Supporting Test Code Generation with an Easy to Understand Business Rule Language

Christian Bacherler, Ben Moszkowski
Software Technology Research Lab
DeMontfort University
Leicester, UK
christian.bacherler@email.dmu.ac.uk, benm@dmu.ac.uk

Christian Facchi
Institute of Applied Research
Ingolstadt University of Applied Sciences
Ingolstadt, Germany
christian.facchi@haw-ingolstadt.de

***Abstract*—The paper addresses two fundamental problems in requirements engineering. First, the conflict between understandability for non-programmers and a semantically well-founded representation of business rules. Second, the verification of productive code against business rules in requirements documents. As a solution, a language to specify business rules that are close to natural language and at the same time formal enough to be processed by computers is introduced. A case study with 30 test persons indicates that the proposed language caters to a better understandability for domain experts. For more domain specific expressiveness, the language framework permits the definition of basic language statements. The language also defines business rules as atomic formulas, that are frequently used in practice. This kind of constraints is also called common constraints. Each atomic formula has a precise semantics by means of predicate or Interval Temporal Logic. The customization feature is demonstrated by an example from the logistics domain. Behavioral business rule statements are specified for this domain and automatically translated to an executable representation of Interval Temporal Logic. Subsequently, the verification of requirements by automated test generation is shown. Thus, our framework contributes to an integrated software development process by providing the mechanisms for a human and machine readable specification of business rules and for a direct reuse of such formalized business rules for test cases.**

Keywords*—*Requirements engineering; business rules; common constraints; natural language; testing; logic.

I. INTRODUCTION

In software development, different stakeholders with different knowledge and intention cooperate, typically domain experts and developers. Requirements engineers are acting as negotiators between these two worlds and prepare requirement specifications in a way that can be understood by both sides. Unstructured natural language in requirements documents does not ensure identical interpretations by different stakeholders, especially by domain experts and developers. In order to overcome the problem of divergence between specification and implementation we proposed *AtomsPro Rule Integration Language* (APRIL) [1] [2], a business rule language that is both, understandable enough to domain experts and translatable to executable representations. To raise the expressiveness of APRIL we have also defined a framework to add new language constructs.

By the introduction of APRIL, we propose a means to develop a formalized version of business rules specifications by precise semantics that support human- as well as machine-readability. The APRIL statements representing business rules are easy to design and can be customized by the construction of tailored statements, a feature, which we introduce via a novel combination of pattern building mechanisms. In this paper, we show how to utilize the framework for extending APRIL's expressiveness using atomic formulas that constitute the link between statements that are like natural language and formal frameworks. Moreover, we also present some common constraints that are incorporated as atomic formulas.

Formal specifications enhance the established software development process. As a general advantage, such specifications allow consistency checking of business rules, e.g., reveal conflicts or proof properties. The aspect we want to focus on in this work is based on the fact that in the established software development process, code and corresponding tests are developed based on the natural language specification. In order to reduce complexity of the development process, we support automated creation of tests based on formal APRIL statements representing business rules. With our method, human understandable formal specifications can be used to directly generate formal logical conditions and behavior specifications for testing. This approach shifts the creation of the test code from the developer to the requirements engineer, which helps to improve test-driven development projects [3] [4].

The paper is structured as follows: Section II gives an impression of the context and the facets of the work presented. Section III presents the framework for our language to describe business rules close to natural language. After laying down the fundamentals, we demonstrate in Section VII the transformation of example statements in our language into computer processable test code. In Section IV, we present the utilization of the extension mechanism to incorporate a set of frequently used constraints, known as common constraints, into APRIL. Section VII-B deals with usability aspect of APRIL, explored in a case study. After the discussion of related work (Section VIII), a conclusion will be drawn and future work will be presented (Section IX).

II. OVERVIEW

The APRIL framework can be embedded into standard software development processes. As an example, the seamless integration into the V-Model is shown in Figure 1. Aspects that will be detailed in this paper are highlighted in dark grey.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

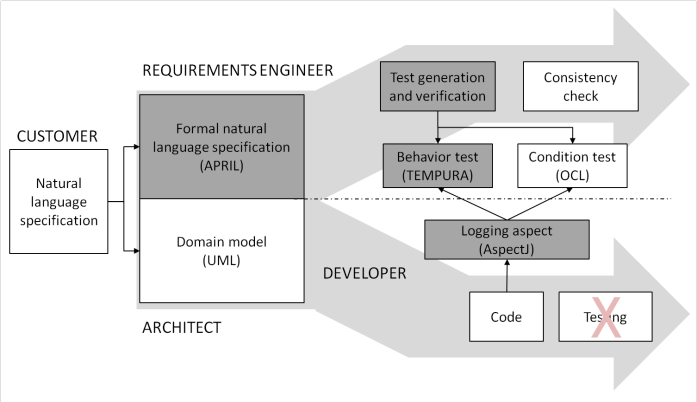

Fig. 1. Overview of the software development process using APRIL.

Next to the clear definition of business rules, our framework aims at supporting the generation of computer executable test code from formal specifications that are close to natural language and thus enable the verification of the productive code against the original user specification. In Section III, a detailed explanation of the substantial concepts of the APRIL language is given, exemplifying the formalization of business rules as APRIL statements in Section III-A. Section VII-B presents the results of a case study that shows that ARPIL is understandable even to untrained test persons. The specification of complex real-world business rules using mix-fix notation and decomposition into reusable sub-statements (APRIL-Definitions) is presented in Section III-B. Section III-C deals with support for customizing parts of the language using so-called atomic formulas. These are verbalized versions of operations on sets, predicate-logic formulas and special common constraints and provide a precise semantics for APRIL Definitions. In Section IV we present common constraints that are incorporated into APRIL using the extension mechanism to add atomic formulas. In practice, these frequently used constraints can make up a significant part of the overall constraints defined on a software system.

Tests based on APRIL statements can be generated to check conditions using invariants, pre- and post-conditions in the Object Constraint Language (OCL) [5] notation. Checking process behavior is done by the use of a subset of Interval Temporal Logic (ITL) called Tempura [6]. The rationale for applying our testing-framework is laid down in Section VII-A. Section VII-B presents the testing-framework by example, taking into account the significant concepts for defining a custom atomic formula for modeling a simple example-process and the relation to the semantic frameworks presented in Section VI. This section will also include a presentation of the automated test generation for behavior testing using Tempura. Due to space limitations, the detailed presentation of generating OCL-statements is omitted and can be reviewed in [2]. Some translation examples are shown alongside the introduction of the APRIL language.

After the discussion of related work (Section VIII), a conclusion will be drawn and future work will be sketched (Section IX).

III. The APRIL framework - Specifying business rules in formal natural language

Business rules are restrictions of certain object constellations and behaviors based on domain models [7]. Typically in software development, requirements engineers produce business rules in natural language and hand them to developers along with the respective domain models to enable the development of a software-system compliant to these input artifacts. Mostly, those natural language business rules are informal and suffer from ambiguity and imprecision. Therefore, we introduced APRIL, which is a language to specify business rules, close to natural language and such is easy to use. On the other hand, APRIL has a formal semantics, which is based on OCL and in consequence, an unambiguous description of business rules is possible.

