Automated Test Code Generation Based on Formalized Natural Language Business Rules

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

Christian Facchi, Andreas Huebner
Institute of Applied Research
Ingolstadt University of Applied Sciences
Ingolstadt, Germany
{christian.facchi|andreas.huebner}@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. For more domain specific expressiveness, the language framework permits customizing basic language statements, so called atomic formulas. Each atomic formula has a precise semantics by means of predicate and 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 example is utilized to illustrate the verification of requirements by automated test generation based on our formalized natural language business rules. 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; 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 requirements specifications in a way that can be understood by both sides. Nonetheless, unstructured natural language in requirements documents does not ensure identical interpretations by different readers, which has always been a fundamental problem in software engineering [1]. Moreover, machine-readability of a requirements document can be a big asset but requires a formal syntax that is not provided by unstructured natural language [2].

By the introduction of *AtomsPro Rule Integration Language* (APRIL) [3], 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 extend APRIL's expressiveness using atomic formulas that constitute the link between statements that are like natural language and formal frameworks.

Formal specifications enhance the established software development process (V-Model). 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 [4] [5].

The paper is structured as follows: The next Section II will give 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 Sec. IV the transformation of example statements in our language into computer processable test code. After the discussion of related work (Section V), a conclusion will be drawn and future work will be presented (Section VI)

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.

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.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-230-1

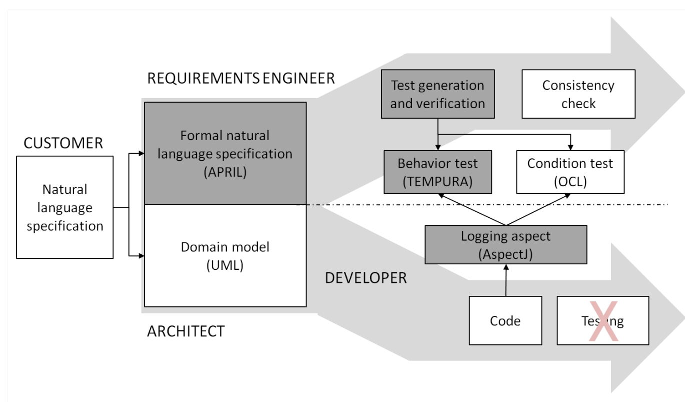

Fig. 1. Overview of the software development process using APRIL.

The treatment of complex real-world business rules using mixfix 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. Atomic formulas provide a precise semantics for APRIL Definitions.

Tests based on APRIL statements can be generated to check conditions using invariants, pre- and post-conditions in the Object Constraint Language (OCL) [6] notation. Checking process behavior is done by the use of a subset of Interval Temporal Logic (ITL) called Tempura. The rationale for applying our testing-framework is laid down in Section IV-A. Section IV-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 III-D. 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. Some translation examples are shown alongside the introduction of the APRIL language.

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

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 [2]. 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. APRIL supports the specification of business rules that are formal enough to be processed by computers, but still close enough to natural language to ensure readability and comprehensibility for humans.

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 [7]. Despite the fact that there are fundamental intentional differences, these rule types have one aspect in common: The projection of the semantics of parts of the real world into formal representations by means of logic. In APRIL we use UML-class models [8] 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 [2], which is not possible with a single pair of pre- and post-condition.

In Figure 2, a simple domain model of a car-rental system, with the basic concepts Car, Rental and Customer, is shown as UML-class model. As an example of APRIL usage on the class-model, the corresponding statement for the invariant underageCustomers can be seen in Listing I.

1	***Invariant*** *underageCustomers* ***concerns*** *Rental:*
2	*All underage customers who rent a Porsche must pay*
3	*plus 150 percent.*

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

The header (line 1) of a rule contains its name *(underageCustomers)* and the token after the keyword **concerns**, which represents the context set (represented by the class name *Rental*) 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.

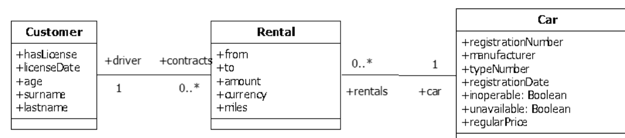

Fig. 2. UML-model of the car rental example.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-230-1

Moreover, a detailed specification of APRIL including default logic- and set- operators, is given in [3].

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 [9]. 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 [1].

Despite the convenience that mix-fix operators provide to humans, it is quite challenging to implement the parser logic [10], 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 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 [11] parser-/compiler-generator framework. The framework allows to specify semantic annotations [12], which are actually user defined code snippets (e.g., in Java) that get 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. E.g., when a token with the value *Rental* 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 car rental 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)) must pay plus (ratio* ***as*** *Number) per cent* ***yielding*** *Boolean* ***is defined as every*** *customer* ***satisfies that*** *contracts.amount = contracts.car.regularPrise * (1 + ratio).*

(D.2) ***Definition*** *underage customers who rent (type* ***as*** *Integer)* ***yielding*** *Collection(Customer)* ***is defined as each*** *customer* ***in all instances of*** *Customer* ***where*** *customer.age* $<$ *21* ***and*** *customer.contracts.car.typeNumber=type.*

(D.3) ***Definition*** *a Porsche* ***yielding*** *Integer* ***is defined as*** *911.*

In (D.1), the exact offset ratio is mapped to a set of customers. On the other hand, (D.2) is a set-comprehension on the set of all customers defining, what an underage customer is that rents a certain car type. Furthermore, (D.3) maps an identifier-constant (911) of type integer to a name representing the intended car type.

