An Approach to Modularization in Model-Based Testing

Teemu Kanstrén, Olli-Pekka Puolitaival, Juho Perälä
VTT Technical Research Centre of Finland
Oulu, Finland
{teemu.kanstren, olli-pekka.puolitaival, juho.perala}@vtt.fi

***Abstract*—Test models in model-based testing are typically represented as state machines in terms of states and transitions. These states and transitions also are typically the focus of the test modeling approaches. Yet these test models are basically software components for the test automation domain, and need to be considered from test automation and software engineering viewpoints. In this paper, we describe a modeling approach that takes better into account these viewpoints. Taking these viewpoints into account, we propose a modularization approach for modeling in model-based testing and present a tool for supporting this modularization approach.**

Keywords-model based testing; test automation; modularization

I. INTRODUCTION

Model-based testing (MBT) is an advanced test automation technique focused on generating test cases from state-based models. In recent years several MBT tools have been presented and the industrial adoption of MBT techniques has been increasing [1,2]. The underlying modeling approach in these different tools is typically state-based, augmented by some programming language constructs to embed test instructions inside the test model to produce executable test cases from the state-machine transitions. The test generation is guided by the algorithms analyzing the model and parsing these programming language constructs to form test sequences and test data.

As such, this modeling approach can be seen as similar to other programming tasks. State transitions are executed to move from one state to another and these executions are performed in terms of programming language instructions embedded in these transitions. Yet the test modeling approach is based almost solely on state-machine notions– transitions between states and guard statements defining when transitions are allowed. While it is recognized that different domains and abstraction levels are important in MBT (e.g., [3]), the domain of test automation in itself is not considered in the common MBT modeling approaches. Relations between the test models and other software engineering artefacts are sometimes considered (e.g., [4]) but not the composition of the test models themselves. As these test models are in practice software components in themselves, we believe it is possible to provide a more efficient test modeling approach by introducing good practices from the software engineering domain, and specifically the software test automation domain into the test modeling approach itself.

Based on this background, we present a modularization approach for test modeling in MBT. This includes further modularization of the different traditional state-machine elements (transitions and guards) as well adding new ones specific to test automation (test oracles as specific transitions). We further present a modeling approach for describing test input and expected output in terms of a taxonomy of runtime invariance as described by our earlier work [5]. Finally, we identify a set of additional topics for future study that we believe will enable taking these approaches further. Similar to our inspiring domain of generic software engineering, we believe the end result helps achieve easier test model creation, evolution and maintenance. This forms a basis for more effective test modeling. The approach is implemented in a tool called OSMOTester, available as open-source [6].

The rest of the paper is structured as follows. Section II describes the background concepts relevant to this paper. Section III describes our test model modularization approach. Section IV discusses the concept in a wider context. Finally, conclusions summarize the paper and future works.

II. PRELIMINARIES

This section introduces the background concepts relevant for this paper.

A. Modularization

Modularization is one of the basic concepts in software engineering in general. Van der Hoek and Lopez [7] provide an overview of software modularization and its different aspects in software engineering. They show how modularization has been considered important from the early days of programming and has since evolved to all aspects of modern software engineering, and continues to be an important research question. The aspects they describe include programming languages, software architectures and software evolution. They also note the need to avoid excess modularization where not necessary. In terms of addressing modularity, van der Hoek and Lopez [7] list a number of benefits such as reduced complexity, enabling parallel work, enabling evolution (easier understanding, resilience to change, etc.), and easing reuse. The cost is described in terms of requiring added effort for composition of the modular pieces to form a whole.

While this shows that modularity is considered important and is addressed in many software engineering domains, the consideration in MBT has focused on state-machine concepts [8] and not considered modularization in terms of test automation concepts and model elements generally. These are the concepts we address in this paper, in providing extended means to express modularity in test models, including im-

 ISBN: 978-1-61208-168-7

portant test automation concepts, and in mitigating the costs by providing automated support for module composition.

Considering modularity in software engineering in general, Sarkar et al. [9] list seven main properties of modularity. *Similarity of purpose* refers to grouping together elements related to providing a specific service. *Encapsulation* refers to encapsulating the internals of a module from its environment and external collaborators. *Compilability* refers to possible issues in compiling a module due to issues such as circular dependencies. *Extensibility* refers to providing means to extend a specific module without accessing its internals. *Testability* refers to the ease of testing the module. *Cyclic dependencies* negate many of the benefits of modularization and as such need to be avoided. *Module size* should be overall roughly equal. In the following sections we present our approach in Section IV also discuss how it relates to these properties.

