KoDEgen: A Knowledge Driven Engineering Code Generating Tool

Reuven Yagel, Anton Litovka and Iaakov Exman

Software Engineering Department, The Jerusalem College of Engineering – Azrieli,
POB 3566, Jerusalem, 91035, Israel

Abstract. KDE – Knowledge Driven Engineering – is an extension of MDE to a higher level of abstraction, in which ontologies and their states replace the standard UML models. But in order to test in practice the KDE approach one needs to actually *run* the highly abstract ontologies and resulting models and code. This work describes the design and implementation of KoDEgen – a KDE tool for code generation – based on ontologies, ontology states and a scenario file. The implementation uses a modified Gherkin syntax. The tool is demonstrated in practice by generating the actual code for a few case-studies.

1 Introduction

Discovery of bugs in early development stages reduces costs, is a widely accepted tenet [5], even though agile approaches challenge its exact formulation. Within KDE – Knowledge Driven Engineering – early implies higher levels of abstraction.

Exman et al. [9] have recently proposed Runnable Knowledge – bare concepts and their states – as the highest system abstraction level. Exman and Yagel [10] made a further step by proposing ROM their Runnable Ontology Model testing approach, actually starting from ontologies and ontology states.

This paper embodies the ROM proposal in the KoDEgen tool. One assumes for a certain domain the a priori given relevant ontology and its states. KoDEgen generates, from the ontology and its states, classes of the system under development (SUD), while submitting them to tests to be applied according to given specifications.

KoDEgen is being gradually built to automatically generate the running code from the abstract model and its tests. This paper describes a mostly automatic version, at times with human intervention. The interactions refine the SUD and KoDEgen itself.

1.1 Related Work: From Executable Specifications to Code Generation

A very condensed review of the literature is presented here. The Agile software movement has stressed in recent years early testing methods, e.g. Freeman and Pryce [11]. Its main purposes are faster understanding of the software under development obtained by short feedback loops, and guiding the software system development in rapidly changing environments.

Early testing methods stemmed from Test Driven Development (TDD), the unit-

Yagel R., Litovka A. and Exman I..
KoDEgen: A Knowledge Driven Engineering Code Generating Tool.
DOI: 10.5220/0004640400240033
In *Proceedings of the 4th International Workshop on Software Knowledge* (SKY-2013), pages 24-33
ISBN: 978-989-8565-76-1

testing practice by Beck [4]. In such methods, scripts demonstrate the various system behaviors, instead of just specifying the interface and a few additional modules. Since the referred scripts' execution can be automated, the referred methods are also known as automated functional testing.

Among TDD extensions one finds Acceptance Test Driven Development (ATDD) also known as Agile Acceptance Testing, see e.g. Adzic [2]. Another such extension is Behavior Driven Development (BDD) North [14], emphasizing readability and understanding by stakeholders. Recent representatives are Story Testing, Specification with examples Adzic [3] or Living/Executable Documentation, e.g. Brown [6], and Smart [19].

There exist common tools to implement TDD practices. FitNesse by Martin [1] is a wiki-based web tool for non-developers to write formatted acceptance tests, e.g. tabular example/test data. The Cucumber (Wynne and Hellesoy [21]) and SpecFlow [20] tools directly support BDD. They accept stories in plain natural language (English and a few dozen others). They are easily integrated with unit testing and user/web automation tools. Yagel [22] reviews extensively these practices and tools.

An introductory overview of ontologies in the software context is found in Calero et al. [7]. Ontology-driven software development papers are found in Pan et al. [16]. The combination of ontology technologies with Model Driven Engineering is discussed at length in Parreiras [16].

In the remaining of the paper we introduce the Ontology abstraction level (section 2), describe testing with the Gherkin syntax of the Cucumber tool (section 3), study automatic code generation implemented in the KoDEgen tool (section 4), describe two case studies (section 5) and conclude with a discussion (section 6).

