KODEGEN: A Code Generation and Testing Tool Using Runnable Knowledge

Iaakov Exman(✉), Anton Litovka, and Reuven Yagel

Software Engineering Department,
The Jerusalem College of Engineering – Azrieli,
POB 3566, 91035 Jerusalem, Israel
{iaakov, robi}@jce.ac.il, antonli@post.jce.ac.il

Abstract. KDE – *Knowledge Driven Engineering* – is a generalization of MDE – Model Driven Engineering – to a higher level of abstraction than the standard UML software models, aiming to be closer to the system designer concepts. But in order to reach an effective technology applicable in industry, one needs to implement it in a tool using *Runnable Knowledge*, i.e. which can be run and tested. This work describes KODEGEN – a KDE tool for testing while generating code – whose input consists of system ontologies, ontology states and scenario files. Incidental concepts not part of the ontologies are replaced by mock objects. The implementation uses a modified Gherkin syntax. The tool is demonstrated in practice by generating the actual code for a few case-studies.

Keywords: KDE · Runnable knowledge · Ontology · Ontology states · Model testing · Mock objects

1 Introduction

Software system development starting from the natural system concepts facilitates the system designer work and its understanding. KDE – *Knowledge Driven Engineering* – is a generalization of MDE (or MDA [6]) to a higher abstraction level than UML models, exactly to support a conceptually neat approach to system development.

From a slightly different point of view, "earlier bug discovery reduces costs", is a widely accepted wisdom [5]. Within KDE earlier means higher abstraction levels. Thus, also from this aspect, KDE offers novel development perspectives.

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 their Runnable Ontology Model, starting from ontologies and ontology states, and incorporating concrete scenario files for code generation and testing.

This paper embodies the latter 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 scenario specifications.

KODEGEN is being built to gradually develop software systems in a semi-automatic approach, at times with human intervention. The interactions refine the SUD and KODEGEN itself. The ultimate goal in our vision is to automatically generate the running code from the abstract model and its tests.

A. Fred et al. (Eds.): IC3K 2013, CCIS 454, pp. 260–275, 2015.
DOI: 10.1007/978-3-662-46549-3_17

1.1 Related Work: From Executable Specifications to Code Generation

A concise literature review 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-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. 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], see also [8]) and Spec-Flow [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 development 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 Pan et al. [16] and in Parreiras [17].

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

2 Runnable Knowledge: The Ontology Abstraction Level

Runnable Knowledge (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 – Runnable Knowledge, the highest abstraction level, is also 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 UML 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 UML 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 simplified version of an ATM (Automatic Teller Machine) ontology. An ATM appears later on in the case studies – in Sect. 5.

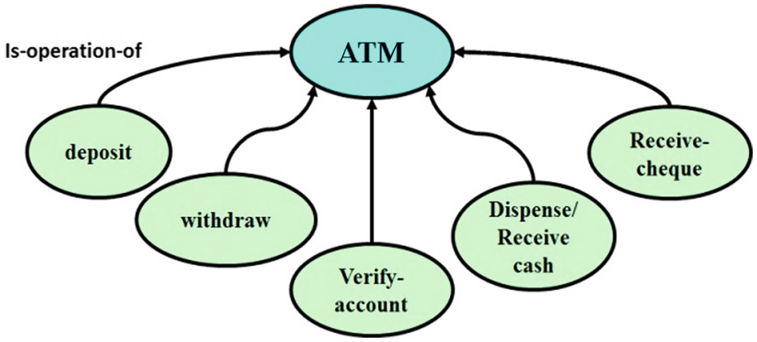


Fig. 1. An ontology for an ATM – Five concepts standing for five possible ATM operations are displayed, besides the ATM concept itself.