A. Business Rules in APRIL

In general, the different types of business rules in the industrial practice are: Integrity Rules, Derivation Rules and Rules to describe behavior [8]. Despite the fact that there are fundamental intentional differences, these rule types have one aspect in common: The description of the semantics of parts of the real world into formal representations by means of logic. In APRIL, we use UML-class models [9] to formally represent business domain models. The reason is that the UML-class model is widely used for representing conceptual schemas and is easily understood by people. APRIL requires UML-class models as the domain of discourse to specify business rules as constraints, which are of the following types: **invariant, pre-, post-condition and behavioral rules**. Invariants describe allowed system states that must not be violated during any point in time. This is unlike the pre- and post-conditions, which have a restricted scope right before and after a transition. The fourth rule type describes behavior explicitly. Behavioral rules can describe operations lasting over multiple state transitions [7], which is not possible with a single pair of pre- and post-condition.

In Figure 2, a simple domain model of an order system, with the basic concepts Order, Customer Shipment, Vehicle and Product is shown as UML-class model. As an example of

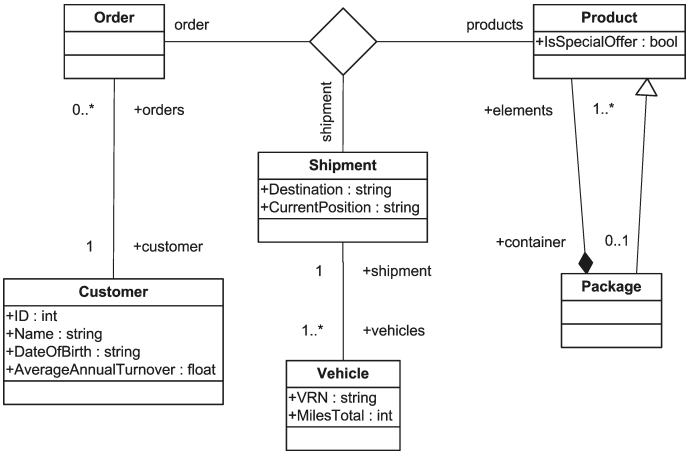

Fig. 2. UML-model of the example domain model.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

APRIL usage on the class model, the corresponding statement for the invariant rule1 can be seen in Listing I.

```
1  Invariant rule1 concerns Customer:
2      A premium customer who buys a special offer must pay
3      0 EURO for the shipment of that order.
```

Listing I. TOP-LEVEL RULE, COMPOSED OF SEVERAL APRIL DEFINITIONS.

The header (line 1) of a rule contains its name *(rule1)* and the token after the keyword **concerns**, which represents the context set (represented by the class name *Customer*) of the business rule to which the formula after the colon applies. With respect to UML-models, the context in invariant rules is represented by a class name and by a qualified method name in the case of pre- and post-conditions respectively. The rule body (lines 2-3) contains the actual business rule. In order to use a natural language sentence in the needed formal way, a couple of definitions have to be installed, which are explained in Section III-B continuing this example. Moreover, a detailed specification of APRIL including default logic- and set- operators, is given in [2].

The mathematical representation of the rule can be expressed as predicate logic formula as follows.

$$\forall c \in \{\text{Customer}_{allInstances}\}\ (\text{isPremium}(c)) \implies$$
$$\exists p \in \{c.\text{orders.product}\}\ \text{isSpecialOffer}(p) \implies$$
$$\forall o \in \{c.\text{orders}\}\ \{\text{isSpecialOffer}(o.\text{product}) \wedge$$
$$\text{isFreeOfCharge}(o.\text{shipment})\})$$

B. APRIL-Definitions

APRIL Definitions are special mix-fix operators, which allow the intuitive construction of patterns that decompose large business rules into smaller, comprehensible and reusable sub-statements. Mix-fix is a particularly useful technique to form natural language statements [10]. Mix-fix operators allow to compose an operator's constants and placeholders in arbitrary order. The design of the APRIL-Definition's headers is based on sequences of static name parts and placeholders. Both static name parts and placeholders can be arbitrarily composed to express a business statement reflected as a natural language sentence pattern. This makes them particularly easy to construct for humans [11]. The below given example D.1, shows a definition signature between the *Definition* and the *yielding* keyword. Here placeholders, wrapped by the outmost brackets, and keywords are mixed together to constitute a pattern. Sentences based on the pattern come close to a natural language sentence, when the placeholders get filled out with the correct concepts of the domain model.

Despite the convenience that mix-fix operators provide to humans, it is quite challenging to implement the parser logic [12], especially for nested definition calls. The problem is that the parser has to recognize a *definition call* embedded inside an ID-token sequence in what is in the grammar specification another *definition call* (see highlighted EBNF-grammar rules in Listing II). As a consequence, a conventional context free grammar provides only insufficient means to specify sub ID-token streams with a different semantics to their embedding ID-token streams. To overcome this, we use the ANTLR v3 [13] parser-/compiler-generator framework. The framework allows to specify semantic annotations [14], which is actually user defined code (e.g., in Java), that gets inserted into the proper positions of the grammar to guide parser decisions, based on the semantics of tokens. Consider Listing II, where the Boolean return-values of the semantic annotations indicated by α_0 and α_1 influence the generated parsers resolution algorithm. The semantic annotations indicated by the symbols α_n represent Java code that gets integrated into the parser. The implemented logic performs the link between syntax and semantics. For instance, when a token with the value *Customer* gets recognized, the semantic annotation allows to conclude on further decision steps for the parser. Or also trigger some type-checking mechanism. However, for parsing mix-fix operators, we limit the nesting depth to three, which was shown to be sufficient in our preliminary case study.

```
definition::= 'Definition' nameSignature 'yielding'
              typeDef 'is defined as' ruleBody '.'
nameSignature::= (ID | parameterDef)+
parameterDef::= '(' name=ID 'as' type=ID ')';
typeDef::= ID | ID '(' typeDef ')';
ruleBody::= statement+ ;
statement::= ... | referenceOrDefinitionCall | ...;
referenceOrDefinitionCall::= {α0}modelReference
                           |{α1} definitionCall | ...;
definitionCall::= ID (ID | referenceOrDefinitionCall)* ;
```

Listing II. GRAMMAR SNIPPET FOR APRIL DEFINITIONS

Given the example from Section III-A, the APRIL-Definitions (D.1)-(D.3) decompose the business rule statement from Listing I into reusable and easy to define sub-statements with a signature in mix-fix notation.

(D.1) ***Definition*** *All (customers* ***as*** *Collection(Customer)) who buy (products* ***as*** *Collection(Product)) must pay (price* ***as*** *Number) EURO for the shipment* ***yielding*** *Boolean*
is defined as
every *customer* ***satisfies that every*** *"ordered product"* ***satisfies that*** *shipment.fee = prize*
with
"ordered products" (orderer as Customer) ***is defined as***
each *product* ***where*** *product.order.customer = orderer.*

(D.2) ***Definition*** *premium customer* ***yielding*** *Collection(Customer)*
is defined as
each *customer* ***in all instances of*** *Customer* ***where*** *customer.AverageAnnualTurnover > 20,000 .*

(D.3) ***Definition*** *special offer* ***yielding*** *Collection(Product)*
is defined as
each *product* ***in all instances of*** *Product* ***where*** *product.IsSpecialOffer.*

In (D.1), the orders of specific customers are mapped to a shipment prize. On the other hand, (D.2) is a set-comprehension on the set of all customers defining, what a premium customer is. Furthermore, (D.3) defines attributes that characterize special offers.