B. Example System

Throughout the rest of this paper, we will use a simple vending machine example to illustrate the concepts discussed. This vending machine is a modified version of the example used in [2]. The relevant part of the operation of the machine can be described as the set of following properties:

- Accepts 10, 20, and 50 cent coins.
- When a total of 100 cents have been inserted the action *vend* is enabled.
- When *vend* is enabled, no more coins can be inserted (this assumption is relaxed later).
- When *vend* is activated, a bottle is produced, reducing the total number available and resetting the number of inserted cents to 0.
- When the machine is empty (no bottles), all actions are disabled.

Notice that for the sake of providing a concise example, this model simplifies several aspects such as providing change to the user when going over the total of 100 cents.

C. Model-Based Testing

The term model-based testing (MBT) can be defined in different ways. We follow Utting and Legeard [2] who describe MBT as "Generation of test cases with oracles from a behavioural model". The model describes the expected behaviour of the system under test (SUT), and is used by a MBT tool in order to generate test cases, in a form suitable for the test target, such as method invocation sequences and input data. The SUT output is checked by the test oracles also encoded into the model.

A typical model applied in the context of MBT is based on some form of states and transitions (sometimes referred to as pre/post conditions, historical or functional notations [8]), such as an extended finite state machine (EFSM). This describes the system behaviour in terms of states and transitions between these states. Basically a state can be described as a relevant combination of system internal variables. A transition forms an invocation of some functionality of the SUT, possibly affecting the observed state. Finally, guard statements over the transitions impose constraints on when a transition is allowed to happen.

To illustrate these concepts, Figure 1 shows an example snippet of a state-machine for the vending machine. In this example, *state 1* is where the machine includes 10 bottles and 100 cents have been inserted. As defined by the example guard statements, the *vend* transition to *state 2* is allowed if there are bottles available in the machine and 100 cents have been inserted.

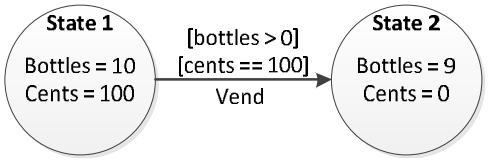

Figure 1. Vending machine model snippet.

In this case, both of these conditions have been met and thus this transition is enabled. Once it is taken, the number of bottles is reduced to 9 as one is deducted and the number of inserted cents is reset to zero. In this example, state is composed of the number of bottles available as well as the number of inserted cents.

To produce suitable test cases from such a model, this information is then transformed to test data for the SUT as defining the input to be given to invoke this transition on the SUT. This can be, for example, a message to the vending machine to produce a *vend* action. This is information encoded into the model by the user.

The test model must also define the expected output for each taken transition (the test oracle) in order to validate the correctness of the responses from the SUT for the given input. For example, the *vend* request should impact the reported number of bottles and produce a response as the "bottle" itself. This can be encoded in the test model along with the transition as an expected result, forming a test oracle.

Thus, to produce a suitable test model, the model must embed in itself means to identify test input to stimulate the SUT, the expected output for each input to produce a test oracle that gives verdict if the test passes or not, and a test harness that actually links the execution of the tests with the actual SUT.

III. TEST MODEL MODULARIZATION

The typical EFSM modeling notation for MBT was presented in the previous section. From this, we can list the following components that are needed to produce a suitable test model for test generation.

- Transitions to define what are the possible test steps for a specific SUT in its current state.
- A representation of the SUT state for the model.
- Test oracles to check the correctness of the received output and the internal state of the SUT when different transitions are taken.
- Guard statements to define when a specific transition is allowed to be taken.
- Test input to be linked to each transition/test step.

A. Traditional Modeling Approach for MBT

Figure 2 shows an example snippet of a test model for the vending machine, using the notation of the OSMOTester