2 The Ontology Abstraction Level

The highest Runnable Knowledge abstraction level (Exman et al. [9]) is an abstraction level above standard UML models. Since UML models separate modeling structure and behavior into different diagrams – typically class diagrams and statecharts – the highest abstraction level also is designed to separate structure from behavior.

Ontologies – mathematical graphs with concepts as vertices and relationships as edges – represent the static semantics of software systems. From ontologies one can, by means of appropriate tools, to naturally generate structures, viz. classes.

Ontology states – mathematical graphs with concepts' states as vertices and labeled transitions as edges – are our representation of the dynamic semantics of software systems. From ontology states one can, by means of appropriate tools, to naturally generate behaviors, viz. statecharts. Ontology states are a higher abstraction of statecharts, abstracting detailed attributes, functions and parameters. Ontology states are not the only alternative to represent dynamic semantics (see e.g. Pan et. al. [16]).

For illustration, Fig. 1 displays a graphical representation of a version of an ATM (Automatic Teller Machine) ontology. An ATM appears later as one of the case studies – in section 5.

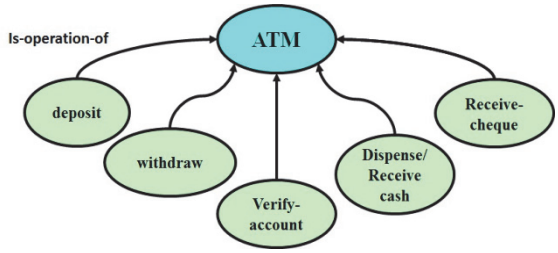


Fig. 1. An ontology for an ATM – Five concepts standing for five possible ATM operations are displayed, besides the ATM concept itself.

3 Testing with Modified Gherkin Syntax

To test the ontology and ontology states, we use a modified Gherkin Syntax specification as in Fig. 2. This file is usually developed by the system's stakeholders.

```
Feature:  Account Withdrawal

Scenario: Successful withdrawal from an account
   Given an account has a balance of <amount>$100
   When <amount>$20 are withdrawn from an ATM
   Then the account <balance>balance should be $80
```

Fig. 2. ATM withdrawal operation specification – It specifies successful cash withdrawal from an ATM. It is expressed in the modified Gherkin style. Tags added by the developer - marked in bold red within angular brackets - to facilitate test script generation (see section 5).

The keywords shown here in blue are:

a) *Feature* – provides a general title to the specification;
b) *Scenario* – provides a title for a specific walk through;
c) *Given* – pre-conditions before some action is taken;
d) *When* – an action that triggers the scenario;
e) *Then* – the expected outcome.

For further details see [21] and our previous work [10].

Running this specification alone fails as it lacks code supporting. A domain model is needed. A tool like Cucumber can suggest steps to satisfy the given specification. Mock objects could also stand for the missing concepts. Cucumber's mode of usage is iteration and refinement until the specification is complete. This is checked by test scripts. These may catch software regressions caused by new system features.

KoDEgen goes a step further and fills the generated steps with actual code that exercises the interactions between the ontology classes. The ontology may not be complete, or the specifications, sometimes written by non-technical persons, may contain yet more gaps. KoDEgen is designed to maximize automation with the known ontologies. Thus, KoDEgen hints to the developer to slightly modify the specification with tags to be used to generate the code.

4 KoDEgen Software Architecture: Generation of Running Code

KoDEgen has three *inputs*:

- Initial Specification – obtained by elicitation of system requirements;
- Ontology – obtained by specialization of generic domain ontologies;
- Ontology States – obtained by setting transitions between concept states.

The fusion module uses ontology concepts and their states to generate the *outputs*:

- *MUT* – code skeletons of the model under test;
- *Test Scripts* – unit tests to test the MUT.

If the tests results are negative, one modifies the specifications and/or the ontology and repeats the loop. Otherwise the system model is approved.

The Runnable Knowledge model – i.e. the ontologies and their states – is the utmost abstract level in the software layers hierarchy. It is runnable in the sense that, a suitable tool can make transitions between states. Mock objects may obtain a fast and efficient translation of Runnable Knowledge into an *actually* running model.