In order to provide a precise semantics, APRIL **atomic formulas** are used. They are verbalized versions of operations on sets, predicate-logic formulas and special common constraints sketched by Halpin [10]. For example, the *every-satisfies-that*-statement of Definition (D.1) is an atomic formula in APRIL that constitutes a universal quantification that is by default incorporated into the language. Some more operators are

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

described in [2]. Default atomic formulas are for maintaining sufficient expressive power and straight-forward translation into executable representations.

Moreover, we have defined some syntactical rules for the default atomic formulas to make their syntax a bit more appealing. For example the auto-mapping of plural to singular symbols. Like in D.1, in which the symbol "ordered products" represents a collection of objects of type Product and the symbol "ordered product", which is used as iterator symbol for the univesal quantification. One auto-mapping rule says that if any iterator symbol postfixed with an "s" equals a symbol that is in the scope of the same function or definition the short form, omitting the "in Collection(<Type>)" declarator can be used. This only applies if the types can be resolved and the symbol is unique in the entire scope stack.

In order to resolve symbols from their usage to their definition, APRIL uses different scope levels, e.g., like global and local variables known from the most programming languages. The precendences for resolving symbols are as follows:

- Atomic formulas (with iterator(s))
- Local variable / local method symbols
- Definition signatures (symbolic name with types of parameters)
- Class names and role names from the UML model used in the rule header after the *concerns* keyword

If we consider the example from Listing D.1 the body of the definition contains two nested universal quantification opertors $\forall_1.customer(i_1|P(i_1))$ with $P(i_1) := \forall_2.f_{LM}(i_1)(i_2|P(i_2)))$ and $f_{LM}(i_1) :=$ "ordered product"(i_1) with i_n are iterator variables, bound to the respective universal quantification operator. Note that we have marked the universal quantifiers with indexes, which makes it easier to refer to them later. The following annotated excerpt of D.1 illustrates the earlier definition:

every$_{\forall_1}$ customer **satisfies that** (
every$_{\forall_2}$ "ordered product" **satisfies that** (shipment.fee = prize)$_{P_2}$)$_{P_1}$

In this case the iterators i_1 and i_2 are related by $i_2 \in Ret_1 := f_{LM}(i_1)$, whereas $f_{LM}(i_1)$ is actually defined in the local members (LM) section of definition D.1 after the *with* keyword. Ret_1 is the return value yielded by f_{LM}. That is the local method f_{LM} with the symbolic name "*ordered products*", which takes a single parameter of type *Customer*. The method itself is implicitly typed as collection by the set comprehension function used in the proposition $P_{f_{LM}}$ of its body, which is:

"ordered products" (orderer as Customer) **is defined as**
(**each** product **where** product.order.customer = orderer)$_{P_{f_{LM}}}$.

The inference mechanism of the typing of f_{LM} works as described earlier by simply adding a "s"-postfix of the iterator so the short form of the operator "***each*** *product* ***where*** ..." can be resolved to the conventional form "***each*** *product* ***in*** *products* ***where*** ...". The symbol *products* can be resolved within the scope of D.1 as this is one of the parameters of type "Collection(Product)". Thus, the set comprehension also yields the same type. If we memorize $\forall_1$ that uses the symbol "*ordered product*" as iterator symbol and we also apply the "s"-postfix mechanism then "ordered products" is resolvable as local method. As $\forall_2$ is nested in $\forall_1$ that uses an iterator variable (i_1) represented by symbol *customer* of type Customer, the call to f_{LM} does not necessitate to explicitly state the parameter, which would be the iterator of the surrounding operator i_1 of $\forall_1$. This abbreviation mechanism is similar to that, e.g., used in λ-expressions in C# 4.0. Resuming the body statement of $\forall_2$, the scope stack now adds the symbol "ordered product", which is actually i_2, at its lowest level. Thus, both the immediate short navigation *shipment.fee* and the conventional navigation "*ordered product*"*.shipment.fee* are both valid in this context. We chose the short form for our example. The ability to unambiguously resolve the types of the used symbols is obligatory to detect trivial typing faults during design time of a business rule. Moreover, it is helpful in the translation process into the target language as it gives at least some evidence that the business rule is formally correct.

APRIL uses OCL as target language for translating invariants and pre-and post conditions. Behavioral rules are translated into Tempura, which is briefly explained later.In order to extend APRIL's expressiveness over general purpose operators provided by OCL, we allow the customization of atomic formulas that can be tailored to a certain domain. This delegates the design of the atomic formulas as natural language statements to the human user, who is still the best choice for this creative task.

C. Extending APRIL with Custom Atomic Formulas

Like definitions, customizable atomic formulas are defined using textual business patterns *(bp)*. Here, a requirements engineer can, e.g., reuse his already existing, informal textual business patterns [11], which, unlike the more abstract Definitions, express a very basic business rule- or business process pattern that regulates the business concepts and facts under consideration. For example, if a requirements engineer wants to verbalize business process statements which specify that in a warehouse all elements in a goods-stock move to a dedicated truck-loading bay and have to pass a certain gate on their way, she would have to specify parts of the grammar. Generally, a context free grammar consists of a start symbol, production rules, terminals and non-terminals [15]. Therefore, a state of practice language implementation mechanism described by Parr [14] is used. First, a formal production rule of the new atomic formula must be specified. Formal production rules are used to generate text recognition algorithms of a parser that processes statements of a language to generate a parse tree. Second, a parse tree rewrite rule has to be specified along with the production rule. Parse tree rewrite rules are instructions for the parser on how to construct the *abstract syntax tree* (AST) from the parse tree.

The AST is a condensed version of the parse tree that can be influenced by semantic considerations to form a concise and expressive logical representation of the parsed statements. For APRIL the AST provides the necessary flexibility to incorporate user defined language parts and also makes it particularly easy to extract the necessary parameters for the compiler. For clarification, Listing III sketches the definition of a user defined atomic formula. It formalizes the example operator that reflects the scenario mentioned above. In line 1,

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

the production rule with the name of the non-terminal (atomic formula) *moveTo* is introduced. The definition of the new atomic formula's regular syntax is defined in the lines 2-7. Here, the non-terminal *referenceOrDefinitionCall* is similar to that in Listing II. This non-terminal is a predefined APRIL concept and can either refer to an element of the related domain model (e.g., to class names Store, Bay, Gate) or to values in the scope stack of the parent rule or definition, in which the formula is used. The references to the parse tree nodes of type *referenceOrDefinitionCall* in the lines 3, 5 and 7 are stored one by one in the local variables source, target and routeNode. Line 9 concludes the specification of the grammar rule with the parse tree rewrite rule. It is delimited from the syntax rule by the "→" sign. It tells the parser to construct a tree with the MOVETO-terminal as root node having three leaves: source, target and routeNode.

```
1  moveTo :
2  'all elements in'
3  source=referenceOrDefinitionCall
4  'move to'
5  target=referenceOrDefinitionCall
6  'over'
7  routeNode=referenceOrDefinitionCall
8
9  → ^(MOVETO $source $target $routeNode);
```

Listing III. GRAMMAR RULE AND PARSE TREE REWRITE RULE FOR THE OPERATOR **MOVETO** IN ANTLR 3.0.

