Mapping ASM Specifications to Spec Explorer: Guidelines, Benefits and Challenges

Jameleddine Hassine
Department of Information and Computer Science
King Fahd University of Petroleum and Minerals, Dhahran, Kingdom of Saudi Arabia
jhassine@kfupm.edu.sa

***Abstract*—Model-based testing (MBT) focuses on the generation of test suites, using models of system requirements and behavior. In order to get full benefits from model-based testing and to drive its adoption by practitioners, automation support is required. Spec Explorer 2010 is an MBT tool that offers a rich set of features allowing for modeling and analyzing software behavior using graphical visualization and automatic generation of test code from models. In this paper, we propose and discuss a set of guidelines to map the core Abstract State Machines (ASM) concepts and constructs into Spec Explorer 2010. We illustrate our mapping approach using examples and features from ASM-based formal specification languages *CoreASM* and *AsmL*. Furthermore, we discuss the benefits, the challenges and the limitations of our proposed mapping guidelines. Finally, we illustrate our approach using a case study of the well known dining philosophers problem.**

***Keywords*-Model-based testing; Spec Explorer 2010; Abstract State Machines (ASM); CoreASM; AsmL**

I. INTRODUCTION

Model-based testing (MBT) is a variant of testing that relies on explicit behavior models that describe the intended behaviors of a System Under Test (SUT) and/or the behavior of its environment. Test cases are then generated from one of these models or their combination, and then executed on the SUT [1]. MBT offers significant promise in reducing the cost of test suite generation, increasing the effectiveness of the produced test cases, and shortening the testing cycle.

Driven by technological advances and by the ever-growing need for producing high quality software, model-based testing (MBT) has emerged as a potential research domain. Given the popularization of models in software design and development, MBT has moved from a research topic [2], [3] to an emerging practice in the industry [4], [5], with increasing commercial tool support [6]. Dias-Neto et al. [2] have performed a systematic review of MBT research literature and have proposed a classification of the MBT approaches. A recent systematic review of state-based MBT tools by Shafique and Labiche [6] provides a detailed classification of nine commercial and research MBT tools. In a more recent work, Utting et al. [1] have developed a taxonomy of six dimensions that covers the key aspects of MBT approaches.

Model-based testing relies on three key aspects - the modeling notation, the algorithms used to guide test generation and the tools supporting on-the-fly generation or off-line realization of executable tests.

Many different notations and languages, with different expressive power, have been used for modeling the behavior of systems for test generation purposes. Many of these languages are discussed and classified by Hierons et al. [3] and, more recently, by Utting et al. [1]. Examples of such notations classification include *state-based Notations* (e.g., *Z*, *B*, etc.), *transition-based Notations* (e.g., FSMs, statecharts, etc.), *history-based notations* (e.g., temporal logics, *MSC*, etc.), *operational notations* (e.g., process algebra such as *CSP* and *CCS*).

Many test generation algorithms have been proposed [1], [2], [3]. Examples of such test generation approaches include *Random generation* (achieved by sampling the input space of a system), *Search-based algorithms* (e.g., graph search algorithms), *Model Checking* (for checking system properties), *Symbolic execution* (generating symbolic traces that should be instantiated to obtain test cases), *Deductive theorem proving* (used to check the satisfiability of formulas), *Constraint solving* (useful for selecting data values from complex data domains), etc.

Utting et al. [1] have proposed a classification of MBT tools [6] according to which kinds of test selection criteria they support (e.g., *structural model coverage*, *data coverage*, *requirements-based coverage*, *fault-based criteria*, etc.). Microsoft's *Spec Explorer 2010* [7], is one of the commercial tools that have led the MBT scene. The new release of *Spec Explorer 2010* [7], version 3.5 and integrated within Visual Studio 2010, offers a rich and powerful set of features for modeling and analyzing system functional behavior, as well as automatically generating test code.

The widespread interest in model-based testing techniques and tools provides the major motivation of this research. In particular, this paper serves the following purposes:

- Bridge the gap between ASM-based languages (e.g., *CoreASM* [8], *AsmL* [9], etc.) and the new release of *Microsoft Spec Explorer 2010* [7], where the SUT model is specified using .NET $C\#$. The previous version, *Spec Explorer 2006* [10], accepts models written in *AsmL* (Abstract State Machines Language) [9], which is no longer supported in *Spec Explorer 2010* [7]. To bridge this gap, we propose a set of guidelines to map ASM core concepts and constructs into *Spec*

 ISBN: 978-1-61208-230-1

Explorer 2010 modelling notation (i.e., $C\#$ program models and *Cord* scripts [11]).
- Allow for ASM model exploration and analysis. Indeed, *Microsoft Spec Explorer 2010* [7] provides several strategies for managing the exploration of a model, including data coverage of parameter values and structural model criteria [1]. Furthermore, the resulting *Spec Explorer 2010* models can be used to automatically generate test cases.
- Discuss the benefits, the challenges, and the limitations of the *ASM* to *Spec Explorer 2010* mapping approach.