 ISBN: 978-1-61208-168-7

MBT tool. This is based on existing works such as Model-JUnit described in [2], modified to better address the modularization aspects discussed in this paper. In this example, the variable *sut* represents the system and can either directly delegate the commands to the SUT itself or to a scripter writing a test script for later execution. The internal state of the model is composed of the *cents* and *bottles* variables. The guards are methods identified by the *@Guard* annotation and providing a Boolean value (*true* for allowing the transition). A transition is matched to its guard by the name given for both the *@Transition* and *@Guard* annotations.

```
private int cents = 0;
private int bottles = 10;
private VendingMachine sut = new VendingMachine();

  @Guard("10cents")
  public boolean allow10cents() {
    return cents <= 90 && bottles > 0;
  }

  @Transition("10cents")
  public void insert10cents() {
    sut.insert(10);
    cents += 10;
    assertEquals(cents, sut.cents());
  }

  @Guard("vend")
  public boolean allowVend() {
    return cents == 100;
  }

  @Transition("vend")
  public void vend() {
    sut.vend();
    cents = 0;
    bottles--;
    assertEquals(cents, sut.cents());
    assertEquals(bottles, sut.bottles());
  }
```

Figure 2. Example model snippet.

Test oracles are represented inside the transitions by the *assertEquals()* method calls (here from the JUnit test framework [10]) that compare the state of the test model to the state of the SUT. Input is given to the SUT in each transition. For example, *sut.insert(10)* in the "10cents" transitions, where 10 represents the number of cents inserted.

This is an example of the traditional approach to MBT, where every transition encodes all test components, including test input, and test oracles. In this approach, there is also a direct mapping from a single guard to a single transition. This is how traditionally most MBT tools expect the test models to be provided and how they process them (see e.g., [1] for comparison).

Figure 2 also illustrated two basic aspects of a test model in the terminology of this paper. What we term as control-flow in this aspect is the way the MBT tool traverses the EFSM expressed by this model, in evaluating the guards and taking suitable transitions as chosen by the active test generation algorithm. Together with this, we use the term data-flow to describe how the state variable values of the model evolve as the MBT tool traverses over the control-flow. For the vending machine, this translates to the evolution of the *cents* and *bottles* variables over time.

B. Modularizing the Control-Flow Modeling

Besides representing the guards, transitions and states of the EFSM as their own components, the traditional approach presented in the previous subsection is not very modular. It does not consider the separation of the different aspects of test inputs and test oracles (and associated test output). Furthermore, by assuming a direct one-to-one mapping from guards to transitions, the flexibility of the EFSM modeling is limited. As these test models are in practice software components themselves, this leads to several problems from their evolution and maintenance viewpoints such as duplication, low cohesion and weak separation of concerns.

To address these issues, we introduce new test model components and refine existing components. This includes more advanced guard composition, extensions for more explicit test oracles as components of the test model itself, and objects for generating input data and evaluating output data. In this subsection we discuss the control-flow elements and in the following subsection the data-flow elements.

The typical approach to modeling guard statements in an EFSM is to provide a specific guard attached to a specific transition as illustrated by Figure 2, where both the *10 cents* and *vend* are transitions and have a single dedicated matching guard statement. Taking the guard statement for *10 cents* as an example, it provides assertions over two separate concepts, the number of bottles and the number of cents. To help separate these concerns and provide a manageable modeling notation, we extend guard modeling by allowing decomposition of guard statements for a transition into several separate guards, and to share a single guard statement across several transitions as needed.

The decomposition aspect is illustrated in Figure 3. This decomposition provides for more cohesive structure where different concerns are addressed by different guards. In this case, the checks over the different state variables have been split into separate guard statements, each mapping to their specific transition by their name (e.g., "10cents"). The end result is more cohesive guard and better separation of model concerns.

```
@Guard("10cents")
 public boolean checkMaxCents() {
   return cents <= 90;
 }

 @Guard("10cents")
 public boolean checkBottles() {
   return bottles > 0;
 }
```

Figure 3. Guard decomposition.

However, decomposition alone does not fully address the need for providing cohesive guard statements over all the different transitions in the model. For example, if we consider the vending machine example, there are several transitions for inserting different types of coins (*10, 20, 50 cents*). In practice, none of these or the *vend* transition, should be allowed to execute if there are no bottles available. To support this, we need to provide specific shared guard statements.

This is illustrated in Figure 4 where the first guard *checkBottlesExist()* is shared by all transitions in the model,

and the second one is shared by the three listed transitions (*10, 20, 50 cents*). The first of these examples is an example suitable for our simple vending machine example as is. The second one is an example of how we might model a common guard in a case where the user is allowed to insert enough coins for several bottles at once and we need to check that the total of inserted coins does not go over the number of available bottles.

```
@Guard
public boolean checkBottlesExist() {
  return bottles > 0;
}

@Guard({"10cents", "20cents", "50cents"})
public boolean allowMoreCoins() {
  return bottles >= (cents/100);
}
```

Figure 4. Shared guard example.