The grammar rule and the parse tree rewrite rule in Listing III get injected into dedicated areas of the APRIL core grammar. Parameterization of the APRIL-compiler is straight forward, which is depicted in Figure 3. In the second pass, a so called tree parser interprets the AST (of the rewrite rule MOVETO) and decides, which target language template to apply to the AST of the atomic formula. It then passes the values of the leaf-nodes (here the values of the variables $source, $target and $routeNode) to the parameters of the respective template. The instantiated template is the actual translation of the atomic formula into the target language, representing the semantics of the respective operator. Please see Listing V as an example instantiation.

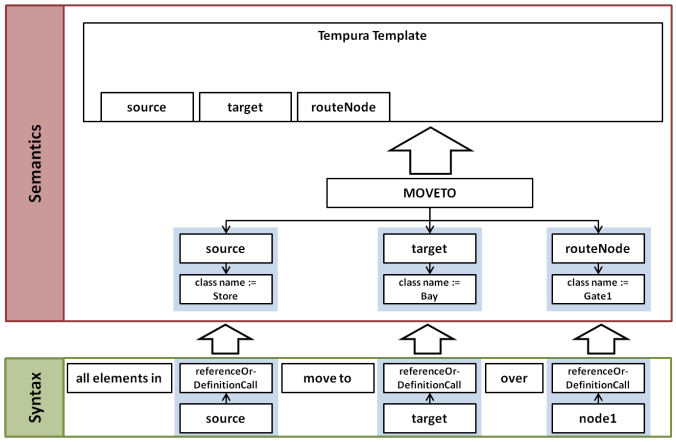

Fig. 3. Translation example of the atomic operator **moveTo**.

IV. FREQUENTLY USED BUSINESS RULES AS ATOMIC FORMULAS

A central aspect that increases the expressiveness of APRIL is the utilization of language constructs that allow to shortly specify business rules. These abbreviations are frequently used in practice and would be partly complicated to formulate in the underlying target languages. Such constraints are also often referred to as common constraints. Costal et.al. [16] show that these types of business rules can cover a significant amount of the overall constraints occurring in real life systems. In order to give the presented common constraints a structure, we have grouped them together based on the taxonomy presented in Figure 4, which was inspired by Halpin et al. [10] and Miliauskaite et al. [17] [18] and will be explained in the following subchapters.

A. Constraints on Values

Restricting values of variables can be done in several ways. For example assigning an integer data type to a variable restricts its values to a given range of natural numbers. Another way is to use relational operators with, e.g., constants to explicitly constrain variables. Therefore, the conventional and well known binary relational operators (e.g.,{<,>,=,<>,<=,>=}) are used. Although APRIL's is meant to be close to natural language, we use the mathematical representation for the afore mentioned operators as atomic formulas as we think this is well known enough to anyone. Moreover, if this might be too disconcerting for a user to use in a language like APRIL, it is possible to redefine that particular part of the grammar to give these operators a natural language syntax (e.g., "A>B" may become "A greater than B"). Here is an example:

Invariant Values **concerns** Vehicle:
MilesTotal<100000 .

B. Identifier

According to Miliauskaite et al. [18], a useful and strongly demanded constraint is the identifier or primary identifier known to ERM [19], ORM [20], xUML [21] and relational database management systems (RDBMS). UML's class attributes are predestined for holding a primary identification rule stated in APRIL or OCL, as UML class diagrams by default lack such means. This can be shown with the help

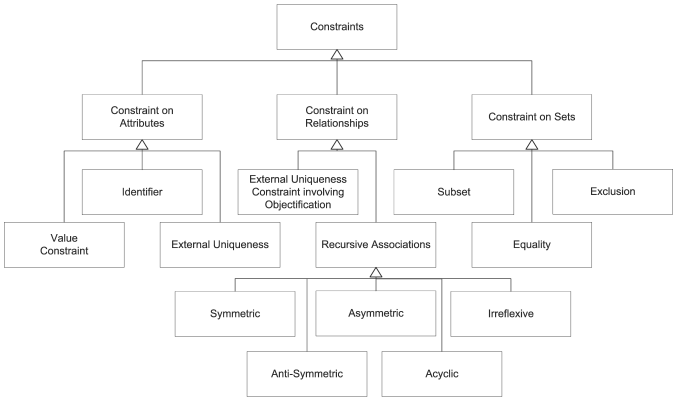

Fig. 4. Taxonomy of some important common constraints.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

of the model in Figure 2. A common scenario is that an object is identified by an attribute that carries a unique value over all objects of the entire population. This can be formalized as follows:

APRIL:

Invariant id **concerns** Customer:
each ID is unique .

OCL:

context id **inv** Customer:
Customer.allInstances→isUnique(ID)

A more general version of the primary identifier constraint is the internal uniqueness constraint also called composed identifier. It says that value combinations of two or more attributes of an object are unique [10] [17]. The APRIL and OCL versions look like:

APRIL:

Invariant composedId **concerns** Customer:
each Name, DateOfBirth combination is unique .

OCL:

context Customer **inv** composedId:
Customer.allInstances→forAll(c1,c2 | c1 <> c2 implies
not((c1.Name = c2.Name and c1.DateOfBirth = c2.DateOfBirth)))

Towards a reasoning of common constraints we can say that if a class has a tuple of attributes $\{a_1,...,a_i\}$ out of which at least one has to obey a primary identifier constraint it is redundant to additionally specify that the combination of $\{a_1,...,a_i,...,a_n\}$ has to obey a composed identifier constraint.

C. External Uniqueness

A uniqueness constraint is denoted as external if the identification scheme is bound to another attribute of an associated class. For example an object *a* may be related to an object *b* only once. This does not restrict the overall occurrence of object *b* throughout the entire model as it might be that *a* is also related to an object *c*.

In the example below, the constraint only holds if different instances of `Vehicle` with potentially equivalent values in `VRN` (vehicle registration number) are not linked to the same instance of `Shipment`. The combination of `Shipment` and the attribute `VRN` of the class `Vehicle` is inherently done by the navigation path from `Shipment` to `Vehicle` by stating its concrete role name, `vehicles`. This collects all instances of `Vehicle` that are linked with the current instance of `Shipment`.

APRIL:

Invariant externalUniqueness **concerns** Shipment:
VRN is unique in vehicles.