The remainder of this paper is organized as follows. The next section presents some related work. A brief introduction to *Spec Explorer 2010* and an overview of its specification language are presented in Section III. Section IV represents the core of our paper, where we describe and discuss the *ASM* to *Spec Explorer 2010* mapping guidelines. In order to demonstrate the applicability of our proposed mapping approach, a case study of the well known *Dining Philosophers* problem is presented in Section V. A discussion of the challenges and the limitations of our proposed approach is presented in Section VI. Finally, conclusions are drawn in Section VII.

II. Related Work

Several approaches have been proposed to translate ASM specifications to languages and notations in order to enable testing and verification. Winter [12] has proposed a schema for transforming ASM models into the language of SMV. A similar approach has been implemented in *AsmetaSMV* by Arcaini et al. [13] for mapping and verifying ASM models, written in *AsmetaL*, into NuSMV notation. Gargantini et al. [14] have introduced an algorithm to automatically encode an ASM specification in *Promela*, the language of the model checker *SPIN*.

Grieskamp et al. [15] have proposed an algorithm that derives a finite state machine (FSM) from a given abstract state machine (ASM) specification. The generated ASM states, often infinite, are grouped into hyperstates which are the nodes of the FSM. The links of the FSM are induced by the ASM state transitions. This transformation allows for the integration of ASM specifications with the existing tools for test case generation from FSMs. In a related work, Barnett et al. [16] have presented a tool environment for model-based testing with the Abstract State Machine Language (*AsmL*) [9]. The environment supports semi-automatic parameter generation, FSM generation from ASMs [15], call sequence generation and conformance testing.

The most closely related work to ours is the one by Veanes et al. [17]. The authors have provided a symbolic analysis of model programs written in AsmL [9], in terms of a background $\mathcal{T}$ of linear arithmetic, sets, tuples and maps.

III. Overview of Spec Explorer 2010

Spec Explorer 2010 [7], which we simply refer to as *Spec Explorer*, provides a model editing, composition, exploration, and visualization environment within Visual Studio, and can generate on-line and off-line test suites. A Spec Explorer model consists of a set of rules, written in $C\#$, expressed in a model program (i.e., the *.cs* file) combined with behavioral descriptions coded in a scripting language called *Cord* [11] (i.e., the *.cord* file). The model program and the *Cord* script work together to make a testable model of the SUT.

Spec Explorer 2010 Specification

Model (.cs)

```
// importing libraries
using System;
using System.Collections.Generic;
// … other libraries
namespace myNameSpace
{
    public static class myClass
    {  // Defined state variables
    static int x, y;
    [Rule(Action = "Rule1()")]
    static void Rule1()
    {  // Rule1 body
    }
    [Rule]
    static void Rule2()
    {   // Rule2 body
    }
  }
}
```

Coordination Script (.cord)

```
Using myNameSpace
config Main
{
action abstract static void RuleAdapter.Rule1();
action abstract static void RuleAdapter.Rule2();
}
machine Program() : Main
{
construct model program from Main
}
machine Interleaving() : Main
{
 Rule1() ||| Rule2()
}
machine TestSuite() : Main
{
construct test cases where Strategy =
"ShortTests" for Interleaving
}
```

Figure 1. Structure of a typical *Spec Explorer* specification

Figure 1 illustrates a typical structure of a *Spec Explorer* specification. The model program (.cs) defines the classes of the system (e.g., *myClass*), the fields used to hold state data (e.g., x, y), and the rule methods for the actions (e.g., *Rule1* and *Rule2*) already declared in the *Cord* configuration. The model program is joined by a *Cord* script that defines the context signature (e.g., *Main*) and action machines (e.g., *Program*) expressing behaviors that further constrain the model for purposes of exploring submodels and generating practical test cases. A machine is based on one or more configurations and represents the unit of exploration. Many operators can be used to compose machine behavior from configurations and other behaviors. For example, the parallel behavior operator $|||$ is used in machine *Interleaving* to construct the interleaved parallel composition of *Rule1* and *Rule2*. *Cord* scripts also provide the means to generate test cases from the model. For a detailed description of *Spec Explorer*, we refer the reader to [7].

IV. ASM to Spec Explorer Mapping Guidelines

In this section, we present a brief overview of the basic ASM concepts and we provide a set of guidelines for translating them into *Spec Explorer*. For a detailed description of Abstract State Machines concepts and features, an interested reader is referred to [18].

