ROM: A Runnable Ontology Model Testing Tool

Iaakov Exman and Reuven Yagel

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

Abstract. In the quest for the highest possible abstraction of software systems, Runnable Knowledge has been proposed for MDA. But in order to test in practice such a system design one needs to actually run the model. This work precisely describes the necessary steps by which ROM – a Runnable Ontology Model tool – automatically generates a running model from the designed Runnable Knowledge. The novel idea is to use ready-made mock object libraries to efficiently obtain the running model code. The tool feasibility is demonstrated by substituting its modules by the semi-automatic concatenation of existing tools, each performing the role of one of its modules. Detailed examples are provided to illustrate each of the ROM generation steps.

1 Introduction

It is common wisdom that software development costs are reduced when one discovers errors as early as possible along the development. Within an MDA – Model Driven Architecture – approach, early means in a higher level of abstraction. Within software engineering lifecycles, early means already at the requirement phase.

Elsewhere Exman et al. [8] have proposed Runnable Knowledge as the highest possible abstraction level, viz. one starts from the bare system concepts and their possible states.

This paper proposes a systematic approach to Model Testing starting from Runnable Knowledge by running the model. Runnable Knowledge is expressed by a set of ontologies and their relevant states. We assume that for a given problem domain the relevant ontology and its states are given a priori. The tool generates from the ontology and its states the classes of the system under development (SUD). The tool also generates in parallel the tests to be applied to the SUD.

ROM – a Runnable Ontology Model tool – is described. It is expected to automatically generate a running model and its tests, based on higher level knowledge extracted from specifications and related ontologies. The version described in this paper is semi-automatic.

Next we discuss alternative executable approaches of testing of specifications and design for software systems.

1.1 Executable Approaches to Specification and Design Testing

One could roughly divide executable approaches to specification and design testing

Exman I. and Yagel R..
ROM: A Runnable Ontology Model Testing Tool.
DOI: 10.5220/0004176800220030
In *Proceedings of the 3rd International Workshop on Software Knowledge* (SKY-2012), pages 22-30
ISBN: 978-989-8565-32-7

into two categories.

In the first approach one gradually builds an executable specification which can exercise the developing system. The final product of this approach is a full system specification that while it has been built it has also validated that a running system behaves as expected. A typical example is using tools like Cucumber, see e.g. the book by Wynne and Hellesoy [15].

In the second approach one essentially stays at the model level, and the final product of this approach is an executable model, which must later on be converted into the actual software system. A typical example is Executable UML in the book by Mellor and Balcer [10].

Our own proposed approach has an intermediate character. On the one hand it produces running code in a programming language. On the other hand, this code contains only synthetic (mock) objects and must still be converted later on into the actual software system. The synthetic objects are as usual mainly intended for agile testing.

1.2 Related Work

Here we present a concise review of the relevant literature.

In recent years the Agile software community emphasizes various early testing methods, e.g. Freeman and Pryce [9]. One of its main purposes is to obtain short feedback loops in order to properly guide the development of software systems in a world of fast changing environments and technologies.

Those testing methods came out from the unit-testing practice of Test Driven Development (TDD) by Beck [4]. These methods develop scripts demonstrating the various system behaviors, instead of code specification of just the interface and required behavior of certain modules. These are also known as automated functional testing, as the referred scripts' execution can be automated.

One such method is known as Acceptance Test Driven Development (ATDD) or alternatively just as Agile Acceptance Testing, e.g. Adzic [2]. Another group of methods extending TDD is Behavior Driven Development (BDD) North [12]. It stresses stakeholder readability and shared understanding. Recent representatives of the above approaches are e.g. Story Testing, Specification with examples Adzic [3] or just Living/Executable Documentation, e.g. Brown [5].

Some related tools for implementing these practices are FitNesse and Cucumber.

FitNesse by Adzic [1] (and its predecessor FIT) are wiki-based web tools allowing non-developers to write acceptance tests, in a formatted manner, e.g. tabular example/test data.

Cucumber, see e.g. the book by Wynne and Hellesoy [15] and related tools, e.g. SpecFlow [14], directly support BDD. Their main feature is the ability to run stories written in plain natural language (originally English but by now a few dozen others). This tool is also highly connected to unit testing tools and user/web automation tools. In a previous work by Yagel [16] we have discussed more extensively these practices and tools.