OCL:

context Shipment **inv** externalUniqueness:
vehicles→isUnique(VRN)

D. External Uniqueness Involving Objectification

This type of constraints deals with associations that are regarded as objects. Hence, objectification [29], known as reification in UML, aims to combine multiple classes or attributes to a single one in order to apply constraints on the combination. In UML, this is typically done using association classes which objectify the association between two classes [8]. Hence, an object of an association class identifies a unique n-tuple of linked objects. In an attempt to generalize several UML concepts, Gogolla et al. [23] uncover how to transform association classes and association-qualifiers into n-ary associations (with n>=2 at this point). As they show in [24], there are several problems with the use and constraining of n-ary associations and thus, the n-ary association has to be transformed into a proper set of binary associations. For our example in Figure 2 it would mean that the association diamond in the middle of the three associated classes (Order, Shipment, Product) is transformed into an additional synthetic class, e.g., called ASSOC being associated to each of the afore mentioned classes with a binary association. In order to handle objectification in business rule statements between two or more Classes $\{c_1,...,c_N\}$ the APRIL *"each c_1,...,c_N combination"* expression is used. It returns a set of synthetic association objects instanciated from class ASSOC each of which is associated to one object of the corresponding type of c_1,...,c_N. The first example shows how to constrain tuples of classes. We omit the prose explanation for the APRIL constraint here because we consider it to be self explanatory.

APRIL:

Invariant externalUniqueness **concerns** Product:
each Product, Order, Shipment **combination** is unique .

OCL:

context Product
inv externalUniqueness:
Product.allInstances→forAll(c |
Order.allInstances→forAll (b |
Shipment.allInstances→forAll (a |
ASSOC.allInstances→select(assoc | assoc.product = c and
assoc.order = b and assoc.shipment = a)→size()<=1)))

E. Recursive Associations

A UML class can be associated with itself (see class Product in Figure 2). This allows recursions between objects. Rules on such models are called ring constraints [10]. Common ring constrains follow typical association properties. Here are some examples:

- *Irreflexive* constraints do not allow objects to refer to theirselves which is formally stated below. Note that the OCL keyword `self` corresponds to the lower-cased class name in the APRIL rule body.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

APRIL:

Invariant irreflexive **concerns** Package:
package **is not in** elements .

OCL:

context Package **inv**:
elements→excludes(**self**)

- A *transitive* constraint says that if a first object bears the relationship to a second and the second to a third one then the first also bears a relationship to the third. This can be formally stated as follows.
- An *intransitive* constraint is the negation of the coresponding *transitive* constraint.
- *Symmetric* means that if the first bears the relationship to the second, then the second bears the relationship to the first. However, a ring constraint defined in a UML class diagram is by default symmetric so there is no need to state that an association is symmetric if it is meant to be optional. If not the following example shows how it can be stated.
- *Asymmetric* means that if the first bears the relationship to the second, then the second cannot bear the relationship to the first.
- *Anti-Symmetric* means that if the objects are different, then if the first bears the relationship to the second, then the second cannot bear that relationship to the first.
- *Acyclic* means that a chain of one ore more instances, which are linked to objects of the same type cannot form a recursive loop (cycle).

As acyclic constraints are common practice in business rule modelling [10] we can define a new atomic formula. That is the "*deep collection of*"-operator. The notation of the operator is exemplified in the listing below. The semantics of the operator is as follows:

Let A be a set of objects of type τ and $\{a_0 \dots a_n\} \cup \emptyset$ be elements of A. Let $R_n(a_n, A_n := \{a_{n+1,j} \dots a_{n+1,k}\} \backslash \emptyset)$ be the representation of a set of relations between an element a_n and a non-empty set $A_n \subset A$.

Then $A_{deep} := \bigcup A_m(R_m)$; with $0 \leq m \leq n$. After all $(a_0, A_{deep}) \models$ *"deep collection of"*(a_0).

APRIL:

Invariant acyclic **concerns** Package:
package **is not in deep collection of** elements.

OCL:

context Package **def** :
successors(): Collection(Product) =
self.elements→
union(self.elements.successors())
context Package **inv** acyclic:
self.successors()→excludes(self)

In the OCL example, the first constraint defines a synthetic operation named `successor()` that is recursively called within an OCL union operation which is called on the set of `elements` of the current object. The intent is to unite all the `Package` objects linked with the current objects `elements` that also play this role in the linked subordinated objects. This construct is inherently typed as collection type `Collection`.

F. Sets

The upper part of the Table I shows the verbalization of common constraints on sets according to Costal et al. [16] and Miliauskaite et al. [18]. Note that lower case latters denote elements of sets and upper case letters denote sets.

The lower, folded part of the table handles a specialization of the natural-join operator, indicated by $\bowtie'$. In APRIL it is used to "navigate" through UML class models, gathering (sets-of) objects along association graphs, which then can be utilized to formulate constraints. Hence, this is one of the most important constructs in APRIL and deserves special mention. The semantics is equivalent to OCL's [5] *collect*-operation.

mathematical	APRIL	OCL
$a \in A$ $B \in A$	*a is in A* *B is in A*	$A \rightarrow$ *includes* (a) $A \rightarrow$ *includesAll* (B)
$A = B$	$A = B$	$A = B$
$A \cap B = \emptyset$ $a \notin A$	*A is not in B* *a is not in A*	$B \rightarrow$ *excludesAll*(A) $A \rightarrow$ *excludes*(a)
$A \bowtie' B$	$A.B$	$A.B$ $A \rightarrow$collect(B)

TABLE I. SOME COMMON CONSTRAINTS ON SETS.

More conventional and common constraints in APRIL can be found in [2].

V. CASE STUDY ON THE ACCEPTANCE OF APRIL

The goal of the case study was to discover, if the APRIL syntax is understandable to untrained users with a basic understanding of logic. For this, a representative group of thirty computer science students in their first and second year could be motivated to participate. The major part was completely inexperienced in the field of UML-modeling and has never heard of OCL before. We considered OCL version 2.0 as the benchmark language. That was because OCL 2.0 -as a part of the UML specification- is an established and well defined language that is close to our purpose: defining business rules on UML-class models. Two days before the case study, an information sheet was handed to the test persons, that explained the very basics of the APRIL and OCL syntax. This included predicate logic operators (e.g., universal and existential quantifier), operators on sets (e.g., for union, exclusion and intersection of sets) and the very important *join* operator. Moreover, necessary concepts of UML-class models were explained, necessary to comprehend the APRIL and OCL materials. This comprised the use of the most important class models concepts, e.g., classes, associations, roles and multiplicities. Directly before launching the case study session, a brief introduction into the domain of discourse was given, on which the APRIL and OCL constraints were written against. The case study sheet consisted of four sections. The first section dealt with questions on an example UML-class model and intended to show how mature the skills of the experimentees in UML modeling were

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

and also if the essentials of the related UML-model were comprehended that were necessary to understand the tasks given in the succeeding parts. The second and third section demanded to try to interpret and write down the meaning of a given sequence of 18 APRIL and 18 OCL constraints in own words. The 36 constraints were based on an example UML-class model that consisted of 5 classes. Each APRIL and OCL constraint had its semantic counterpart in the opposite language. The complexity of the constraints was increasing continually, whereas the simplest constraint was like the listing for the value constraint in Section IV-A. The APRIL constraint with the highest complexity was comparable to that in Listing I including all related Definitions from D.1 to D.3 and Listing IV as its OCL representation, respectively. In the last and shortest section, the test persons had to formulate OCL and APRIL constraints, based on business rules, given in real natural language. The conduction of the case study was organized as follows. The group of test persons was divided into two equally sized subgroups each starting either with the third part (OCL) or the second part (APRIL). This was to counterbalance potential learning effects. Remember, each constraint had its semantic counterpart in the opposite language and that is why learning effects could not be precluded. The results were as follows with respect to the average ratio of correct answers or correctly interpreted APRIL or OCL business rules:

- UML-part: 74% with an average spread of 10%.
- Simple expressions: OCL: 38%, APRIL: 69%
- Complex expressions: OCL: 36%, APRIL: 57%
- Writing expressions: OCL: 12%, APRIL: 27%

Textual feedback of the test persons:

- About 50 per cent of the test persons spent less than 20 minutes in their preparation phase. About 30 per cent were unprepared. Persons of the remaining 20 per cent invested one to two hours to prepare for the survey.
- About 90 per cent of the test persons subjectively estimated that APRIL is more understandable than OCL. Whereas, 2 students found both languages equally understandable and one student with a significant background in other formal languages found, that OCL is more understandable.