 ISBN: 978-1-61208-230-1

Table I
EXAMPLES OF MAPPINGS OF ASM BUILT-IN TYPES

CoreASM [8]	**AsmL** [9]	**Spec Explorer** [7]
Boolean	Boolean	bool
enum/universe	enum	enum
String	String	string
Number	Integer/Byte/short	int/byte/short
	Long/Float/Double	long/float/double
Set	Set of T	SetContainer<T>
List	Seq of T	SequenceContainer<T>
Map	Map of T to S	MapContainer<T,S>

Although we illustrate the mapping guidelines using features and examples from *CoreASM* [8] and *AsmL* [9] languages, our proposed mapping guidelines can be applied to any ASM-based language, thus maintaining the generality of the discussion.

A. Mapping of States

Abstract State Machines (ASM) [18] define a state-based computational model, where computations (runs) are finite or infinite sequences of states $\{S_i\}$ obtained from a given initial state S_0 by repeatedly executing transitions δ_i:

$$S_0 \xrightarrow{\delta_1} S_1 \xrightarrow{\delta_2} S_2 \quad \cdots \quad \xrightarrow{\delta_n} S_n$$

The states of an ASM are multi-sorted first-order structures, i.e., domains of objects with functions and predicates defined on them.

1) Domains:: A domain consists of a set of declarations that establish the ASM vocabulary. Each declaration establishes the meaning of an identifier within its scope. *Spec Explorer* is based on $C\#$ language and it offers a rich set of data types covering almost all domains of ASM-based languages. Hence, the mapping of domains into their corresponding *Spec Explorer* data types is straightforward. Table I shows some examples of supported ASM-based built-in types and their corresponding *Spec Explorer* data types.

In order to match the mutable nature of ASM sets, sequences, and maps, we use the *Spec Explorer* mutable containers types: *SetContainer*, *SequenceContainer* and *MapContainer*.

2) Function Names:: Each function name *f* has an arity (number of arguments that the function takes). Function names can be static (i.e., fixed interpretation in each computation state) or dynamic (i.e., can be altered by transitions fired in a computation step). Static nullary (i.e., 0-ary) function names (i.e., called constants) and dynamic nullary functions (i.e., called variables) are mapped respectively into *Spec Explorer* constants and variables of the types presented in Table I.

ASM *n-ary* functions (i.e., f: T_1 x T_2 x ..... $T_n \rightarrow$ T), may be described using the *Spec Explorer MapContainer* construct. The same mapping applies to the ASM monitored, controlled and shared functions. For example, the following *CoreASM* code defines three enumeration domains *T1*, *T1* and *T*, each having 2 elements, and a 2-ary function *foo* defined over *T1* and *T2* and returning a value of type *T*:

```
enum T1 = {A1,B1}
enum T2 = {A2,B2}
enum T  = {A,B}
foo: T1 * T2 -> T
```

Model.cs

```
public enum T1 {A1, B1};
public enum T2 {A2, B2};
public enum T  {A, B};
public class T1T2
  {
   public T1 p1;
   public T2 p2;
   public T1T2(T1 Param1, T2 Param2)
   {
    this.p1 = Param1;
    this.p2 = Param2;
   }
  }
public class MyModelProgram
  {
   public static MapContainer<T1T2, T> foo = new MapContainer<T1T2, T>();
  }
```

Figure 2. Implementing n-ary functions in Spec Explorer

Figure 2 illustrates its corresponding *Spec Explorer* model implementation. A new class *T1T2* is created to map function *foo* input types *T1* and *T2*. The arity of the function defines the number of the new class attributes. The function name *foo* is mapped to a *MapContainer* object having keys of type *T1T2* and values of type *T*. It is worth noting that the new class *T1T2* could have been created as CompoundValue (i.e., public class T1T2 : CompoundValue), which is mutable but it cannot be updated outside of the constructor.

Another possible solution, proposed by Arcaini et al. [13], is to create $\prod_{i=1}^{n} |D_i|$ variables to cover all combinations of the domains elements. Considering the example above, four variables of type *T* can be created to cover the product of enumerations *T1* and *T2*: *public static T foo_A1_A2, foo_A1_B2, foo_B1_A2, foo_B1_B2;* This solution does not scale well in presence of domains with large sets of elements.

B. Mapping of Transition Rules

The transition relation is specified by guarded function updates, called *rules*, describing the modification of the functions from one state to the next. An ASM state transition is performed by firing a set of rules in one step.