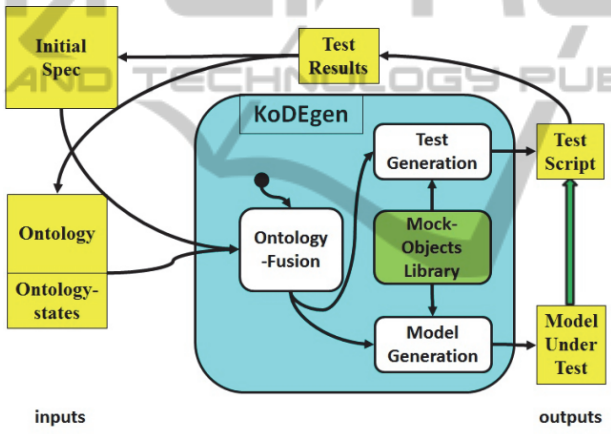


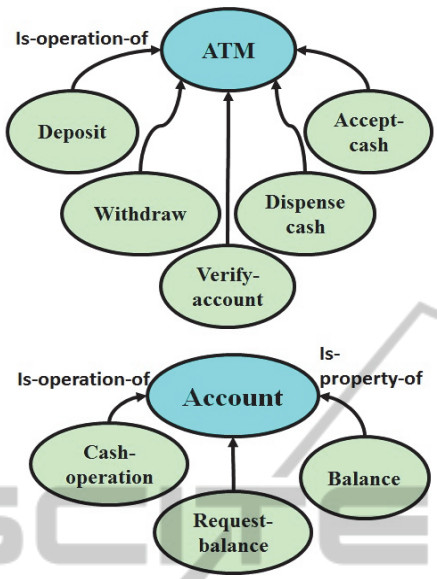
Fig. 3. KoDEgen Software Architecture – modules are round (white) rectangles, while inputs and outputs are regular (yellow) rectangles. Mock-Objects may complement generated code. The wide arrow upwards means that Test-Script is used to the test the MUT.

5 Case Studies

Here we describe two case studies from the given input, to the generated code. The first is an ATM, Automatic Teller Machine, with cash withdrawal transactions.

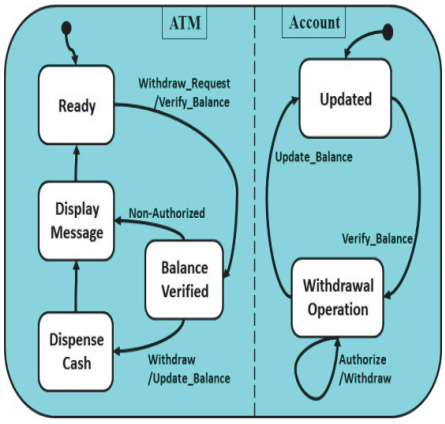
5.1 ATM

Two ontologies, *ATM* and bank *Account*, are used in the ATM example (in Fig. 4). Their ontology states are shown in Fig. 5




```
<?xml version="1.0"?>
<ontology>
    <class>
          <name>ATM</name>
          <attribute
id="0">Deposit</attribute>
          <!--attribute
id="1">Withdraw</attribute-->
          <attribute id="2">Verify account
                              </attribute>
          <attribute id="3">Dispense cash
                              </attribute>
          <attribute id="4">Accept cash
                              </attribute>
    </class>
    <class>
          <name>Account</name>
          <attribute id="0">Cash operation
                              </attribute>
          <attribute id="1">Request
                       balance</attribute>
          <attribute id="2">
                       Balance</attribute>
    </class>
</ontology>
```