A representative reference for Executable UML is the book by Mellor and Balcer, [10]. An introductory overview of ontologies in the software context is found in Calero et al. [6].

In the remaining of the paper we introduce the Runnable Knowledge abstraction level (section 2), describe agile design and testing for Runnable Knowledge (section 3), propose a method for automatic code generation from Runnable Knowledge (section 4), describe a case study in detail (section 5) and conclude with a discussion (section 6).

2 The Runnable Knowledge Abstraction Level

Runnable Knowledge (Exman et al. [8]) is an abstraction level above standard UML models. The latter usually separates modeling of structure and behavior, typically into class diagrams and statecharts.

Both classes and statecharts may contain detailed attributes and functions with their parameters. Runnable Knowledge is based upon concepts, their relationships and possible states, abstracting detailed attributes, functions and parameters.

In the next sub-sections we first describe concepts and relationships within an ontology and then their respective possible states.

2.1 Ontology Concepts and States

To gradually illustrate our ideas, step by step, we have recourse to a widely used example – cash withdrawal from an Automatic Teller Machine (ATM).

Two ontologies – ATM and bank account – are used to deal with the ATM example. These are seen in Fig. 1. In Fig. 2 one can see these ontologies' states.

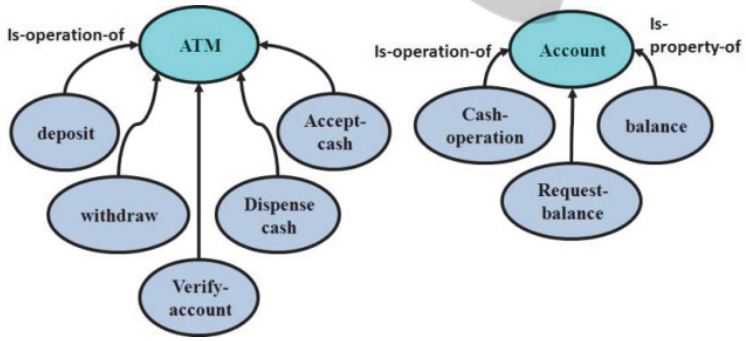

Fig. 1. ATM and bank Account Ontologies – The concepts in the ATM ontology (left) are operations that can be performed by the machine. The concepts in the Account ontology (right) are either operations (Cash-operation and Request-Balance) or a property (balance) of the Account.

3 Testing Runnable Knowledge

In this section we handle the same ATM example from an agile testing angle. In Fig. 3 one sees a requirements example for correct cash withdrawal from an ATM system.

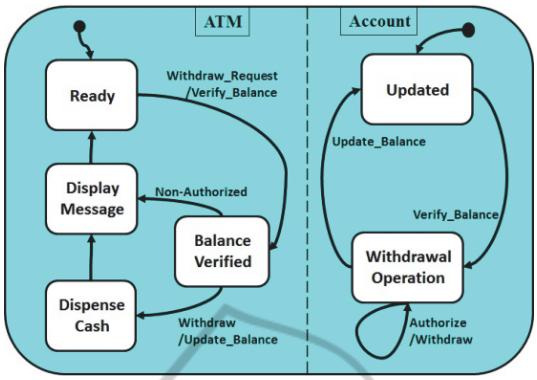

Fig. 2. ATM and bank Account Ontologies' States –These are the parallel states for the ATM and the account. This diagram only displays the states for a cash withdrawal operation. Other operations, such as a deposit or retrieval of account balance have similar diagrams.


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

Fig. 3. Initial Requirements for ATM – These are requirements for successful cash withdrawal from an ATM. They are expressed in the Gherkin style used by the Cucumber tool.

The example in Fig. 3 uses a simple syntax called Gherkin (see Chelimsky in ref. [7]). Its main feature is the ability to describe specification examples in close to plain natural language but in a way that will be easy to transform it into a runnable test script.