In addition to having guards and transitions govern how test sequences are generated, we have to consider the evaluation of the test results. This is done by a test oracle and is traditionally part of each transition as shown in Figure 2 (the assert statements). In many cases, specific checks are needed for transitions to check their specific results. However, it is also commonly important to evaluate the general state of the model against the matching state in the SUT. To support more explicit modeling of test oracles, we extend our modeling notation to add general test oracles for program state over several transitions. These are similar in decomposition to the shared guards and identified by the *@Oracle* annotation. Figure 5 illustrates this with an example for the two state variables shown in Figure 2 for the vending machine. In practice, our MBT tool sees these as specific transitions to be executed between other transitions. It is also possible to relate them to specific transitions with the style of *@Oracle("transition-name")* similar to guards. In our models, generic oracles apply regardless of existence of specific ones. Any oracle matching a transition is always evaluated.

```
@Oracle
public void evaluateBottles() {
  assertEquals(bottles, sut.bottles());
}

@Oracle
public void evaluateCents() {
  assertEquals(cents, sut.cents());
}
```

Figure 5. Generic oracle example.

As these generic checks can be expressed separately and evaluated specifically by OSMOTester, they not only allow for the modular expression of generic test oracles but also add more power to the verification functionality of the test model and the MBT approach itself. In Figure 5, the state of the model and the state of the SUT are now evaluated to match continuously without the need to express them explicitly over each transition. Any deviation is thus captured as soon as it occurs and not possibly several transitions later in the *vend* transition (if at all) as was the case in Figure 2.

Finally, we also need to consider how and where test generation should be stopped. The typical approach in MBT is to describe the test model as a state machine with the expectation that test cases can be generated and the model can be traversed at different points, where test generation should practically always be enabled. The choice of what transitions to take and when to stop the test generation is mainly up to the test generation algorithm. However, there are points where it is possible that no transition is enabled and the typical modeling approach gives no indication to test generation as to how this should be evaluated. The generic algorithms used to generate tests from the test model cannot know how to evaluate this condition for a specific SUT and its test model. For example, in the case of the vending machine example, if the *vend()* transition is taken 10 times, the number of bottles will reach zero and the shared guard *checkBottlesExist()* will cause a state where no new transitions are available. At this point, the test generation tool cannot know if this should be treated as a failure or as a clue to end test generation for this step.

To enable the model to express this kind of information, we add a new annotation called *@EndCondition* and as illustrated in Figure 6. When a method with this annotation returns *true,* it is taken as an indicator that the current test generation from this model should be stopped and a new test case should be started. If no transitions are available and there is no *@EndCondition* that returns *true*, the current test case is reported by the tool as a failure. This will most likely indicate a problem in the test model itself. It should be noted that this annotation is not required for the test generation algorithms to stop test generation but they can also stop in other phases where found appropriate by the algorithm. It can also be used at any point to describe conditions to stop test generation, regardless of the state of the model.

We also provide specific notations for setting up new test cases and shutting down a running system using notations such as *@BeforeTest*, *@AfterTest*, *@BeforeSuite*, and *@AfterSuite*. We borrow these concepts from familiar tools such as JUnit [10] and TestNG [11], helping also to provide familiar concepts for other tool users. They basically define methods that are to be executed before and after a test case or a test suite respectively, regardless of the algorithms and end conditions.

```
@EndCondition
public boolean endIfNoBottles() {
  return bottles == 0;
}
```

Figure 6. Expressing end conditions.

Using our new control-flow modeling notations we can thus produce the model shown in Figure 7. Notice that there is now only a single test oracle where all test assertions are centralized. The transitions can now focus on performing actions on the SUT and updating the model state accordingly. Also, the transition *10cents* no longer requires a guard statement as it is fully covered by the shared guard statement (also applying to *vend*). Finally, the model can no longer enter an unknown state as the end condition for a state with no bottles is explicitly specified.

 ISBN: 978-1-61208-168-7

```
private int cents = 0;
private int bottles = 10;
private VendingMachine sut = new VendingMachine();

  @Guard
  public boolean checkBottlesExist() {
    return bottles > 0;
  }

  @Transition("10cents")
  public void insert10cents() {
    sut.insert(10);
    cents += 10;
  }

  @Guard("vend")
  public boolean allowVend() {
    return cents == 100;
  }

  @Transition("vend")
  public void vend() {
    sut.vend();
    cents = 0;
    bottles--;
  }

  @Oracle
  public void evaluateState() {
    assertEquals(bottles, sut.bottles());
    assertEquals(cents, sut.cents());
    assertTrue(cents >= 0);
    assertTrue(cents <= 100);
    assertTrue(bottles >= 0);
  }

  @EndCondition
  public boolean endIfNoBottles() {
    return bottles == 0;
  }
```

Figure 7. The model snippet in updated notation.