Fig. 4. ATM and bank Account Ontologies – A graphical representation is in the left hand side. The concepts in the ATM ontology (upper) are operations performed by the ATM. The concepts in the Account ontology (lower) are operations (Cash-operation and Request-Balance) and a property (balance) of the Account. The XML representation is in the right hand side.




```
<?xml version="1.0"?>
<state_machine>
 <ontology_state>
          <class_name>ATM</class_name>
 <state>
          <state_name>ready</state_name>
     <edge> <edge_name>add</edge_name>
     <edge_to>wait</edge_to> </edge>
  </state>
  </ontology_state>
  <ontology_state>
          <class_name>ATM</class_name>
  <state>
        <state_name>withdrawal_operation
     </state_name>
   <edge><edge_name>withdraw</edge_name>
          <edge_to>withdrawal_operation
   </edge_to> </edge>
  </state>
  </ontology_state>
</state machine>
```

Fig. 5. Ontology States of the ATM and bank Account Ontology- In the left hand side the *ATM* and *account* parallel states display the states for a cash withdrawal operation. In the right hand side an XML representation for internal manipulation within KoDEgen.

Generated Model and Running Code Implementation. KoDEgen is fed with an XML ontology and say, the ATM specification in Fig. 2. It generates model classes and a test script. Here the classes are in the Ruby language.

```
class ATM
      attr_accessor :deposit
      attr_accessor :Verify_account
      attr_accessor :dispense_cash
      attr_accessor :Accert_cash

      def withdraw(amount)
      end
end

class Account
      attr_accessor :Cash_operation
      attr_accessor :request_balance
      attr_accessor :balance
end
```

Fig. 6. ATM: Extracted model – Ruby generated classes.

KoDEgen also generates a test script, seen in Figure 7, which realizes the specification – code snippets executed sequentially and exercise the various classes.

```
require "test/unit/assertions"
require "/usr/lib/ruby/vendor_ruby/cucumber/rspec/doubles"
World(Test::Unit::Assertions)

Given /^account has a balance of <balance> \$(\d+)$/ do |balance|
        @account = Account.new
        @account.balance = balance
end

When /^<amount> \$(\d+) are withdrawn from ATM$/ do |amount|
        atm = ATM.new
        atm.withdraw( amount )
end

Then /^the account balance should be <balance> \$(\d+)$/ do |balance|
        @account.stub(:balance).and_return(balance)
        assert balance == @account.balance.to_s
end
```

Fig. 7. ATM: Runnable test script.

5.2 Internet Purchase

Here we describe an internet purchase case-study. Its classes are the shopping cart and products (that can be put in the cart). We show its ontologies (in Fig. 8) and states (in Fig. 9), directly in the internal XML representation. Testing of these classes is shown by a transaction in which two product types are purchased.

```
<?xml version="1.0"?>
<ontology>
      <class>
            <name>shopping cart</name>
            <attribute id="0">products</attribute>
            <attribute id="1">items per product</attribute>
            <attribute id="2">tax</attribute>
            <attribute id="3">current price</attribute>
            <attribute id="4">total price</attribute>
      </class>
      <class>
            <name>product</name>
            <attribute id="0">name</attribute>
            <attribute id="1">price</attribute>
            <attribute id="2">serial number</attribute>
            <attribute id="3">part number</attribute>
      </class>
</ontology>
```

Fig. 8. XML representation of Shopping cart and Product Ontologies – The Shopping cart ontology shows objects contained by the cart (product and items-per-product) and purchase properties (total-price, current-price and tax). The Product concepts are just its properties.


```
<?xml version="1.0"?>
<state_machine>
      <ontology_state>
            <class_name>shopping cart</class_name>
            <state>
                  <state_name>wait</state_name>
                  <edge>
                        <edge_name>add</edge_name>
                        <edge_to>wait</edge_to>
                  </edge>
                  <edge>
                        <edge_name>contains</edge_name>
                        <edge_to>calculated</edge_to>
                  </edge>
            </state>
      </ontology_state>
</state_machine>
```

Fig. 9. XML representation of Shopping-cart Ontology States – The cart default is empty. A product can be added, its price or final price-&-tax calculated, ending the transaction.