3 Modified Gherkin Syntax for Testing

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

The keywords shown in blue in Fig. 2 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].

```
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 are added by the developer – marked in bold red within angular brackets – to facilitate test script generation (see Sect. 5).

Running this specification alone fails as it lacks supporting code. A domain model is needed. A tool like Cucumber can suggest steps to satisfy the given specification. Mock objects could also stand for the concepts missing in the ontologies.

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 Modules: Generation of Running Code

KODEGEN, whose software modules are seen in Fig. 3, has three *inputs*:

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

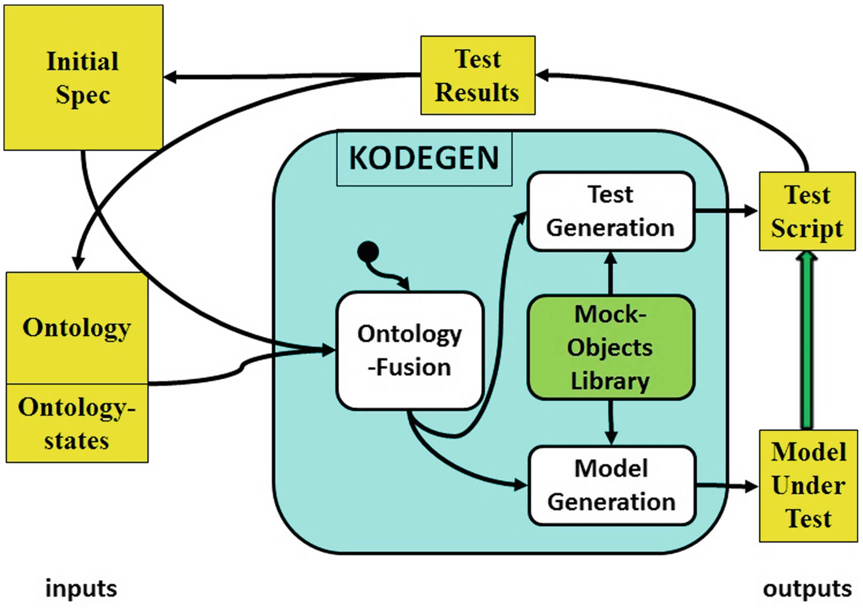


Fig. 3. KODEGEN software modules – modules are round (white) rectangles, while inputs and outputs are regular (yellow) rectangles. Mock-Objects may be needed to complement generated code. The wide arrow upwards means that test-script is used to the test the MUT (Model Under Test) (Color figure online).

The fusion module coordinates two sub-modules: a model generator and a test generator. These use 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 (using e.g. NUnit [15]).

If there are concepts in the scenario that do not appear in the ontologies, KODEGEN inserts them into the generated code by means of a Mock Objects library (see e.g. Moq [13], RSpec [18]). Mock objects are a fast and efficient addendum to Runnable Knowledge to obtain an *actually* running model.

If the tests results are negative, one should modify the specifications and/or the ontology and then repeat 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.

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. We further elaborate this example in Subsect. 5.3.

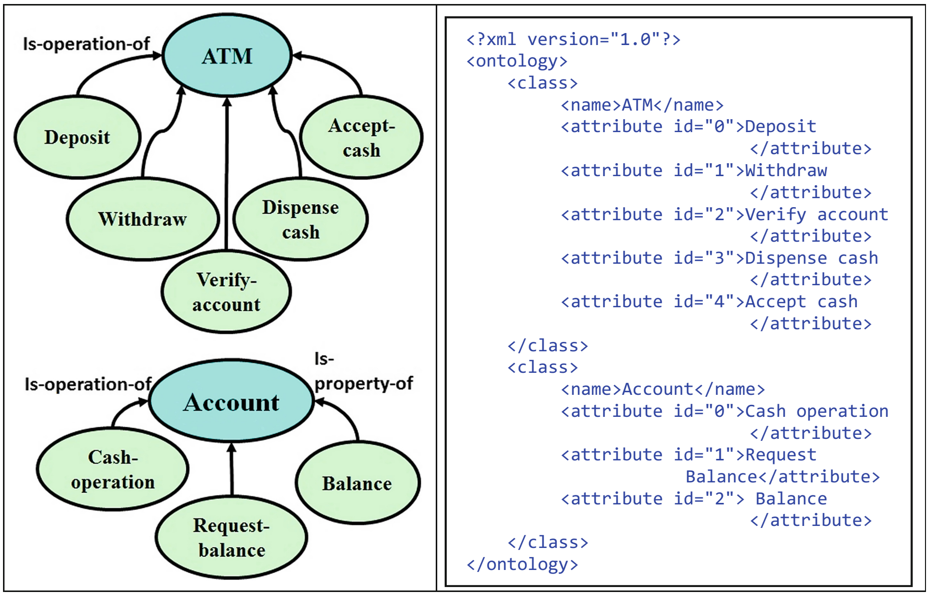
5.1 ATM

Two ontologies, *ATM* and bank *Account*, are used in the initial ATM example. In Fig. 4 these ontologies are displayed side-by-side in two formats: a- a schematic graphical format for easy reader comprehension (in the left-hand-side); b- a corresponding XML format, for internal KODEGEN usage, providing more details (in the right-hand-side). Both ontologies show their operations, such as withdraw and dispense-cash. The Account ontology also shows a property, viz. account Balance.

The respective ontology states are shown in Fig. 5. Also in this figure one discerns two formats: a- a schematic graphical format, purposefully very similar to a statechart; b- a corresponding XML format. The ATM and Account are parallel states, meaning that they are orthogonal or "independent", as they should be. In other words, an Account can certainly exist independently of its use by means of an ATM. An ATM machine is certainly built and tested independently of any specific Account.

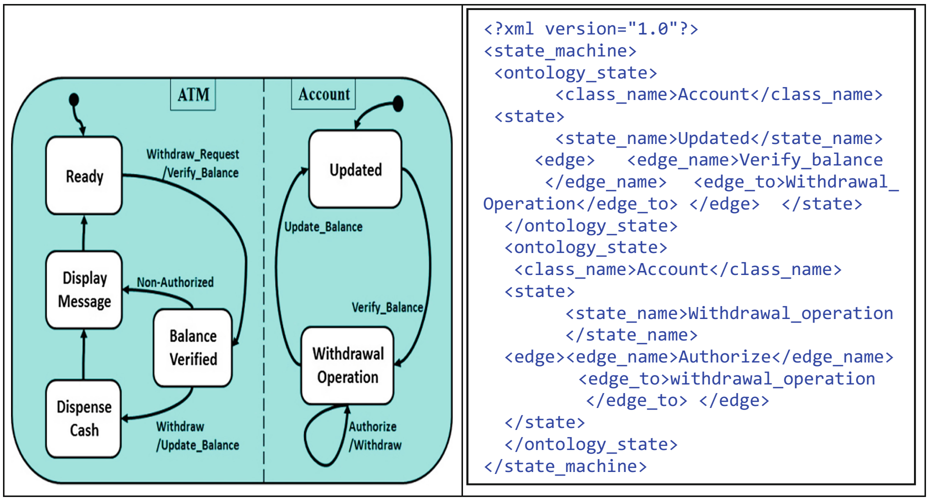
Generated Model and Running Code Implementation. KODEGEN is fed with an XML ontology and say, the ATM specification in Fig. 2. It generates model classes and a test script. Here the classes are in the Ruby language (Fig. 6).

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