The keywords shown here in blue are: a) *Feature*: for giving a title to this specification part; b) *Scenario*: a title for a specific walk through; c) *Given*: the pre-conditions before some action is taken; d) *When*: an action that triggers the scenario; e) *Then*: the expected outcome; f) Optional: *And, But*: additional steps happening in one of the previous stages.

Some other keywords are available which are not covered here.

3.1 Agile Design and Testing

The requirements written in a specification file – such as in Fig. 3 – are usually developed together with the system's stakeholders or business experts.

At first, running this specification will fail since there is no code supporting it. This will lead the developers to design a domain model to satisfy this specification.

A tool like cucumber can suggest needed steps for satisfying the suggested scenarios. Mock objects could stand for the missing concepts and allow to re-run the feature.

Usually an initial version of a specification is developed and tested. This specification is iterated and refined until a complete as possible specification is

achieved. This specification is realized by detailed runnable test scripts which demonstrate the correctness of the system. They can even catch software regressions while adding new features to a system.

4 Automatic Generation of Running Code

In this section we carefully describe ROM, the tool for automatic generation of running code from the Runnable Knowledge level of abstraction.

4.1 ROM Software Architecture

Software system development begins with the elicitation of system requirements into the initial specification.

The initial specification, still needing verification, together with the ontology and its states are the input to the ontology fusion module of ROM (in Fig. 4). The fusion module uses the concepts in the ontology to generate a model with interfaces. It uses the ontology states to generate the test script.

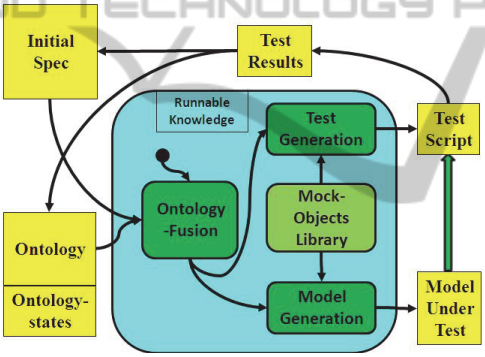

Fig. 4. ROM Software Architecture – modules are represented as round (green) rectangles, while input and output are regular (yellow) rectangles. The wide arrow pointing from the Model Under Test (MUT) to the Test Script, means that the latter is used to the test the MUT.

From the latter one generates, using a mock object library, the mock objects which constitute the runnable code of the Model under test (MUT). This is tested by unit tests generated in parallel.

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 – consisting of the ontologies and their states – is the utmost abstract level in the software layers hierarchy. It is runnable in the sense that, by use of a suitable tool, one can make transitions between states.

The essential role of mock object libraries, proposed in this work, is the usage to obtain a fast and efficient translation of Runnable Knowledge into an *actually* running model. This is illustrated in the following case study.

5 Case Study: Internet Purchase

Here we describe in detail a case-study - an internet purchase case - havinga clearly different nature from the above ATM cash withdrawal example, in order to show the range of applicability of our approach.

The Internet Purchase case study requires modeling of two classes: the shopping cart and products that can be put in the cart for later purchase. Testing of these classes is schematically represented by a transaction in which two products are purchased. We first show the relevant ontologies and states.

Internet Purchase Runnable Knowledge. The internet purchase Runnable Knowledge consists of the ontology (seen in Fig. 5) and its states (seen in Fig. 6) of two classes: 1) Shopping cart – that may contain products; 2) Product – with properties, say its Price.

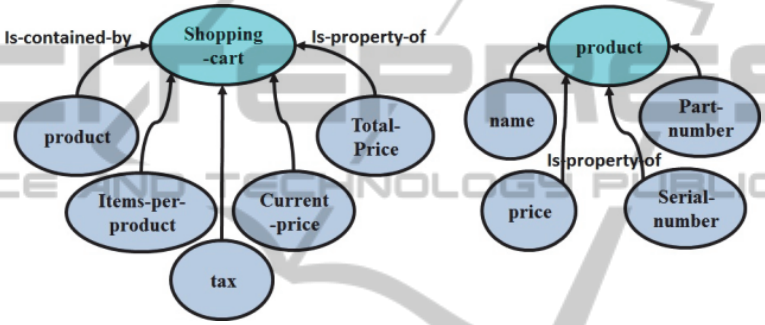

Fig. 5. Shopping cart and Product Ontologies – The Shopping cart ontology (left) shows objects contained by the cart (product and items-per-product) and purchase properties (total-price, current-price and tax). The Product ontology (right) concepts are just its properties.