The resulting percentage of APRIL, reflecting the correctly interpreted constraints, allows to conclude that it is possible for untrained test persons to understand APRIL statements. A surprisingly high number of test persons was able to write rules. This discipline has been considered to pose a bigger problem, regarding the low preparation effort of 20 minutes for the major part of the testees. Students who invested more time for preparation gained better results in both interpreting and writing APRIL rules. For OCL we were not able to observe a similarly strong coherence between preparation time and improved results. The unexpectedly very good understandability of UML-class models, even without any preparation, might be a good indication that the combination of a graphical notation to represent concept models and a textual notation for constraints is suitable to specify understandable business rules.

VI. APRIL'S TARGET LANGUAGES

APRIL makes use of the logical frameworks OCL and Tempura to underpin its language constituents with a well defined semantics. Both languages are briefly introduced in the subsequent sections.

1) OCL: As part of the UML, OCL 2.3.1 is the target language for APRIL-invariants, pre- and post- conditions. For the sake of brevity, we give a rudimentary introduction to OCL because it is well known. The interested reader should consult the literature on OCL. The specification of OCL 2.3.1 can be found on [5].

OCL restricts UML-class models using predicate logic and operations on sets. Arithmetic-, Boolean- and relational operators are used in the conventional way. Existential and universal quantifiers allow to quantify on propositions holding on an object population derived from a class model. In order to give an idea of the OCL syntax, we provide in Listing IV a translation into OCL of the example mentioned earlier in Listing I and the definitions from (D.1)-(D.3). Here, we used OCL's decomposition mechanisms to cater to an improved readability.

```
context Customer inv rule1:
Customer::
All_customers_who_buy_products_must_pay_price_for_shipment(
  Customer::premium_customers(),
  Product::special_offers(),
  0)

context Customer def:
All_customers_who_buy_products_must_pay_price_for_shipment(
    customers : Collection(Customer),
    products : Collection(Product),
    price : Real) : Boolean =
      customers→forAll(customer |
        products→select(product |
          product.order.customer = product)→forAll(orderedProduct |
            orderedProduct.shipment.price = price))

context Customer def:
premium_customers() : Collection(Customer) =
  self.AverageAnnualTurnover > 20,000 EURO)

context Product def:
special_offer(): Collection(Product) =
    self.IsSpecialOffer = true
```

Listing IV. POSSIBLE OCL-TRANSLATION OF LISTING I

2) Tempura: Tempura is an executable subset of Interval Temporal Logic (ITL) [6]. ITL enhances predicate calculus with a notation of discrete time, expressed by separated states, and associated operators. A key feature of ITL and Tempura is that the states of a predicate are grouped together as nonempty sequences of states called intervals σ_{plus}. For example the shortest interval of states σ on a predicate can be represented by $< s >$ where s is a state. Please note that here the length σ := $|\sigma|$ = 0, which is generally the number of states in σ minus 1. The semantics of ITL keeps the interpretations of function and predicate symbols independent of intervals. Thus, well known operators like {+, -, *, and, or, not,...} are interpreted in the usual way. The characteristic operator for ITL is the operator *chop (;)*, which says that a prefix subinterval is *followed by* a suffix subinterval. Both subintervals share one state "between" them. Conventional temporal logic operators such as *next* (○) and *always* (□) examine an interval's suffix

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

subintervals whereas chop splits the interval into two parts and tests both. Furthermore, Moszkowski [6] shows how to derive operators such as always and sometimes from chop. In ITL, the formula $w := w_1; w_2$ is true if $\Im_{\langle \sigma_0..\sigma_i \rangle} [\![w_1]\!]$ and $\Im_{\langle \sigma_i..\sigma_{|\sigma|} \rangle} [\![w_2]\!]$ are true in the respective sub-formulas. Note that w_1 and w_2 share the same subinterval σ_i. We adopt some examples from [6], which are as follows:

σ	P	R
s	1	2
t	2	1
u	3	1

The lenght of interval σ is expressed by $|\sigma|$ and is defined as the number of the states in σ minus one. Thus, in our example, $|\sigma| = 2$.

The following formulas on the predicates P and R are true on the interval $< stu >$:

- $P = 1$. The initial value of P is 1.
- $\bigcirc(P) = 2$ and $\bigcirc(\bigcirc(P)) = 3$. The next value of P is 2 and the next next value of P is 3.
- $P = 1$ and P gets $P + 1$. The initial value of P is 1 and P gets increased by 1 in each subsequent state.
- $R = 2$ and $\bigcirc(\Box(R)) = 1$ The initial value of R is 2 and R is always 1 beginning from the next state.
- $P \leftarrow 1$; $P \leftarrow P + 1$; $P \leftarrow P + 1$. The formula $e_2 \leftarrow e_1$ is true on an interval if $\sigma_0(e_1)$ equals $\sigma_{|\sigma|}(e_2)$. Thus, $\leftarrow$ is called temporal assignment.

We adopt Tempura because it is able to model operations lasting over multiple state transitions, which would not be possible with a single pair of OCL pre- and post-conditions. Moreover, the reader will recognize similarities with the rationale of the test-definitions given in Section VII-A.

VII. GENERATING TEST CODE FROM APRIL STATEMENTS

This section clarifies the connection between APRIL and its target languages utilizing the *moveTo*-operator example introduced earlier. Section VII-A describes the basic rationale that influence the test framework presented in Section VII-B. The test framework is applied to an application, which helps to track movements of goods in a logistics centre. For testing the correct routing, we use the example operator *moveTo* described in Section III-C.

A. Testing

For generating proper test-code based on APRIL statements, the classification of different test types into black- and white-box testing has to be clarified. Our definition of the test types is as follows: Each function f_i in the set of functions F ::= $\{f_0, \ldots, f_n\}$ of a component under test (CUT) triggers a state transition and obeys a predefined signature. This signature requires a tuple of input values (f_{IN}) and yields a tuple of output values (f_{OUT}). A signature of a function is an interface describing a contract [22] with IN- and OUT-data, which is specified in UML-class models. We assume that a composite function g_{ik} is a conglomerate of some functions f_i to f_k, for some natural numbers $0 <= i < k <= n$. Then, any OUT-signature of a proceeding function f_j must correspond to the IN-signature of the succeeding function f_{j+1}, for some natural numbers $k < j <= i$. This convention of the inner structure can be formalized by $OUT(f_j) == IN(f_{j+1})$, which we want to abbreviate with D_j. It represents an element of a function sequence. Moreover, the following holds $IN(g_{ik}) == IN(f_i)$ and $OUT(g_{ik}) == OUT(f_k)$.