```
<?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>Account</class_name>
 <state>
       <state_name>Updated</state_name>
     <edge>  <edge_name>Verify_balance
      </edge_name>  <edge_to>Withdrawal_
Operation</edge_to> </edge>  </state>
  </ontology_state>
  <ontology_state>
   <class_name>Account</class_name>
  <state>
        <state_name>Withdrawal_operation
        </state_name>
  <edge><edge_name>Authorize</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 of the partial account ontology states, for internal manipulation within KODEGEN.

```
class ATM
    attr_accessor :deposit
    attr_accessor :Verify_account
    attr_accessor :dispense_cash
    attr_accessor :Accept_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.

```
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 – The stub (here and in subsequent figures) is a method with a signature, but not implemented yet, needed to pass the test.

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.

A Gherkin specification file is given in Fig. 10.

```
<?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.

Figure 11 contains the generated model classes.

Figure 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

```
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.

```
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.

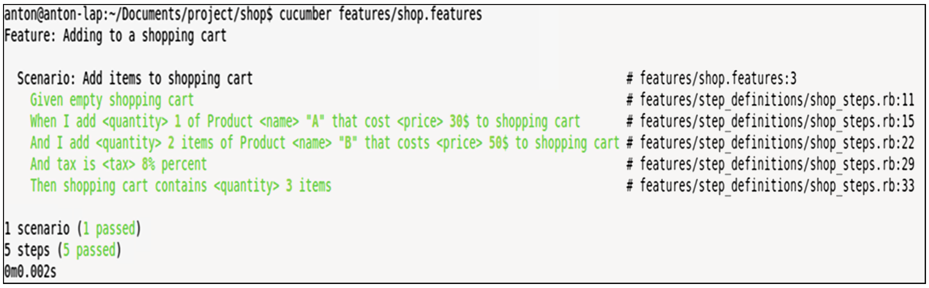
mock expectations are met by adding calls to stub objects – in bold red in Fig. 16. The script adds products A and B to empty cart, applies tax and make assertions.

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. 13). This test script can later be reused and re-issued to check correctness of the actual developing implementation.