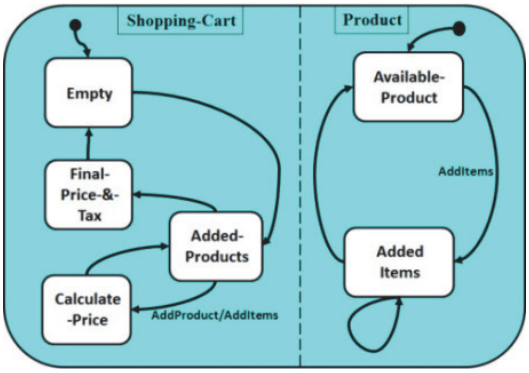

Fig. 6. Shopping-cart Ontology States – it shows two parallel states: *shopping-cart* and *product*. The product is either available (default state) or added to the cart. The cart default is empty. A product can be added, its price or final price-&-tax calculated, ending the transaction.

Internet Purchase Model Testing. A feature file is seen in Figure 7.

```
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 ($20 each)
   And the tax is 8%
   Then the shopping cart contains 3 items
  And the total price is 54$
```

Fig. 7. Shopping-cart: Scenario – Adding items to a shopping cart.

Internet Purchase Model and Running Implementation. We now show an example of ROM's output: a C# code using the following tools/libraries:

- SpecFlow (a .NET Cucumber variant) for running the feature [14];
- NUnit (a .Net unit test framework) hosting test run and assertions [13];
- Moq (a .Net mock library) for setting up domain mocked objects [11].

Running ROM on the above specification and ontology is expected to output:

- model with interfaces containing properties and method signatures (Fig. 8),
- mock expectation setups (Fig. 9),
- a runnable test script (Fig. 10).


```
namespace ShoppingCartModels
{  public interface IShoppingCart
    { void Add(IProduct product);
      int Quantity {get; set;}
      double TotalPrice {get; set;}
      double Tax {get; set;}}
   public interface IProduct
    { int Price {get; set;}}}
```

Fig. 8. Shopping-cart: Extracted model – one sees two interfaces corresponding to the concepts in the ontology: IShoppingCart and IProduct. For each property, there will be generated getter and setter methods.

```
public partial class StepDefinitions
{ private static Mock<IShoppingCart>
                    shoppingCartMock;
  private IShoppingCart shoppingCart;
  private static Mock<IProduct>
                productAMock;
  private static Mock<IProduct>
                productBMock;
  [BeforeTestRun()]
  public static void SetupMockExpectations()
  {  shoppingCartMock = new     Mock<IShoppingCart>();
     shoppingCartMock.Setup(sc => sc.Quantity).Returns(3);
     shoppingCartMock.Setup(sc => sc.TotalPrice).Returns(54.0);
     productAMock = new Mock<IProduct>();
     productAMock.Setup(product => product.Price).Returns(10);
     productBMock = new Mock<IProduct>();
     productBMock.Setup(product => product.Price).Returns(20);
   }
}
```

Fig. 9. Shopping-cart: Mock expectation setups – for each of the mock objects expectations are setup: the shopping cart's quantity and total price, the product A price and the product B price.

Figure 9 shows the expectation setups for the properties of the mock objects, viz. the shopping cart and the products A and B.

```
[Binding]
 public class StepDefinitions
 {[Given(@"I have an empty shopping cart")]
   public void GivenIHaveAnEmptyShoppingCart() {
         shoppingCart = shoppingCartMock.Object; }
   [When(@"I add one item of Product A \(10\$\)")]
   public void WhenIAddOneItemOfProductA10() {
         IProduct productA = productAMock.Object;
             shoppingCart.Add(productA); }
   [When(@"I add two items of Product B \(20\$ each\)")]
   public void WhenIAddTwoItemsOfProductB20Each()
         {IProduct productB = productBMock.Object;
           shoppingCart.Add(productB); shoppingCart.Add(productB);}
   [When(@"the tax is 8%")]
   public void WhenTheTaxIs8() {
          shoppingCartMock.Setup(cart => cart.Tax).Returns(8.0);}
    [Then(@"the shopping cart contains 3 items")]
    public void ThenTheShoppingCartContains3Items()
          {Assert.That(shoppingCart.Quantity, Is.EqualTo(3));}
    [Then(@"the total price is 54\$")]
    public void ThenTheTotalPriceIs54() {
       Assert.That(shoppingCart.TotalPrice, Is.EqualTo(54.0));
       shoppingCartMock.VerifyGet(cart => cart.TotalPrice);
         }
    }
```