C. Modularizing the Data-Flow Modeling

The modeling notation described so far in the previous section shows how we can modularize the control-flow aspects of test modeling in MBT. From the viewpoint of data-flow we need to consider also the input- and output-data values and their respective constraints. Input data needs to be generated for the different parameters given to the SUT, and needs to respect the set of expected constraints for the SUT functions they are linked to. But since full coverage of most input combinations is not possible to achieve, we must also define a set of constraints to define what type of test data should be generated. The output must similarly consider the constraints for the output values received from the SUT as response to the provided stimuli (input).

To support modeling these data-flow constraints, we provide a generic library of objects we term as invariants objects. These are based on our previous work in identifying different aspects of runtime invariance in software behavior [5]. Each invariant object allows one to specify a set of constraints over the data value it represents and to use these as a basis to perform actions such as generate input data or evaluate the conformance of given (output) values. These allow for addressing data-flow invariance for a specific value in a single object, effectively modularizing the constraints over a single variable in a single object.

An updated model for the vending machine using this notation is shown in Figure 8. This time, the use of the invariant objects for data-flow representation allows for a compact representation, and this includes transitions for all possible coin types and vending. By adding the shared guard and end condition from Figure 7 the model will include all the important model elements for the vending machine. The invariant objects presented in the figure are specified for integer data types, and we currently support the basic data types of integers, floating points, Booleans and character strings. The constraints supported by these are defined according to the taxonomy presented in [5].

Note that this model slightly changes the expected behavior of the vending machine towards a more realistic one. This specification now accepts any number of coins and deducts 100 from the number of inserted coins when *vend* is applied. It also collapses all *insertXXCents()* transitions (*10, 20, 50*) from the previous models into a single one, where the input is represented by a single invariant object defining the allowed input values. This is the *ci* object (short for *centInput* for space reasons in the figure) for the input value definitions. Test oracle expectations for both *cents* and *bottles* variables are expressed by the *co* and *bo* variables (short for *centOracle* and *bottleOracle* for space reasons in the figure).

```
  private IntInvariant ci = new IntInvariant();
  private IntInvariant co = new IntInvariant();
  private IntInvariant bo = new IntInvariant();

  public void TestModel() {
    //set up allowed input values
    ci.addValue(10);
    ci.addValue(20);
    ci.addValue(50);

    //set up evaluation constraints
    co.setMin(0);
    bo.setMin(0);
    bo.setMax(10);
  }

  @Transition("insertCoins")
  public void insertCents() {
    int coin = ci.input();
    sut.insert(coin);
    cents += coin;
  }

  @Guard("vend")
  public boolean allowVend() {
    return cents >= 100;
  }

  @Transition("vend")
  public void vend() {
    sut.vend();
    cents -= 100;
    bottles--;
  }

  @Oracle
  public void evaluateState() {
    assertEquals(bottles, sut.bottles());
    assertEquals(cents, sut.cents());
    co.evaluate(cents);
    bo.evaluate(bottles);
  }
```

Figure 8. Data-flow modularization.

D. Further modularization support

In the previous sections we have shown how to build a modularized test model in our notation. So far we have included the elements needed to build a useful model for generating test sequence and test data. Additionally, it is also important in test automation to be able to express how the generated test cases cover different requirements, a concept also supported by different MBT tools [2]. We support this through a special requirements object as illustrated in **Figure 9**. Note that defining the requirements twice is not required (add() and covered() methods) but doing so allows the tool to report the coverage percentage.

```
@RequirementsField
private Requirements req = new Requirements();
@TestSuiteField
private TestSuite s = null;

public TestModel() {
  req.add("vend");
}

@BeforeTest
public void startTest() {
  System.out.println("Starting Test "+s.count());
}

@Transition("vend")
public void vend() {
  req.covered("vend");
  …
}
```

Figure 9. Additional supported test model components.

