> Integer`
`concat`	`concat`	`String X String -> String`
`size`	`size`	`String -> Integer`

C Examples for translating APRIL functions into OCL

The following table gives an overview of the basic functions in APRIL and how to translate them into OCL.

APRIL	OCL
"all instances of" <ID>	$< ID >$ ->allInstances()
number of <iterator> in <source>	<source>->count<iterator>
number of <source>	<source>->size()
sum of <source>	<source>->sum
sum of <source> from <body>	<source>->select(<body>)->sum()
union of <source> with <CollectionExp>	<source>->union(<CollectionExp>)
union of <source> with <SingleObjExp>	<source>->including(<SingleObjExp>)
NavigationPath := <ID>{"." <ID>}	<NavigationPath> := <ID>{"." <ID>}
each <iterator> [in <source>] where <body>	<source>->select([<iterator>|]<body>)
intersection of <source> with <body>	<source>->intersection(<body>)
every <iterator> [,<iterator2>] [in <source>] satisfies that <body>	<source>->forAll(<iterator>[,<iterator2>] | <body>)
at least one <iterator> [in <source>] satisfies that <body>	<source>->exists(<iterator> | <body>)
<source> is not in <SingleObjExp>	<source>->excludes(<SingleObjExp>)
<source> is not in <CollectionExp>	<source>->excludesAll(<CollectionExp>)
<source> is in <CollectionExp>	<source>->includesAll(<CollectionExp>)
<source> is in <SingleObjExp>	<source>->includes(<SingleObjExp>)
<operator1> = <operator2>	<operator1> = <operator2>
<source> is empty	<source>->isEmpty()
<source> is not empty	<source>->notEmpty()
<source> is unique in <body>	<body>->isUnique(<source>)
collection <body> as set	<body>->asSet()
collection <body> as sequence	<body>->asSequence()
collection <body> as bag	<body>->asBag()
collection <body> as ordered set	<body>->asOrderedSet()
element at position <pos> in <body>	<body>->at(<pos>)
element at last position in <body>	<body>->last()

Another central part in specifying requirements is to be able to express common constraints according to Halpin et.al. [7]. For that reason APRIL introduces special language constructs to express common constraints that are easily specified in natural language (adopted into APRIL) and at the same time are very expensive in OCL. The following table gives a brief overview of those common constraints expressed in APRIL as well as its corresponding version in OCL.

APRIL	OCL
each <AttributeOrClassNameList> combination is unique	<ClassName>.allInstances->forAll(c1,c2 | implies not(c1.<attribute> = c2.<attribute> and c1.<attribute2> = c2.<attribute2> {and c1.<attributeN> = c2.<attributeN>}))
each <AttributeOrClassNameList> combination is unique	value(<AttributeOrClassNameList>,1). allInstances->forAll(c1 | value(<AttributeOrClassNameList>,2). allInstances->forAll(c2 | [value(<AttributeOrClassNameList>,n). allInstances->forAll(cN |] ASSOC.allInstances->select(assoc | assoc.value(<AttributeOrClassNameList>,1) = c1 and assoc.value(<AttributeOrClassNameList>,2) = c2 [and assoc.value(<AttributeOrClassNameList>,n) = cN)]-> size()<=1[]))) // using helper function: value(CommaSeparatedList::String, index::Integer)::String { list::Array = new Array() list = CommaSeparatedList.splitBy(",") return list[index] }
for every element [in <source>] all or none of the following holds: <Expression1>, <Expression2> {, <ExperssionN>}	[<source>->](<Experssion1> and <Expression2> { and <ExpressionN>} or not (<Expression1> or <Expression2> { or <ExpressionN>}))
for every element [in <source>] exactly one of the following holds: <Expression1>, <Expression2> {, <ExperssionN>}	[<source>->] ((not (<Expression1>) and <Expression2> {and <ExpressionN>}) or (<Expression1> and not (<Expression2>) {and <ExpressionN>}) or ... (<Expression1> and <Expression2> and not (ExpressionN)))
reachable objects along <PathName> <SetFunctions(bodyOfSetFunction)>	context <Classname> def: <PathName>_successors(): Set(<Classname>) = self.<PathName>->union(self.<Pathname>.successors()) self.<PathName>_successors()-> <SetFunctions>(<bodyOfSetFunction>)

HOCHSCHULE
UNIVERSITY OF
APPLIED SCIENCES
INGOLSTADT

Impressum

Herausgeber
Der Präsident der
Hochschule für angewandte
Wissenschaften FH Ingolstadt
Esplanade 10
85049 Ingolstadt
Telefon: 0841 9348-0
Fax: 0841 9348-200
E-Mail: info@haw-ingolstadt.de

Druck
Hausdruck

Internet
Dieses Thema können Sie, ebenso wie die früheren Veröffentlichungen aus der Reihe „Arbeitsberichte – Working Papers“, unter der Adresse www.haw-ingolstadt.de nachlesen.

ISSN 1612-6483