1) Conditional Rule:: Typically, an ASM transition system appears as a set of guarded rules:

if *cond* **then** *RuleThen* **else** *RuleElse* **endif**

RuleThen and *RuleElse* are transition rules and *cond*, the guard, is a term representing a boolean condition. The ASM conditional rule can be implemented using *Spec Explorer*

Condition class, which provides conditions that can control the exploration of a rule. *Spec Explorer* does not generate a transition for a rule if it encounters in the execution path of the rule any condition that is not satisfied. The *Condition* class contains a rich set of methods for controlling exploration (e.g., IsTrue(Boolean), IsFalse(Boolean), IsNull(Object), etc.

Figure 3 illustrates the mapping of the ASM conditional rule. Condition *Condition.IsTrue(cond)* enables the exploration of *RuleThen()* if *cond* evaluates to true, while condition *Condition.IsFalse(cond)* enables the exploration of *RuleElse()* if *cond* evaluates to false.

```
static class ModelProgram
{
  static bool cond=true;
  static int stateVar;
  [Rule(Action = "RuleThen()")]
  static void RuleThen()
  {
   Condition.IsTrue(cond);
   // update state Variables
  }
```

```
[Rule(Action = "RuleElse()")]
  static void RuleElse()
  {
  Condition.IsFalse(cond);
  // update state Variables
  }
}
```

Figure 3. Implementation of the conditional rule in Spec Explorer

2) Update Rule:: The basic form of a transition rule is a function update:

$$f(t_1,t_2,\ldots,t_n) := \text{value}$$

$(f, (t_1,t_2,\ldots,t_n))$ represents the location and *value* is its content. An update can be implemented in *Spec Explorer* as a rule containing simple assignment statements in the case of dynamic nullary functions (e.g., var = value) or as an update of a *MapContainer* in case of n-ary functions, where an update results in adding a new element to the *MapContainer* (if the key does not exist), or as an update of a *MapContainer* element (if the key does exist). Figure 4 illustrates the mapping of the ASM update rule to *Spec Explorer* using the data structures introduced in Figure 2. The *MapContainer* class method *ContainsKey(D)* returns whether the key exists in the *MapContainer* (e.g., foo.ContainsKey(key)), while the method *Add(D, R)* adds a key-value pair to the *MapContainer* (e.g., foo.Add(key, value)).

Model.cs

```
public static MapContainer<T1T2, T> foo = new MapContainer<T1T2, T>();
[Rule]
public static void UpdateRule(T1T2 key, T value)
 {
  if (foo.ContainsKey(key))
     foo[key] = value;
  else
     foo.Add(key, value);
 }
```

Figure 4. Spec Explorer implementation of the update rule

3) Sequence Rule:: The sequence rule aims at executing rules/function updates in sequence:

seq $Rule1, \ldots, RuleN$ or **seq** $Update_1, \ldots, Update_n$

The resulting update set is the sequential composition of the updates, generated by $Rule1 \ldots RuleN$ in case of rules and generated by $Update_1, \ldots, Update_n$ in case of function updates. ASM sequential function updates can be mapped into one single rule in *Spec Explorer* as described in Figure 5(a), while ASM sequential rules can be implemented using the *Cord* composition operator ";" as described in Figure 5(b).

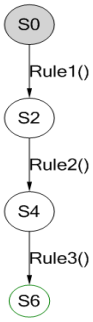
Model.cs

```
[Rule]
static void SequentialUpdates()
   { // Update1
     var1 = v1;
     // UpdateN
     varN = vN;
   }
```

(a) Model program for sequential function updates

Config.cord

```
config Main
{
action abstract static void RuleAdapter.Rule1();
action abstract static void RuleAdapter.Rule2();
action abstract static void RuleAdapter.Rule3();
}
machine Program() : Main
{
construct model program from Main
}
machine Sequence() : Main
{
Rule1() ; Rule2() ; Rule3()
}
```



(b) Cord file and generated FSM for the sequential rules

Figure 5. Spec Explorer implementation of the sequence rule

4) Choose Rule:: The choose rule consists in selecting elements (non deterministically), from specified domains, satisfying the guard φ, and then evaluates *Rule1*. If no such elements exist (i.e., ifnone), *Rule2* is evaluated.

choose x **in** D **with** φ(x) **do** $Rule1$ **ifnone** $Rule2$

Model.cs

```
[Rule(Action = "ChoiceRule()")]
static void ChoiceRule()
 {
  Predicate<int> P = (y => (y >= 1 && y <= 4));
  if (x == Choice.Some<int>(P))
    Rule1(x);
  else
    Rule2();
 }
```

Figure 6. Spec Explorer implementation of the choose rule

The *choose* rule may be implemented in *Spec Explorer* using the non-deterministic method *Some* of the *Choice* class

 ISBN: 978-1-61208-230-1

(i.e., Choice.Some<T>(Predicate<T> predicate)), which selects a value from a range of a specific type *T*, that satisfies a certain condition: *predicate*. If such an element exists, it will be passed to Rule1, otherwise a call to Rule2 is performed. Figure 6 illustrates an example of a mapping of the ASM choose rule to *Spec Explorer*. The predicate P corresponds to the guard φ, while the selected value, if any, is stored in variable *x*. An integer value is selected randomly from the range 1 ... 4, then passed to Rule1.