Figure 9 also illustrates how the modeler can access the test generation history by adding a *TestSuite* field that will be initialized by the MBT tool before starting test generation. This provides useful information for evaluating and debugging the test model itself. It also allows the user access to the current test case object, for example, enabling the user to set the test case as passed or failed for any special reporting extensions in the MBT tool.

So far we have discussed several new notations for MBT that help produce more modular test models. However, an important question of decomposing the objects representing this notation remains. Besides expressing all elements in a single model, we also support decomposing the model objects into several sub-models. When these are registered into OSMOTester, it will parse them all and match all the expressed model elements into a single internal representation. This effectively allows one to, for example, represent the test oracles in one partial model, guards in another, transitions in a third, and the remaining ones in fourth. Merging is based on handling the model element naming across the different model objects as if they were one.

Finally, we also support the basic set of modular aspects for any MBT tool as discussed in [12]. It is possible to plug in different test generation algorithms, algorithms for defining length of generated test suites and test cases, and to attach various test harnesses and test analysis tools. Figure 10 shows an example of a test sequence visualization tool we provide as a plugin to the core OSMOTester itself, using as output a test listener interface provided by the core.

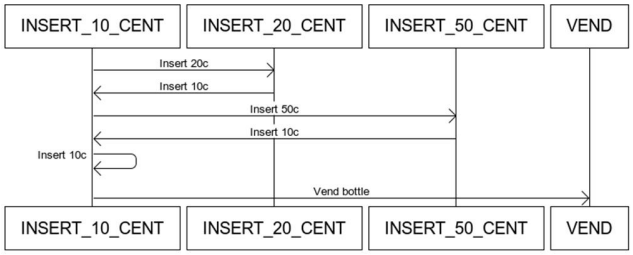

Figure 10. Test sequence visualization.

This visualization shows each transition in the test model as a box, and the sequences of transitions taken as arrows from one to another. In this case, the user has chosen to use "10cents" as the default state, which is why the arrows seem to originate from this box. This is just one of the available visualizations as an example of something that can be plugged in to describe the test cases.

E. Modularization Summary

Figure 11 illustrates the overall flow of the different control-flow modules in our model. Before test suite generation commences, *@BeforeSuite* annotated methods are executed. Before each new test generation, *@BeforeTest* annotated methods are executed. *@Guard* methods are checked for enabled transitions, of which one is picked by the test generation algorithm. After each transition any associated *@Oracle* methods are executed. If *@EndConditions* exist, they are evaluated for stopping criteria for a single test case. If not, the criterion is left to the test generation algorithm. *@AfterTest* methods are executed when a test generation stops, and *@AfterSuite* when all test generation stops.

Data-flow support is defined in terms of invariant objects defining constraints over data-flow values that support generating input and evaluating output. Coverage requirements can be expressed in the test model as objects of their own, and models can be built from separate model objects as best seen fit. We see further developments in terms of more complex combinations of data-flow and control-flow elements as discussed next.

In relation to the different properties of modularization that were discussed in section II.A, we improve on several of these properties. *Similarity of purpose* is supported by more explicit grouping of elements such as test oracles. *Extensibility* is supported by allowing composition of the model elements (test steps/transition components) from several different objects and linking them automatically together. It is possible to add new test oracles, guards and other elements as separate model objects without touching existing ones (also helping address similarity composition). We avoid *cyclic dependencies* and enhance *compilability* by keeping elements separate and associating by the transition name metadata only (vs. strict static linking). Our framework is *encapsulated* in a set of simple annotations, providing only minimal exposure on the test models. We make *module size* easier to manage with finer granularity of model elements. *Testability* is mostly handled in itself by generating tests from the model and executing them against the SUT as in MBT in general, verifying both the test model and the SUT.

 ISBN: 978-1-61208-168-7

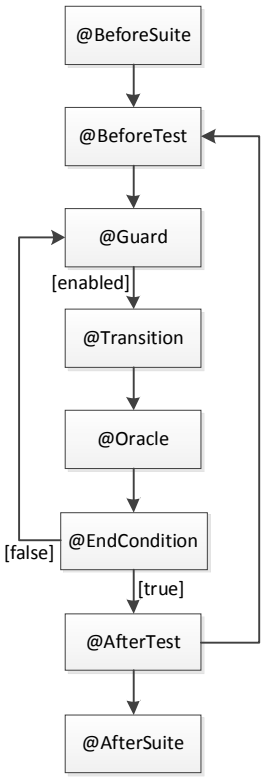

Figure 11. Control-Flow Summary.