A white-box test necessitates the knowledge of the entire sequence of $D_{D_0,\ldots,D_n}$ as the internal structure of g (g_{ik}), which is normally the case as the user knows the source code. If $D(g)$ is unknown, tests are limited to reason on the data given by $IN(g)$ and $OUT(g)$, they are called black-box tests. In APRIL, black-box tests are issued to the invariants, pre- and post-conditions.

For the specification of behavioral models, we extend our recent definition of white-box tests beyond reasoning on D. We use Interval Temporal Logic (ITL) [6] for modeling behavior in white-box-tests. Therefore, we introduce behavioral constraints in APRIL, which we regard as orthogonal to the invariants as well as pre- and post-conditions. Assume D represents a state σ_1 that maps a set of values to their corresponding variables at one certain point in time. Then let σ be an ordered set of states σ_0 to σ_n, each of which describes a different D at different subsequent, discrete points in time. In our understanding, the knowledge of σ is sufficient for applying white-box-tests, which we want to utilize in our framework.

B. Test Framework and Case Study

In this section, we build a representative example around the behavioral all-elements-move-to-operator introduced in Section III-C. The definitions of the previous section are used in our test framework, which deals with logistic processes to handle the material flow in a warehouse. It consists of a simple 3-tier architecture with RFID-readers and light sensors at the field-level and an ERP-system at the top level. Between these two levels, we use an RFID-middleware -Rifidi [23]- for information exchange and filtering.

The connection between a specification in Tempura and a function in the productive code is the test data. Therefore, the user has to provide initial test data $IN(f_0)$, constituting an important part of a test-case. The productive code affects the data $OUT(f_i)$ in the memory for each invocation of f_i, which marks a new interval at the same time. Thus, each time a function under test f_i gets invoked a snapshot of the input data (f_{IN}) prior to the invocation and output data (f_{OUT}) when f_i is left gets generated. The test data for the Tempura-statements is provided by recorded history-data that is stored in a properly formatted log-file containing a condensed version of the data-snapshots. The retrieval of the test data from the running system is achieved via AspectJ [24]. Therefore, AspectJ pointcut statements are generated based on the reference-nodes (see Listing III) to class-attributes found in the AST of an APRIL statement. The use of AspectJ permits us to leave the original code of the productive system untouched.

The use case for the earlier mentioned example with the behavioral operator *moveTo* formalized in Listing III is as follows: Imagine a warehouse that has a high-bay storage

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

and a loading bay for lorries. Both, storage and lorry-bay are connected with a conveyor belt. Each of the three components is equipped with one RFID-reader that can detect tagged-goods in its near field to allow tracking whether the correct thing takes the right path in the right direction. For a customer order, all goods in store contained in the order must go from the store to the lorry-bay via the conveyor belt. For simplicity we assume that each good will be detected by exactly one of the three RFID-readers at a time. This simplification is an abstraction of the real world, which does not influence considerations regarding the presented methodology.

OUTPUT	σ_I	STORE	GATE1	BAY
	I=1	"a","b"		
	I=2	"b"	"a"	
	I=3	"b"		"a"
	I=4		"b"	"a"
	I=5			"a","b"

TABLE II. REPRESENTATION OF LOG-FILE RECORDED FOR EXAMPLE-OPERATOR

The described scenario can be reflected by a log file as depicted in Table II, if the actual memories of the readers holding the IDs of the tags can be accessed in the productive application via the following reference-IDs: STORE for the RFID-reader observing the near-field of the storage, GATE1 for the conveyor and BAY for the lorry-bay. The data in the log file is formatted as array with the symbolic name OUTPUT.

```
define store_moves_to_Bay_over_Gate1 () = {
  len(|OUTPUT|-1) and
  I = 0 and
  I gets I+1 and
  moveAtoB(OUTPUT[I][Store], OUTPUT[I][Gate1]) and
  moveAtoB(OUTPUT[I][Gate1], OUTPUT[I][Bay]) and
  OUTPUT[|OUTPUT|-1] [Bay] ← OUTPUT[0] [Store]
}.

define moveAtoB (A,B) = {
  if (|A| > 0) then {
    first(A) gets last(B) and skip
  }
}.
```

Listing V. TEMPLATE FOR THE ALL-ELEMENTS-MOVE-TO OPERATOR.

With regard to the model, the Tempura statements in Listing V hold. They are actually an instantiation of a template that is used by the APRIL-compiler for translating the move-to-operator if used in an APRIL statement like in Listing VI.

```
all elements in Store move to Bay over Gate1.
```

Listing VI. USAGE OF THE ALL-ELEMENTS-MOVE-TO OPERATOR.

The formatting of the statements is according to String-Template described by Parr [25] and contains generic parts that get filled according to the parameters of the operator in Listing VI.

VIII. RELATED WORK

SBVR-Structured-English (SE) and similarly RuleSpeak [26] are so-called controlled languages to express business rules in a restricted version of natural language. Both are based on SBVR, which defines semantic parts, e.g., terms and facts to determine business concepts and their relations. The syntactic representation of these parts is achieved by text formatting and coloring, which could be used to aid parsing SE-statements. From our viewpoint, mixing technical information with the textual representation is problematic because formalized and natural language semantics have to be maintained in one and the same statement. However, natural language does not utilize text formatting information for transporting semantics.

Nevertheless, SE is used for model representation, which Kleiner et al. [27] utilize as a starting point for translating schema descriptions (in SE) into UML-class models, which is helpful for software development. Unfortunately, they leave the treatment of business rules for further work. Regarding the customizability aspect of business statements, the approach of Sosunovas et al. [28] presents another way, utilizing regular patterns. They pursue a three-step approach to constructing business rule templates that are first defined on an abstract level and then tailored to fit a specific domain with every further refinement step. Therewith, they provide precise meta-model-based semantics to the template elements but -as they admit- not to the business rule resulting from using the template.

In the field of semantic web, several controlled natural language (CNL) approaches have been elaborated. Hart et al. [29] propose a CNL called Rabbit to specify ontologies. The language provides means to specify concepts and relations in a dictionary like manner. Axioms describe the kind of relations between concepts and also allow to specify cardinalities on relations and constraints based on propositional logic. Moreover, Rabbit allows to reference other ontologies to make use of already existing concepts and axioms.

A pragmatic approach to define natural language constructs in CNL is presented by Spreeuwenberg et al. [30] and van Grondelle et al. [31]. They use patterns with a regular syntax consisting of constants and placeholders that can be replaced by instances of meta model concepts. Each pattern is related to a graph in the meta model to represent its semantics exclusively based on that meta model. However, from our viewpoint the interesting thing is that they emphasis the particular simplicity of the construction of patterns even for untrained persons. That is also what we found out with our APRIL definitions.

Another interesting approach in generating tests from requirements specifications is introduced by Nebut et al. [32]. They utilize UML use-case models combined with contracts represented by pre- and post- conditions to specify sequences of state transitions. Based on these contracts, they simulate the modeled behavior by intentionally "instantiating" the use case model. This approach could be a worthy extension to ours, which uses historical data that could also be generated by simulation. Moreover, Nebut et al. show how to generate test-cases from sequence diagrams and test objectives, that cater to a defined test coverage.