5) Extend Rule:: The extend rule is used to construct new elements and add them to a specific domain. The resulting update set is the updates generated by *Rule1*.

extend D **with** $x_1, \ldots, x_n$ **do** *Rule1*

In *Spec Explorer*, the extend rule may be implemented by adding new elements to a mutable container such as SetContainer, SequenceContainer, MapContainer, etc., then call *Rule1*. As shown in Figure 7, a new integer *x1* is added (using the *Add* method) to the existing SetContainer *D*, then *Rule1* is called on the new added element.

Model.cs

```
static SetContainer<int> D= new SetContainer<int>(1,2);
[Rule]
static void ExtendRule(int x1)
  {
   D.Add(x1);
   Rule1(x1);
  }
```

Figure 7. Spec Explorer implementation of the extend rule

It is worth noting that the *CoreASM* [8] extend rule has different semantics depending on whether *D* is a background (e.g., Collection, List, Set, etc.) or a *universe*. Such distinction is not made at the *Spec Explorer* level.

6) Block Rule:: If a set of ASM transition rules have to be executed simultaneously, a block rule is used:

par $Rule_1 \ldots Rule_n$ **endpar**

CoreASM uses the **par**... **endpar** syntax, while *AsmL* uses **step** $Rule_1 \ldots Rule_n$ syntax to describe parallel updates. The update generated by this rule is the union of all the updates generated by $Rule_1 \ldots Rule_n$. A set of ASM updates is called inconsistent, if it contains updates with the same locations, i.e., two elements (loc,v) and (loc,v') with v≠v'. In the case of inconsistency, the computation does not yield a next state. For example, Figures 8(a) and 8(b) show respectively *CoreASM* code for (1) a rule (*rule1*) having a conflicting parallel function update of variable x, and (2) one block rule (*InitRule*) firing conflicting parallel updates from *rule1* and *rule2*. Both examples do not yield a next state.

Since *Spec Explorer* is based on .NET, a rule method code is not different from an ordinary sequential $C\#$ code. Hence, statements within a rule are executed in sequence,

```
// Two conflicting
// parallel updates
rule rule1 =
par
x := 1
x := 5
endpar
```

(a) Inconsistent set of function updates

```
rule rule1 = x := 1
rule rule2 = x :=5
// Conflicting call to rule1 and rule2
rule InitRule =
par
rule1
rule2
endpar
```

(b) Inconsistent set of rule updates

Figure 8. Inconsistent ASM updates

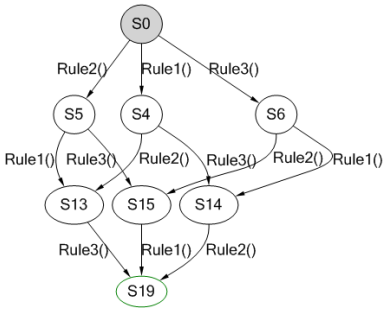
not concurrently. At every step, *Spec Explorer* enables all the rules satisfying their respective preconditions, leading to a non-deterministic interleaving of rule executions. Therefore the parallel update semantics, previously supported in *Spec Explorer 2006* [10]), is abandoned because it conflicts with the default sequential semantics of $C\#$.

If we consider rules updating different locations (i.e., consistent updates), *Spec Explorer* parallel execution of the selected rules can be reduced to a non-deterministic interleaving (with extra created transitions and states). It can be achieved using the *Cord* composition operator ||| as shown in Figure 9.

Config.cord

```
config Main
{
action abstract static void RuleAdapter.Rule1();
action abstract static void RuleAdapter.Rule2();
action abstract static void RuleAdapter.Rule3();
}
machine Interleaving(): Main
{
Rule1() ||| Rule2() ||| Rule3();
}
```

(a) SpecExplorer implementation of the block Rule



(b) Block rule associated FSM

Figure 9. Spec Explorer implementation of the block rule and its corresponding FSM

A possible work-around for detecting inconsistent updates is to design a wrapper method to check and analyze whether there are any inconsistencies between the different update statements.

7) Let Rule:: The *let* rule assigns a value of a term *t* to the variable *x* and then executes the rule *Rule*. The syntax

of a Let rule is:

let (x = t) **in** *Rule* **endlet**

The *let* rule may be implemented in *Spec Explorer* using the *LetBehavior* construct. This construct introduces a set of local variables with an optional associated constraint. The scope of the declared variables is the given behavior. Figure 10 shows an example of a *Cord* configuration relative to a machine that produces a transition of *Rule1* on *x* equal to 2.