```
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.



```
anton@anton-lap:~/Documents/project/shop$ cucumber features/shop.features
Feature: Adding to a shopping cart

  Scenario: Add items to shopping cart                                                    # features/shop.features:3
    Given empty shopping cart                                                             # features/step_definitions/shop_steps.rb:11
    When I add <quantity> 1 of Product <name> "A" that cost <price> 30$ to shopping cart        # features/step_definitions/shop_steps.rb:15
    And I add <quantity> 2 items of Product <name> "B" that costs <price> 50$ to shopping cart # features/step_definitions/shop_steps.rb:22
    And tax is <tax> 8% percent                                                           # features/step_definitions/shop_steps.rb:29
    Then shopping cart contains <quantity> 3 items                                        # features/step_definitions/shop_steps.rb:33

1 scenario (1 passed)
5 steps (5 passed)
0m0.002s
```

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.

Lastly for this case study, Fig. 13 is a screenshot resulting from 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.

5.3 Extended ATM-Account System with a Card

In this case study we extend the ATM-Account system (Sect. 5.1) with a Card, whose ontology is seen in Fig. 14. The respective ontology states are seen in Fig. 15.

```
<?xml version="1.0"?>
<ontology>
<class>
        <name>Card</name>
        <attribute>accepted</attribute>
        <attribute>returned</attribute>
        <attribute>retained</attribute>
    </class>
</ontology>
```

Fig. 14. XML representation of card ontology – This is the third ontology to be added to the two ontologies of the ATM-account system.

```
<?xml version="1.0"?>
<state_machine>
    <ontology_state>
        <class_name>Card</class_name>
        <state> <state_name>insertion</state_name>
            <edge><edge_name>password_entered</edge_name>
                        <edge_to>validation</edge_to> </edge>
        </state>
    </ontology_state>
    <ontology_state>
        <class_name>Card</class_name>
        <state> <state_name>validation</state_name>
            <edge> <edge_name>valid</edge_name>
                         <edge_to>validated</edge_to> </edge>
        </state>
    </ontology_state>
    <ontology_state>
        <class_name>Card</class_name>
        <state> <state_name>validation</state_name>
            <edge><edge_name>not_valid</edge_name>
                <edge_to>rejected</edge_to>
            </edge>
        </state>
    </ontology_state>
    <ontology_state>
        <class_name>Card</class_name>
        <state> <state_name>validated</state_name>
            <edge><edge_name>operation_completed</edge_name>
                        <edge_to>returned</edge_to> </edge>
        </state>
    </ontology_state>
</state_machine>
```

Fig. 15. XML representation of card ontology states – The states are: insertion, validation, validated.

```
Feature: Account Withdrawal
    Scenario: Successful withdrawal from an account
                    Given Account has a balance of <balance> $100
                    When <amount> $20 are withdrawn from ATM
                    Then the account balance should be <balance> $80
    Scenario: Account has sufficient funds
                    Given the account balance is <balance> $100
                    And  the card is valid
                    And the ATM contains <amount> $500
                    When the Account request cash <amount> $20
                    Then the ATM should dispense cash <dispense_cash> $20
                    Then the account balance should be <balance> $80
                    And  the card should be <returned> "returned"
    Scenario: Account has insufficient funds
                    Given the account balance is <balance> $10
                    And  the card is valid
                    And the ATM contains <amount> $50
                    When the Account Holder withdraw <amount> $20 from ATM
          Then the ATM should Print <message>"there are insufficient funds"
                    And  the account balance should be <balance> $10
                    And  the card should be <returned> "returned"
    Scenario: Card has been disabled
                    Given account
                    And the card is not valid
                    And the ATM contains <amount> $500
                    When the Account request cash <amount> $20
                    Then the card should be <retained>"retained"
              And the ATM should Print <message>"the card has been retained"
```

Fig. 16. Scenarios involving a card in the ATM-account system – The card may be valid (in bold green color) and it is returned, but still there may be sufficient or insufficient funds for a withdrawal operation. If the card is not valid (in bold red color, within the last scenario), it is retained. This scenario is based upon a user story found in http://dannorth.net/whats-in-a-story/ (Color figure online).