IX. CONCLUSION AND FUTURE WORK

With APRIL we want to provide a customizable and semantically well-founded notation that is close to natural language and suitable for humans as well as for computers. A core feature of APRIL is the ability to define abstract mix-fix operators that are particularly useful to define natural language expressions as reusable patterns. We consider this pattern building technique as sufficiently intuitive even for untrained

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

persons, which we could show in a case study with 30 test persons. The semantic underpinning of the mix-fix operators is achieved by customizable atomic formulas. Te ease the use of APRIL, we have encorporated additional atomic formulas that are based on frequently used constraints in practice, so called common constraints. The syntax of atomic formulas can be tailor-made for any domain. This is exemplified by a new atomic formula taken from the logistics domain to model behavior. We extend APRIL's grammar and present a mapping to the interpretation function based on Interval Temporal Logic. With the use of the new atomic formula and the transformation into the instantiated Tempura statement, executable test code can be generated. This way, our framework contributes to an integrated software development process by providing unambiguous and understandable business rules that can be used for specification purposes and for automatically generating tests.

From the current viewpoint, some issues are still open. Further evaluation is needed to determine wether the specification of the grammar rules and their corresponding rewrite rules are suitable to a typical requirements engineer. The use of OCL and especially Tempura, for creating the templates requires a considerable amount of skills. Moreover, using APRIL requirements requires a basic understanding of logic and set-theory. It has to be discovered if the aforementioned challenges are manageable by the typical requirements engineer in reasonable amout of training-time. Hence, future work will target on refining the presented approach with a focus on methodologies to improve APRIL's usability.

ACKNOWLEDGEMENTS

The authors are greatful for many hours of inspiring discussion and feedback received from Hans-Michael Windisch.

REFERENCES

[1] Christian Bacherler, B. Moszkowski, C. Facchi, and A. Huebner, "Automated Test Code Generation Based on Formalized Natural Language Business Rules," in *ICSEA 2012, The Seventh International Conference on Software Engineering Advances: IARIA Conference.*, 2012, pp. 165–171.

[2] C. Bacherler, C. Facchi, and H.-M. Windisch. (2010) Enhancing Domain Modeling with Easy to Understand Business Rules. HAW-Ingolstadt. [retrieved: 09,2012]. [Online]. Available: http://www.haw-ingolstadt.de/fileadmin/daten/allgemein/dokumente/Working_Paper/ABWP_19.pdf

[3] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[4] P. Liggesmeyer, *Software-Qualität*. Spektrum, Akad. Verl, 2002.

[5] Object Management Group. (2010) OCL Specification: version 2.3.1. [retrieved: 09,2012]. [Online]. Available: http://www.omg.org/spec/OCL/2.3.1/PDF/

[6] B. Moszkowski, *Executing Temporal Logic Programs*. Cambridge, 1986.

[7] A. van Lamsweerde, *Requirements engineering: from system goals to UML models to software specifications*. Chichester: Wiley, 2009.

[8] J. Cabot, R. Pau, and R. Raventós, "From UML/OCL to SBVR specifications: A challenging transformation," *Information Systems*, vol. 35, no. 4, pp. 417–440, 2010.

[9] Object Management Group. (2010) UML Specification: version 2.2. [retrieved: 09,2012]. [Online]. Available: www.omg.com/uml

[10] T. A. Halpin, "Verbalizing Business Rules : Part 1-16," *Business Rules Journal*, 2006.

[11] C. Rupp, *Requirements-Engineering und -Management: Professionelle, iterative Anforderungsanalyse für die Praxis*, 5th ed. München and Wien: Hanser, 2009.

[12] N. Danielsson and U. Norell, "Parsing mixfix operators," *Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages (IFL 2008)*, 2009.

[13] T. Parr. (2012) ANTLR v3. [retrieved: 09,2012]. [Online]. Available: http://www.antlr.org/

[14] ——, *The Definitive ANTLR Reference*. Pragmatic Bookshelf, 2007.

[15] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, 2007.

[16] D. Costal, C. Gómez, A. Queralt, R. Raventós, and E. Teniente, "Facilitating the Definition of General Constraints in UML," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds. Springer Berlin / Heidelberg, 2006, vol. 4199, pp. 260–274.

[17] E. Miliauskait and L. Nemurait, "Representation of integrity constraints in conceptual models," *Information technology and control, Kauno technologijos universitetas, ISSN*, pp. 34–4, 2005.

[18] E. Miliauskaite and L. Nemuraite, "Taxonomy of integrity constraints in conceptual models," *Proceedings of IADIS Database Systems*, 2005.

[19] S. Navathe and e. Ramez, *Fundamentals of Database Systems*. Addison-Wesley, 2002.

[20] T. Halpin, A. Morgan, and T. Morgan, *Information modeling and relational databases*. Morgan Kaufmann, 2008.

[21] S. Mellor and M. Balcer, *Executable UML: A foundation for model-driven architectures*. Addison-Wesley Longman Publishing Co., Inc, 2002.

[22] B. Meyer, "Applying Design by Contract," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[23] Rifidi Community. (2012) Rifidi-Platform. [retrieved: 09,2012]. [Online]. Available: http://www.transcends.co/community

[24] Eclipse Open Plattform Community. (2012) AspectJ: Version 1.7.0. [retrieved: 09,2012]. [Online]. Available: http://www.eclipse.org/aspectj/

[25] T. Parr. (2012) String Template: Version 4.0. [retrieved: 09,2012]. [Online]. Available: http://www.stringtemplate.org/

[26] Object Management Group. (2008) SBVR Specification: version 1.0. [retrieved: 09,2012]. [Online]. Available: http://www.omg.org/spec/SBVR/1.0/

[27] M. Kleiner, P. Albert, and J. Bézivin, "Parsing SBVR-Based Controlled Languages," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, A. Schürr and B. Selic, Eds. Springer Berlin / Heidelberg, 2009, vol. 5795, pp. 122–136.

[28] S. Sosunovas and O. Vasilecas, "Precise notation for business rules templates," *Databases and Information Systems, 2006 7th International Baltic Conference on*, pp. 55–60, 2006.

[29] G. Hart, M. Johnson, and C. Dolbear, "Rabbit: Developing a Control Natural Language for Authoring Ontologies," in *The Semantic Web: Research and Applications*, ser. Lecture Notes in Computer Science, S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis, Eds. Springer Berlin Heidelberg, 2008, vol. 5021, pp. 348–360.

[30] S. Spreeuwenberg, J. van Grondelle, R. Heller, and G. Grijzen, "Using CNL techniques and pattern sentences to involve domain experts in modeling," *Controlled Natural Language*, pp. 175–193, 2012.

[31] J. van Grondelle, R. Heller, E. van Haandel, and T. Verburg, "Involving business users in formal modeling using natural language pattern sentences," *Knowledge Engineering and Management by the Masses*, pp. 31–43, 2010.

[32] C. Nebut, F. Fleurey, Y. Le Traon, and J. Jézéquel, "Automatic test generation: A use case driven approach," *Software Engineering, IEEE Transactions on*, vol. 32, no. 3, pp. 140–155, 2006.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org