Config.cord

```
config Main
{
action abstract static void Implementation.Rule1(int x);
}
machine LetMachine() : Main
{
  (let int x where (. x == 2 .) in Rule1(x))
}
```

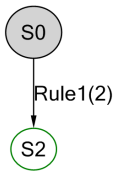


Figure 10. Spec Explorer implementation of the let rule

Other ASM rules such as *forall* rule, *iterate* rule, etc., are not covered in this work due to the lack of space.

V. Case Study: The Dining Philosophers Problem

The dining philosophers problem is a classic problem in concurrent programming invented by E. W. Dijkstra. Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table there is a bowl of rice, and the table is laid with five single chopsticks. In what follows, we map the *CoreASM* specification of the Dining Philosophers [19] into Spec Explorer.

DiningPhilosopher.casm

```
CoreASM DiningPhilosophers
use Standard
enum Chopstick = {c1, c2, c3, c4, c5}
enum Philosophers = {Albert,Herbert,Fredrich,Sina,Juan}
function eating: Philosophers -> BOOLEAN
function hungry: Philosophers -> BOOLEAN
function leftChop: Philosophers -> Chopstick
function rightChop: Philosophers -> Chopstick
function chopOwner: Chopstick -> Philosophers
init initRule
```

(a) Dining Philosophers CoreASM Declaration

DiningPhilosopher.cs

```
public enum Philosophers {Albert,Herbert,Fredrich,Sina,Juan};
public enum Chopstick { c1, c2, c3, c4, c5 };
public class DiningPhilosopher
{
  public static DiningPhilosopher I = new DiningPhilosopher();
  public static MapContainer<Philosophers, bool> Eating;
  public static MapContainer<Philosophers, bool> Hungry;
  public static MapContainer<Philosophers, Chopstick> leftChop;
  public static MapContainer<Philosophers, Chopstick> rightChop;
  public static MapContainer<Chopstick, Philosophers> ChopOwner;
//...
```

(b) Dining Philosophers Spec Explorer Declaration

Figure 11. Dining philosophers CoreASM declarations and their mappings in Spec Explorer

Figure 11 illustrates the *CoreASM* declarations and their mappings in *Spec Explorer*. Enumerations are mapped to Spec Explorer enumerations and functions are mapped to MapContainers. The creation of the set of philosophers (through agents in *CoreASM*) is done implicitly through the class constructor in SpecExplorer. In addition, there is no need to initialize *ChopOwner* as *undef* in *Spec Explorer* since its *MapContainer* is initially empty. Figure 12 illustrates the *CoreASM* init rule and its mapping as a constructor of the class DiningPhilosopher.

DiningPhilosopher.casm

```
/* ---- Initializing the Table ----- */
rule initRule = {
 forall p in Philosophers do {
   Agents(p) := true
   program(p) := @PhilosopherProgram
   eating(p) := false
   hungry(p) := false
 }
 rightChop(Albert) := c5
 leftChop(Albert) := c1
 rightChop(Herbert) := c1
 leftChop(Herbert) := c2
 rightChop(Fredrich) := c2
 leftChop(Fredrich) := c3
 rightChop(Sina) := c3
 leftChop(Sina) := c4
 rightChop(Juan) := c4
 leftChop(Juan) := c5
 /* all chopsticks are intially free */
 forall c in Chopstick do
   chopOwner(c) := undef
   print "TABLE: c1 Herbert c2 Fredrich
c3 Sina c4 Juan c5 Albert c1\n"
 Agents(self) := false
}
```

(a) CoreASM init rule

DiningPhilosopher.cs

```
public DiningPhilosopher()
{
 Hungry = new MapContainer<Philosophers, bool>();
 Eating = new MapContainer<Philosophers, bool>();
 leftChop = new MapContainer<Philosophers, Chopstick>();
 rightChop = new MapContainer<Philosophers, Chopstick>();
 ChopOwner = new MapContainer<Chopstick, Philosophers>();
 foreach (Philosophers value in Enum.GetValues(typeof(Philosophers)))
  {
    Eating.Add(value, false);
    Hungry.Add(value, false);
  }
  // Initialize leftChop
  leftChop.Add(Philosophers.Albert, Chopstick.c5);
  leftChop.Add(Philosophers.Herbert, Chopstick.c1);
  leftChop.Add(Philosophers.Fredrich, Chopstick.c2);
  leftChop.Add(Philosophers.Sina, Chopstick.c3);
  leftChop.Add(Philosophers.Juan, Chopstick.c4);
  // Initialize rightChop
  rightChop.Add(Philosophers.Albert, Chopstick.c1);
  rightChop.Add(Philosophers.Herbert, Chopstick.c2);
  rightChop.Add(Philosophers.Fredrich, Chopstick.c3);
  rightChop.Add(Philosophers.Sina, Chopstick.c4);
  rightChop.Add(Philosophers.Juan, Chopstick.c5);
}
```

(b) Spec Explorer class constructor

Figure 12. CoreASM init rule and its mapping in Spec Explorer

Figure 13 describes the CoreASM derived functions and their mappings in *Spec Explorer*.