The next system input is the set of scenarios, now involving the Card, seen in Fig. 16.

In the next Fig. 17, one can see the Ruby code generated by KODEGEN with the testing steps.

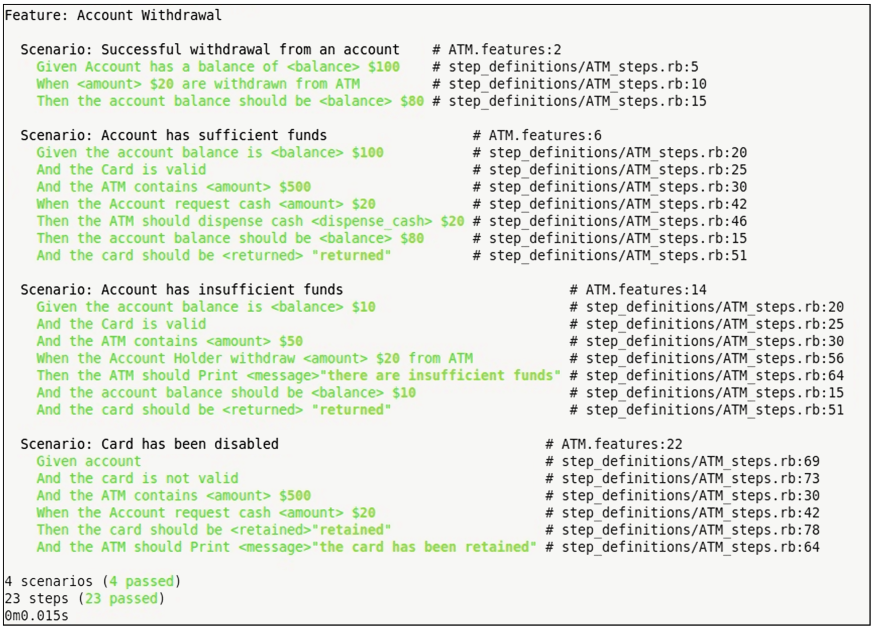
The work order in a test script (such as in Fig. 17) is as follows: objects are created in the *Given* part; messages are printed in the *When* part; verification occurs in the *Then* part. A metaclass is used to save a message to be verified in the *Then* part. This is an example of a Ruby meta-programming feature to dynamically add methods to (yet) non-existing model classes and later test the right interaction with them.

Finally, Fig. 18 displays a screenshot of the respective testing run.

```
require "/usr/lib/ruby/vendor_ruby/cucumber/rspec/doubles"
require "test/unit/assertions"
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+)$/ do |balance|
        @account.stub(:balance).and_return(balance)
        assert balance == @account.balance.to_s             end
Given /^the account balance is <balance> \$(\d+)$/ do |balance|
        @account = Account.new
        @account.balance = balance                          end
Given /^the Card is valid$/ do
        @card = Card.new
        @card.valid()                                       end
Given /^the ATM contains <amount> \$(\d+)$/ do |amount|
        @atm = ATM.new
        @atm.contains = amount
        print_manager=mock("Print_Manager")
        metaclass = class << print_manager; self; end
        metaclass.send :attr_accessor, :text
        def print_manager.print(text)
                @text = text        end
        @atm.print_manager=print_manager                    end
When /^the Account request cash <amount> \$(\d+)$/ do |amount|
        @account.request(amount)                            end
Then /^the ATM should dispense cash <dispense_cash> \$(\d+)$/ do |dispense_cash|
        @atm.stub(:dispense_cash).and_return(dispense_cash)
        assert dispense_cash == @atm.dispense_cash.to_s  end
Then /^the card should be <returned> "(.*?)"$/ do |returned|
        @card.stub(:returned).and_return(returned)
        assert returned == @card.returned.to_s              end
When /^the Account Holder withdraw <amount> \$(\d+) from ATM$/ do |amount|
        account_holder = mock( "Account_Holder ")
        account_holder.stub!( :withdraw ).with( amount ) do
                @atm.withdraw( amount )     end
        account_holder.withdraw( amount )                   end
Then /^the ATM should Print <message>"(.*?)"$/ do |message|
        @atm.print_manager.print( message )
        assert @atm.print_manager.text == message           end
Given /^account$/ do
        @account = Account.new                              end
Given /^the card is not valid$/ do
        @card = Card.new
        @card.not_valid()                                   end
Then /^the card should be <retained>"(.*?)"$/ do |retained|
        @card.stub(:retained).and_return(retained)
        assert retained == @card.retained.to_s              end
```