A Gherkin specification file is given in Fig. 10.

```
Feature: Adding to a shopping cart
Scenario: Add items to shopping cart
   Given An empty shopping cart
   When I add 1 item of Product A ($10)
   And I add 2 items of Product B ($20
each)
   And the tax is 8%
   Then the shopping cart contains 3
items
   And the total price is 54$
```

```
Feature: Adding to a shopping cart
Scenario: Add items to shopping cart
   Given empty shopping cart
   When I add <quantity> 1 of Product
<name> "A" to shopping cart
   And I add <quantity> 2 items of
Product <name> "B" to shopping cart
   And tax is <tax> 8% percent
   Then shopping cart contains
<quantity> 3 items
```

Fig. 10. Shopping-cart – Adding items to a shopping cart. In the left hand side one sees a simple Gherkin specification. In the right hand side a tagged specification, augmented with modifier tags in bold red within angular brackets, to facilitate code generation.

Fig. 11 contains the generated model classes.

```ruby
class Shopping_cart
      attr_accessor :products
      attr_accessor :items_per_product
      attr_accessor :tax
      attr_accessor :current_price
      attr_accessor :total_price

      def add(quantity, product)
      end
      def contains
      end
end
class Product
      attr_accessor :name
      attr_accessor :price
      attr_accessor :serial_number
      attr_accessor :part_number
end
```

Fig. 11. Shopping-cart: Extracted model – Ruby generated classes.

Fig. 12 displays the Shopping cart case study test script. In contrast to the ATM case study, here mock objects are applied (we used the RSpec-Mocks library [18]). The mock expectations are met by adding calls to stub objects –in bold red in Fig. 16. The script adds products A and B to empty cart, applies tax and make assertions.

Once the mock expectations were set and the test script is ready, it only remains to run it in a test runner tool (see the screenshot in Fig. 17). This test script can later be reused and re-issued to check correctness of the actual developing implementation.

```ruby
require "test/unit/assertions"
World(Test::Unit::Assertions)

        Given /^empty shopping cart$/ do
                @shopping_cart = Shopping_cart.new
        end
        When /^I add <quantity> (\d+) of Product <name> "(.*?)"
                to shopping cart$/ do |quantity, name|
                product = Product.new
                product.name = name
                @shopping_cart.add(quantity , product)
        end
        When /^I add <quantity> (\d+) items of Product <name> "(.*?)"
                to shopping cart$/ do |quantity, name|
                product = Product.new
                product.name = name
                @shopping_cart.add(quantity , product)
        end
        When /^tax is <tax> (\d+)% percent$/ do |tax|
                @shopping_cart.tax = tax
        end
        Then /^shopping cart contains <quantity> (\d+) items$/
                do |quantity|
                @shopping_cart.stub(:contains).and_return(3)
                assert quantity == @shopping_cart.contains( )
        end
```

Fig. 12. Shopping-cart: Runnable test script.

Finally, Figure 13 is a screen shot resulting running the generated test script with Cucumber. The steps from the scenario are marked green meaning that the test tool could successfully run and all expectations were met.

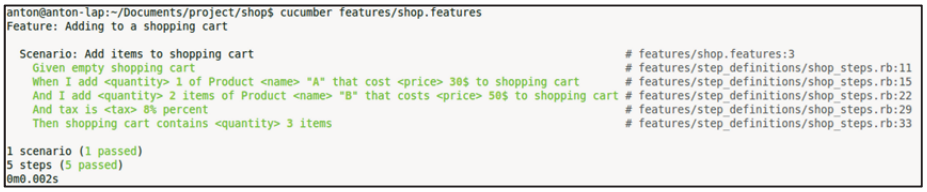


Fig. 13. Shopping-cart: Running test results – screenshot of running of the above test script with Cucumber. It is a passing test, since all expectations where met by the models, all of the steps in the test script were successfully done.

6 Discussion

The KoDEgen agile process is important for the understanding of both the systems under development and the tool itself, which evolved during this work. For instance, gaps between ontologies and execution were filled through the aid of tags. The realization of a specification into a running test script is done through KoDEgen.

Mock object libraries are not necessarily mandatory, but may be used to pass tests in order for the system developer to be able to test the integrity of the model.

KoDEgen is written in Java, and the source code with the discussed examples can be obtained here [12].