IV. DISCUSSION

Modularization is one of the key aspects of good software engineering in general. As we have discussed and illustrated in the previous sections, test modeling in MBT is practically a software engineering activity in itself. It is basically about engineering a piece of software that generates a test suite in terms of the MBT framework, provided by available tools and libraries. While the traditional test modeling approaches for MBT have been lacking proper modularization mechanisms, we provide a set of means to achieve increased modularity in test modeling. This provides for increased separation of concerns, more cohesion and less duplication. As generally in software engineering, this helps achieve higher maintainability and supports model evolution.

While we advocate the use of our new modularized modeling notation, our approach also fully supports the more traditional modeling approach. It is possible to fully compose the model of transitions, guards and embedding all the information inside these without any modularization. This means just using a specific subset of our modeling notation (*@Transition* and *@Guard*). It is our experience that the best result in practice is to combine parts of the different approaches where most benefit can be gained. That is, modularizing the most common parts while keeping specific parts where it makes more sense according to the case at hand.

As our models are written in a standard programming language (Java), it is also possible to decompose the models into modules in terms of classes and methods. This can help achieve some of the benefits discussed here in itself (by using classes and objects), but it is also our experience that as the model is made more explicit in terms of generic test oracles, guards and invariant objects, this helps build a more explicit and understandable model. This also further helps in the model creation, evolution and maintenance, where human understanding is typically the key factor in software engineering. The use of a common programming language also helps more generic modularization as we can make use of the wide set of existing Java libraries.

Another aspect related to the modularization of test models is the modularization of the test models into representing different viewpoints of the SUT behavior. While the test models we have presented in this paper describe the expected correct behavior of the vending machine, another interesting aspect is the modeling of the failure behavior of the SUT. In case of the vending machine, this would include trying to insert incorrect values, access the vending functionality with less than 100 coins, using negative values, and so on. These different viewpoints can be modeled as separate models addressing these specific constraints along with matching test oracle definitions. This is a form of modularization itself.

Related to the *@Oracle* elements we have also found that it is useful to be able to access selected pre-transition state when checking the overall state after the transition. Similarly we have noted that this notation can be used for extended purposes such as supporting data collection for test reporting. Thus we are looking into options to extend this to support both with *@Pre* and *@Post* transition annotations as extensions of the current *@Oracle* approach.

In relation to the invariant objects, we described our support for test modeling in terms of data-flow invariants based on our earlier work on creating a taxonomy of runtime invariance in software behavior [5]. While we currently use this invariance to provide support for data-flow modeling, the taxonomy in itself is more extensive in describing also patterns over control-flow and various invariant scopes. This is a topic for future work in providing more extensive support for modeling software behavior.

Our experience thus far has been that we can modularize control-flow in terms of the generalized test oracles and guard conditions (in a way defining invariant control-flow patterns), but the more advanced support in terms of runtime invariance is challenging to express in a textual support in a way that is natural for human consumption. In terms of data-flow modeling, we also see the use of data-flow constraints to support test generation more widely in terms of boundary conditions, category partitions, and other relevant test data constraint and analysis definitions. This requires pairing data generation algorithms with the invariant constraint definitions. At the same time our experience has also been that using more domain friendly names (e.g., splitting integer invariants into value range and similar objects) is easier to understand and we are evolving the expressiveness of the data-flow elements into this direction. These are some of the more advanced research topics in relation to invariant objects that are out of the scope of the modularization approach described in this paper but relevant for future studies.

Similarly related to extending the modeling approach is the combination of different properties of invariance expressed in the model. For example, in the model presented in

 ISBN: 978-1-61208-168-7

Figure 8, the *ci* variable presents a set of input constraints for the generated test data related to the operations over the *coins* state variable. At the same time the *co* state variable presents a set of constraints over the expected values of the same state variable. Expressing these relations such as how the input should be constrained in relation to the current value of the state variable is challenging. This also applies to combining these invariants more generally over different control-flow aspects and data-flow aspects. Some viable approaches could be found in existing works such as combining evolutionary testing with MBT (e.g., [13]).

Many of these aspects come back to the limitations of the textual modeling approach in relation to the advanced concepts presented by the taxonomy in [5]. Thus one interesting aspect in relation to this is the application of visual modeling tools and domain-specific concepts. We have previously studied combining visual representations in terms of domain-specific models (DSM) to provide more visual support for test modeling [14]. Combining this to provide more intuitive support for representing complex interactions over the invariant properties is another interesting research topic for future studies. While DSM commonly considers the models it builds from the perspective of the application domain of a specific company, the modeling notation we describe here can also be seen as a form of a domain specific language for test modeling in the domain of MBT.