Fig. 17. KODEGEN generated Ruby code with testing steps – Mock objects for concepts (as *printer* and *account_holder*) not appearing in the system ontologies are stressed in bold red (Color figure online).



```
Feature: Account Withdrawal

  Scenario: Successful withdrawal from an account    # ATM.features:2
    Given Account has a balance of <balance> $100    # step_definitions/ATM_steps.rb:5
    When <amount> $20 are withdrawn from ATM         # step_definitions/ATM_steps.rb:10
    Then the account balance should be <balance> $80 # step_definitions/ATM_steps.rb:15

  Scenario: Account has sufficient funds                     # ATM.features:6
    Given the account balance is <balance> $100              # step_definitions/ATM_steps.rb:20
    And the Card is valid                                    # step_definitions/ATM_steps.rb:25
    And the ATM contains <amount> $500                       # step_definitions/ATM_steps.rb:30
    When the Account request cash <amount> $20               # step_definitions/ATM_steps.rb:42
    Then the ATM should dispense cash <dispense_cash> $20 # step_definitions/ATM_steps.rb:46
    Then the account balance should be <balance> $80         # step_definitions/ATM_steps.rb:15
    And the card should be <returned> "returned"             # step_definitions/ATM_steps.rb:51

  Scenario: Account has insufficient funds                                # ATM.features:14
    Given the account balance is <balance> $10                            # step_definitions/ATM_steps.rb:20
    And the Card is valid                                                 # step_definitions/ATM_steps.rb:25
    And the ATM contains <amount> $50                                     # step_definitions/ATM_steps.rb:30
    When the Account Holder withdraw <amount> $20 from ATM                # step_definitions/ATM_steps.rb:56
    Then the ATM should Print <message>"there are insufficient funds" # step_definitions/ATM_steps.rb:64
    And the account balance should be <balance> $10                       # step_definitions/ATM_steps.rb:15
    And the card should be <returned> "returned"                          # step_definitions/ATM_steps.rb:51

  Scenario: Card has been disabled                                      # ATM.features:22
    Given account                                                       # step_definitions/ATM_steps.rb:69
    And the card is not valid                                           # step_definitions/ATM_steps.rb:73
    And the ATM contains <amount> $500                                  # step_definitions/ATM_steps.rb:30
    When the Account request cash <amount> $20                          # step_definitions/ATM_steps.rb:42
    Then the card should be <retained>"retained"                        # step_definitions/ATM_steps.rb:78
    And the ATM should Print <message>"the card has been retained" # step_definitions/ATM_steps.rb:64

4 scenarios (4 passed)
23 steps (23 passed)
0m0.015s
```

Fig. 18. Account withdrawal test run for the four scenarios – A screenshot showing that all the steps in all the four scenarios were passed, after a few interactive improvement iterations.

6 Discussion

The KODEGEN tool, as applied to the case studies described in this work, clearly demonstrate the feasibility of the approach, for applications in their scale range. Thus, the concrete realization of a specification into a running test script is done through KODEGEN. We have opened and resolved a series of specific issues resulting into an evolution of the tool itself.

Next we discuss some of the characteristics of KODEGEN.

6.1 KODEGEN Characteristics

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

KODEGEN embodies quite a significant knowledge as a set of rules to handle common patterns and idioms when dealing with inputs. For example, during the test script generation, an object under test is recognized according to the ontology and by its appearance in 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.