In order to provide a precise semantics to the Definitions, APRIL **atomic formulas** are used. They are verbalized versions of operations on sets, predicate-logic formulas and special common constraints sketched by Halpin [9]. 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 described in [3]. Default atomic formulas are for maintaining sufficient expressive power and straight-forward translation into executable representations. Therefore, 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. Moreover, this approach 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 [1], 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. Therefore, a state of practice language implementation mechanism described by Parr [12] is used. First, a formal production rule of the new atomic formula must be specified. Formal production rules are parts of a context-free grammar [13] and 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 language extension mechanisms that APRIL provides. It formalizes the example operator that reflects the scenario mentioned above. In line 1 the production rule with the name of the 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. It 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 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.

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

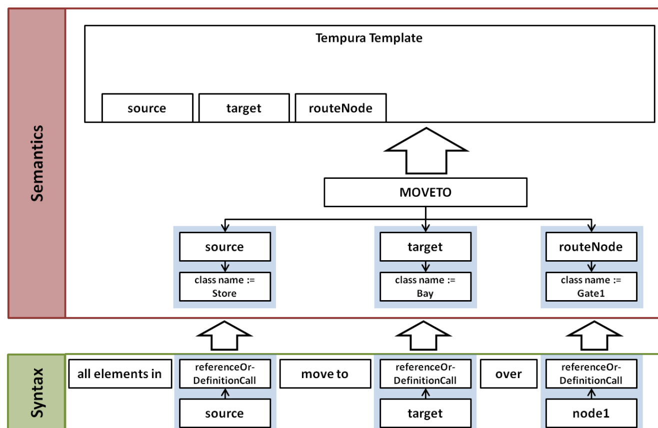

Fig. 3. Translation example of the atomic operator **moveTo**.

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.

translation of the atomic formula.

D. 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: 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 [6].

OCL restricts UML-class models using predicate logic and operations on sets. Arithmetic-, Boolean- and relational operators are used in the conventional way. The well known 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 car-rental 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 Rental inv underageCustomers:
Customer::
All_customers_must_pay_plus_ratio_per_cent(
  Customer::underage_customers_who_rent_type(
    Car::a_Porsche),150)

context Customer def:
All_customers_must_pay_plus_ratio_per_cent(
  customers:Collection(Customer), ratio:integer) :
    Boolean = customers→forAll(customer|
      customer.contracts.amount =
      customer.contracts.car.regularPrise *
      (1+ratio))

context Customer def:
underage_customers_who_rent_type(
  type:integer) : Collection(Customer) =
  allInstances()→select(customer: Customer |
    customer.age < 21 and customer
    contracts.car.typenumber = type)

context Car def: attr a_Prosche : integer = 911
```

Listing IV
OCL-TRANSLATION OF THE INTRODUCTORY CAR-RENTAL-EXAMPLE

2) Tempura: Tempura is an executable subset of Interval Temporal Logic (ITL) [14]. Like some other temporal logics, ITL enhances predicate calculus with with a notation of discrete time 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}. They are called intervals. For example the shortest interval (of states) σ on a predicate P can be represented by $< s >$ with 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 (;)*. Conventional temporal logic operators such as *next* ($\bigcirc$) and *always* ($\Box$) examine an interval's suffix subintervals whereas chop splits the interval into two parts and tests both. Furthermore, Moszkowski [14] 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 [14], 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 IV-A.

IV. 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 IV-A describes the basic rationale that influence the test framework presented in Section IV-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 ::= $\{f_0$ to $f_n\}$ 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 [15] 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 $D_{D_0,...,D_n}$ 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) [14] 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 [16]- for information exchange and filtering.

Copyright (c) IARIA, 2012.

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 [17]. Therefore, AspectJ point-cut 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 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.

The described scenario can be reflected by a log file as depicted in Table I, 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.

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

TABLE I
REPRESENTATION OF LOG-FILE RECORDED FOR EXAMPLE-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. The formatting of the statements is according to String-Template described by Parr [18] and contains generic parts that get filled according to the parameters of the operator in Listing VI.

```
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.

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

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

V. RELATED WORK

SBVR-Structured-English (SE) and similarly RuleSpeak [19] 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. [20] 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. [21] 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.

Another interesting approach in generating tests from requirements specifications is introduced by Nebut et al. [22]. 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.

VI. 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 design principle 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 persons. The semantic underpinning of the mix-fix operators is achieved by customizable atomic formulas. 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 is generated. This way our framework contributes to an integrated software development process by providing unambiguous and understandable business rules that can be reused 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. Also, the use of OCL and especially Tempura, for creating the templates requires a considerable amount of skills. Moreover, using APRIL 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. 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] C. Rupp, *Requirements-Engineering und -Management: Professionelle, iterative Anforderungsanalyse für die Praxis*, 5th ed. München and Wien: Hanser, 2009.

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

[3] 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

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

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

[6] 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/

[7] 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.

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

[9] T. A. Halpin, "Verbalizing Business Rules: Part 14," *Business Rules Journal*, vol. 7, no. 4, 2006.

[10] 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.

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

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

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

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

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

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

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

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

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

[20] 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.

[21] 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.

[22] 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.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-230-1