The program already embodies quite a significant knowledge as a set of rules to handle common patterns and idioms when handling the inputs. For example, during the test script generation, an object under test is recognized according to the ontology and by appearing at the 'when' part of the specification. Thereafter, the actions performed in the following steps are related implicitly or explicitly to this object under test. We continue growing this set as we use the tool for different domains and input sizes.

6.1 Future Work

Among issues still open to investigation is the extent of KodeGen automation: will it remain a useful quasi-automatic tool?

In this work the tools produce code in Ruby which is more concise than, e.g., C#/Java. One can also use specific language features to improve the produced scripts, e.g., using partial classes in C# to separate expectations from the test script.

6.2 Main Contribution

The main contribution of this work is the usage of code generation as a fast implementation means to check system design while still in the highest Ontology abstraction level.

References

1. Adzic, G.: Test Driven .NET Development with FitNesse, Neuri, London, UK, (2008).
2. Adzic, G.: Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing, Neuri, London, UK, (2009).
3. Adzic, G.: Specification by Example – How Successful Teams Deliver the Right Software, Manning, New York, USA, (2011).
4. Beck, K.: Test Driven Development: By Example, Addison-Wesley, Boston, MA, (2002).
5. Boehm, B.W.: “Software Engineering Economics”, IEEE Trans. Software Eng., (1984).
6. Brown, K.: Taking executable specs to the next level: Executable Documentation, Blog post, (see: http://keithps.wordpress.com/2011/06/26/taking-executable-specs-to-the-next-level-executable-documentation/), (2011).
7. Calero, C., Ruiz, F. and Piattini, M. (eds.): Ontologies in Software Engineering and Software Technology, Springer, Heidelberg, Germany, (2006).
8. Chelimsky, D., Astels, D., Dennis, Z., Hellesoy, A., Helmkamp, B., and North, D,: The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends, Pragmatic Programmer, New York, USA, (2010).
9. Exman, I, Llorens, J. and Fraga, A.: "Software Knowledge", pp. 9-12, in Exman, I., Llorens, J. and Fraga, A. (eds.), Proc. SKY Int. Workshop on Software Knowledge (2010).
10. Exman I. and Yagel R.: "ROM: A Runnable Ontology Model Testing Tool", The 4th International Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K) - SKY Workshop, Barcelona, Spain, (2012).
11. Freeman, S., and Pryce N.: Growing Object-Oriented Software, Guided by Tests, Addison-Wesley, Boston, MA, USA, (2009).
12. KoDEgen – the tool: https://github.com/AntonLitovka/R_O_M (2013).
13. Moq – the simplest mocking library for .NET and Silverlight: (see http://code.google.com/p/moq/), (2012).
14. North, D.: "Introducing Behaviour Driven Development", Better Software Magazine, (see http://dannorth.net/introducing-bdd/), (2006).
15. NUnit: (see http://www.nunit.org), (2012).
16. Pan, J.Z., Staab, S., Assmann, U., Ebert, J. and Zhao, Y. (eds.): Ontology-Driven Software Development, Springer Verlag, Heidelberg, Germany, (2013).
17. Parreiras, F.S.: Semantic Web and Model-Driven Engineering, John Wiley, Hoboken, NJ, and IEEE Press, USA, (2012).
18. RSpec mocks library: (see: https://github.com/rspec/rspec-mocks), (2013).
19. Smart J. F.: BDD in Action Behavior-Driven Development for the whole software lifecycle, Manning, 2014 (expected).
20. SpecFlow – Pragmatic BDD for .NET: (see http://specflow.org), (2010).
21. Wynne, M. and Hellesoy, A.: The Cucumber Book: Behaviour Driven Development for Testers and Developers, Pragmatic Programmer, New York, USA, (2012).
22. Yagel, R.: "Can Executable Specifications Close the Gap between Software Requirements and Implementation?", pp. 87-91, in Exman, I., Llorens, J. and Fraga, A. (eds.), Proc. SKY'2011 Int. Workshop on Software Engineering, SciTePress, France, (2011).