A significant step taken in the tool evolution was the modification of the Gherkin syntax through the introduction of <tags>, needed to fill certain gaps between ontologies and executable specification.

Mock object libraries are not necessarily mandatory – as only concepts not essential for the system ontologies need to be implemented by mocks. But mock objects may be used to pass tests, to enable the system developer to test the model integrity.

6.2 Future Work

Among issues still open to investigation is the extent of KODEGEN automation: will it remain a useful quasi-automatic tool? Or will the automation gap be safely and significantly covered, approaching the efficiency and reliability of current compilers?

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. Will a certain language assume a definitive role for KDE?

The case studies in the current work still are of limited scope. Can we extend the current tool and techniques to industrial production of large scale software systems? We are also building a GUI based tool that will better support the iterative human aided process needed for growing the models for a large project.

Given a set of ontologies, how to determine the amount of scenarios needed to develop a consistent and stable system?

Finally, a most important issue is the ability to overcome gradual, ad hoc and localized improvements, to reach a stage of generalized techniques that are independent of specific applications.

6.3 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 *Runnable Knowledge* abstraction level.

References

1. Adzic, G.: Test Driven .NET Development with FitNesse. Neuri, London (2008)
2. Adzic, G.: Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing. Neuri, London (2009)
3. Adzic, G.: Specification by Example – How Successful Teams Deliver the Right Software. Manning, New York (2011)
4. Beck, K.: Test Driven Development: By Example. Addison-Wesley, Boston (2002)
5. Boehm, B.W.: Software engineering economics. IEEE Trans. Softw. Eng. **10**, 4–21 (1984)
6. Brown, A.W.: Model driven architecture: principles and practice. Softw. Syst. Model **3**, 314–327 (2004). doi:10.1007/s10270-004-0061-2
7. Calero, C., Ruiz, F., Piattini, M. (eds.): Ontologies in Software Engineering and Software Technology. Springer, Heidelberg (2006)

8. Chelimsky, D., Astels, D., Dennis, Z., Hellesoy, A., Helmkamp, B., North, D.: The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends. Pragmatic Programmer, New York (2010)
9. Exman, I., Llorens, J., Fraga, A.: Software knowledge. In: Exman, I., Llorens, J., Fraga, A. (eds.) Proceedings of SKY 2011, 2nd International Workshop on Software Knowledge (2010)
10. Exman, I., Yagel, R.: ROM: a runnable ontology model testing tool. In: Fred, A., Dietz, J.L.G., Liu, K., Filipe, J. (eds.) Knowledge Discovery, Knowledge Engineering and Knowledge Management, pp. 271–283. Springer, Heidelberg (2012)
11. Freeman, S., Pryce, N.: Growing Object-Oriented Software, Guided by Tests. Addison-Wesley, Boston (2009)
12. KODEGEN – the tool (2013). https://github.com/AntonLitovka/KODEGEN
13. Moq – the simplest mocking library for .NET and Silverlight (2012). http://code.google.com/p/moq/
14. North, D.: Introducing Behaviour Driven Development. Better Software Magazine (2006). http://dannorth.net/introducing-bdd/
15. NUnit (2012). http://www.nunit.org
16. Pan, J.Z., Staab, S., Assmann, U., Ebert, J., Zhao, Y. (eds.): Ontology-Driven Software Development. Springer, Heidelberg (2013)
17. Parreiras, F.S.: Semantic Web and Model-Driven Engineering. John Wiley and IEEE Press, Hoboken (2012)
18. RSpec mocks library (2013). https://github.com/rspec/rspec-mocks
19. Smart, J.F.: BDD in Action Behavior-Driven Development for the Whole Software Lifecycle. Manning, New York (2014)
20. SpecFlow – Pragmatic BDD for .NET (2010). http://specflow.org
21. Wynne, M., Hellesoy, A.: The Cucumber Book: Behaviour Driven Development for Testers and Developers. Pragmatic Programmer, New York (2012)
22. Yagel, R.: Can executable specifications close the gap between software requirements and implementation? In: Exman, I., Llorens, J., Fraga, A. (eds.) Proceedings of SKY 2011 International Workshop on Software Engineering, pp. 87–91. SciTePress, France, (2011)