Fig. 10. Shopping-cart: Runnable test script – the script contains the following steps: start with empty cart; add product A; add two items of product B; the tax is 8%; assert quantity of items; assert total price and correspondent interaction.

Figure 10 displays the test script for the Shopping cart case study. Note that the test script could be further refactored by using regular expression and joining methods, e.g. adding items with different prices (we left that out for simplicity).

Once the mock expectations were set and the test scrip is ready, there only remains to actually run it in a test runner tool, as seen in the screenshot in Fig. 11. This test script can later be reused and re-issued to check correctness of the actual developing implementation.

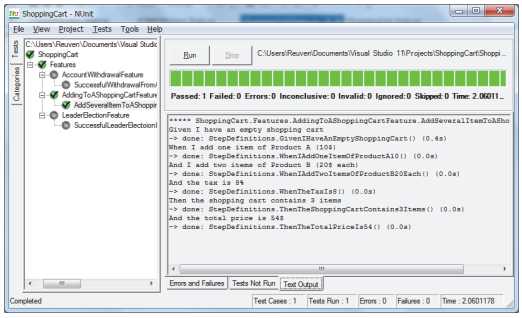

Fig. 11. Shopping-cart: Running test results – screenshot of running of the above test script within NUnit's test runner. It is a passing test, since all expectations where met by the models, all of the steps in the test script were successfully done.

6 Discussion

The agile process embodied in ROM contributes significantly to the understanding of all aspects of a system under development. As an example, we have learned by

working on the shopping cart system's specification that one also needs a "Quantity" property that was previously lacking. This was therefore added to the initial ontology.

Among future issues to be investigated is to what extent ROM can be made fully automatic, or it will remain a useful, but quasi-automatic tool. A more extensive response should deal with each of the ROM modules in separate.

Concerning implementation, ROM will be configured to produce code using other languages\libraries, e.g. Ruby which is more concise than, e.g., C#/Java. The tool can also use a specific language to improve the produced scripts, e.g., using partial classes in C# to separate expectations from the test script.

The main contribution of this work is the usage of mock objects as a fast implementation means to check system design still in the Runnable Knowledge 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, USA, 2002.
5. 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.
6. Calero, C., Ruiz, F. and Piattini, M. (eds.): Ontologies in Software Engineering and Software Technology, Springer, Heidelberg, Germany, 2006.
7. 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.
8. Exman, I, Llorens, J. and Fraga, A.: "Software Knowledge", pp. 9-12, in Exman, I., Llorens, J. and Fraga, A. (eds.), Proc. SKY'2010 Int. Workshop on Software Engineering, 2010.
9. Freeman, S., and Pryce N.: Growing Object-Oriented Software, Guided by Tests, Addison-Wesley, Boston, MA, USA, 2009.
10. Mellor, S. J. and Balcer, M. J.: Executable UML – A Foundation for Model-Driven Architecture, Addison-Wesley, Boston, MA, USA, 2002.
11. Moq – the simplest mocking library for .NET and Silverlight: (see http://code.google.com/p/moq/), 2012.
12. North, D.: "Introducing Behaviour Driven Development", Better Software Magazine, (see http://dannorth.net/introducing-bdd/), 2006.
13. NUnit: (see http://www.nunit.org), 2012.
14. SpecFlow – Pragmatic BDD for .NET: (see http://specflow.org), 2010.
15. Wynne, M. and Hellesoy, A.: The Cucumber Book: Behaviour Driven Development for Testers and Developers, Pragmatic Programmer, New York, USA, 2012.
16. 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, Portugal, 2011.