Figure 14 shows the three rules of the system and their mappings in Spec Explorer. *CoreASM* rules *StartEating* and *StopEating* are called only from the *PhilosopherProgram* rule. In *Spec Explorer*, these two rules are enabled at every transition (see Figure 14(b)). To mimic the *CoreASM*

 ISBN: 978-1-61208-230-1

DiningPhilosopher.casm

```
/* ---- Derived Functions ----- */
derived canPickBothChopsticks =
  (chopOwner(leftChop(self)) = undef)
  and (chopOwner(rightChop(self)) = undef)
// flip of a coin
derived flip = pick b in BOOLEAN
```

(a) CoreASM Derived Functions

DiningPhilosopher.cs

```
public static bool canPickBothChopsticks(Philosophers P)
{
 if (!ChopOwner.ContainsKey(leftChop[P]) &&
!ChopOwner.ContainsKey(rightChop[P]))
   return true;
 else
   return false;
}
public static bool flip()
{
 return Choice.Some<bool>();
}
```

(b) Spec Explorer Functions

Figure 13. CoreASM derived functions and their mappings in Spec Explorer

behavior, two guard conditions *StartEatingEnabled* and *StopEatingEnabled* are added to protect the execution of rules *StartEating* and *StopEating* respectively (see Figure 14(c)).

VI. ASM to Spec Explorer Mapping Challenges and Limitations

At first glance, the mapping seemed to be simple because *Spec Explorer* supports the concept of rules with actions. Later, it appeared that it was a challenging task from many perspectives:

- The most important challenge was dealing with parallel updates/rules generating a single next step when there are no inconsistencies. Indeed, *Spec Explorer 2010* is based on .NET, so a rule method is not different from an ordinary $C\#$ sequential code. Spec Explorer 2010 [7] differs from the original *Spec Explorer 2006* [10], where ASMs were supported directly through *AsmL*. As a core difference, ASM parallel updates semantics are not directly supported in *Spec Explorer 2010* but have to be encoded, if desired.
- A related challenge to encoding parallel updates, is the detection of inconsistencies in updates. One possible work-around for this limitation is to design a wrapper method to check and analyze whether there are any inconsistencies between the different update statements.
- An ASM-based language, such as *CoreASM*, may define a program rule from which all other rules are called (see Figure 14(a)). *Spec Explorer 2010* will fire all rules whenever their preconditions are satisfied. To implement such behavior, the called rules can be guarded with entry and exit boolean conditions (as described in Figure 14(c) using variables *StartEatingEnabled* and *StopEatingEnabled*).
- The proposed mapping guidelines represent a good and direct translation of ASM constructs into *Spec Explorer*. However, different mapping may also be valid. One example is to map an ASM initialization rule as a regular rule, as an event or as a class constructor.

VII. Conclusion and Future Work

The general goal of this work is to apply advanced model-based testing techniques to Abstract State Machines. More precisely, this paper aimed to bridge the gap between ASM-based languages and the new version of *Spec Explorer*. It discussed the core aspects of ASMs and how one can map them to *Spec Explorer 2010*, where modeling is done through a special extension of $C\#$ and coordination language *Cord*. The major limitation of the proposed mappings is the implementation of ASM parallel updates semantics, which are not directly supported in *Spec Explorer 2010*. We believe, this was mainly due to the conceived ease of adoption by testers, and therefore the parallel update semantics was abandoned because it conflicts with the default sequential semantics of $C\#$.

As a future work, we are planning to extend this work to cover more ASM constructs and to investigate the automation of the proposed guidelines in a target ASM language such as *CoreASM*.

References

[1] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, 2011, published online. Paper version to appear.

[2] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: a systematic review," in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies*, ser. WEASELTech '07. New York, NY, USA: ACM, 2007, pp. 31–36.