While we have so far discussed mainly aspects related to dynamic analysis and modeling related aspects, there are also several points where static analysis can be useful. This includes algorithms and techniques such as symbolic execution to generate test paths to reach defined requirements, test end conditions, automated input data boundary analysis, and other similar optimizations. These are aspects supported already in many advanced (commercial) MBT tools. So far, we have focused on the modularization of the dynamic runtime aspects. Extending this to the domain of static analysis of these models is out of the scope of our work but an interesting and relevant topic for future works.

We have applied our approach and tool on several commercial projects and keep evolving it according to our experiences in these projects. Due to the nature of the projects we cannot disclose their details. However, they have been successful in improving the aspects of model creation, management and evolution described here. Similarly, it has helped make the adoption of MBT approach easier, also leading to reduced costs in test automation.

V. Conclusions and Future Work

In this paper, we have presented a modularization approach for test models in model-based testing. This approach extends the traditional approach that focuses on the state-machine abstraction by considering common software engineering aspects and specific components of test automation. While the traditional approach focuses on state-machine abstractions in terms of guards and transitions, we extend this to include new state-machine elements for test models, including shared guards over several transitions, generic test oracles over the general system tests, and test end conditions. These are mainly related to the control-flow aspects of behaviour modeling, and we further provide added support for data-flow representations in terms of objects describing properties of runtime invariance over system behavior.

Finally, we identify potential topics for future works in terms of extending invariant object representations to control-flow aspects and their relation to the data-flow over different scopes (as identified in our previous work), and also more extensively in terms of test automation components such as input value boundary and category analysis. We also identify extensions of the modularization into the domain of static analysis, and providing more human-friendly modeling notations for complex models and invariant objects as interesting topics for future work.

VI. References

[1] O-P. Puolitaival, M. Luo, and T. Kanstrén, "On the Properties and Selection of Model-Based Testing Tool and Technique," in *1st Workshop on Model-Based Testing in Practice (MOTIP)*, 2008.

[2] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*.: Morgan Kaufmann, 2007.

[3] W. Prenninger and A. Pretschner, "Abstractions for Model-Based Testing," *Electronic Notes in Theoretical Computer Science*, vol. 116, pp. 59-71, 2005.

[4] Q. Farooq, M. Z. Zohaib, Z. Malik, and M. Riebisch, "A Model-Based Regression Testing Approach for Evolving Software Systems with Flexible Tool Support," in *17th IEEE In'tl. Conf. and Workshops on Engineering of Computer-Based Systems*, 2010, pp. 41-49.

[5] T. Kanstrén, "Towards a Taxonomy of Dynamic Invariants in Software Behaviour," in *2nd Int'l. Conf. on Pervasive Patterns and Applications (PATTERNS)*, 2010.

[6] T. Kanstrén. (2011, July) OSMOTester. [Online]. http://code.google.com/p/osmo/

[7] A. van der Hoek and N. Lopez, "A Design Perspective on Modularity," in *10th Int'l. Conf. on Aspect-Oriented Software Development*, Pernambuco, Brazil, 2011, pp. 265-279.

[8] M. Utting, A. Pretschner, and B. Legeard, "A Taxonomy of Model-Based Testing Approaches," *Software Testing, Verification and Reliability*, 2011.

[9] S. Sarkar, G. M. Rama, and A. C. Kak, "API-Based and Information-Theoretic Metrics for Measuring the Quality of Software Modularization," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 14-32, Jan. 2007.

[10] (2011, July) junit.org. [Online]. www.junit.org

[11] (2011, July) testng.org. [Online]. www.testng.org

[12] V. V. Kuliamin, "Component Architecture of Model-Based Testing Environment," *Programming and Computer Software*, vol. 36, no. 5, pp. 289-305, 2010.

[13] F. Lindlar, A. Windisch, and J. Wegener, "Integrating Model-Based Testing with Evolutionary Functional Testing," in *Third Int'l. Conf. on Software Testing, Verification and Validation Workshops*, 2010, pp. 163-172.

[14] O-P. Puolitaival and T. Kanstrén, "Towards Flexible and Efficient Model-Based Testing, Utilizing Domain-Specific Modelling," in *10th Workshop on Domain Specific Modelling*, 2010.

 ISBN: 978-1-61208-168-7