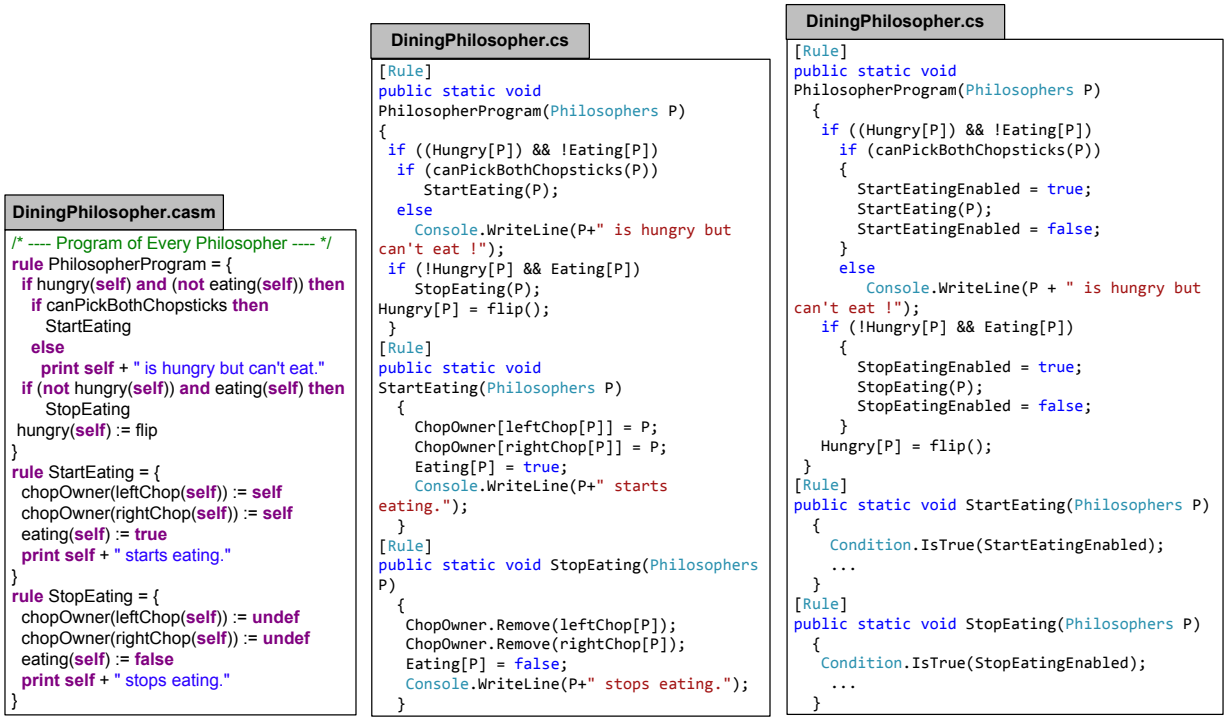
[3] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, pp. 9:1–9:76, February 2009.

[4] M. Sarma, P. V. R. Murthy, S. Jell, and A. Ulrich, "Model-based testing in industry: a case study with two mbt tools," in *Proceedings of the 5th Workshop on Automation of Software Test*, ser. AST '10. New York, NY, USA: ACM, 2010, pp. 87–90.

[5] K. Stobie, "Model based testing in practice at Microsoft," *Electron. Notes Theor. Comput. Sci.*, vol. 111, pp. 5–12, January 2005.

[6] M. Shafique and Y. Labiche, "A systematic review of model based testing tool support," Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, Tech. Rep. SCE-10-04, May 2010.

 ISBN: 978-1-61208-230-1



DiningPhilosopher.casm

```
/* ---- Program of Every Philosopher ---- */
rule PhilosopherProgram = {
  if hungry(self) and (not eating(self)) then
    if canPickBothChopsticks then
      StartEating
    else
      print self + " is hungry but can't eat."
  if (not hungry(self)) and eating(self) then
      StopEating
  hungry(self) := flip
}
rule StartEating = {
  chopOwner(leftChop(self)) := self
  chopOwner(rightChop(self)) := self
  eating(self) := true
  print self + " starts eating."
}
rule StopEating = {
  chopOwner(leftChop(self)) := undef
  chopOwner(rightChop(self)) := undef
  eating(self) := false
  print self + " stops eating."
}
```

(a) CoreASM Rules

DiningPhilosopher.cs

```
[Rule]
public static void
PhilosopherProgram(Philosophers P)
{
 if ((Hungry[P]) && !Eating[P])
  if (canPickBothChopsticks(P))
     StartEating(P);
  else
     Console.WriteLine(P+" is hungry but
can't eat !");
 if (!Hungry[P] && Eating[P])
    StopEating(P);
Hungry[P] = flip();
 }
[Rule]
public static void
StartEating(Philosophers P)
  {
    ChopOwner[leftChop[P]] = P;
    ChopOwner[rightChop[P]] = P;
    Eating[P] = true;
    Console.WriteLine(P+" starts
eating.");
  }
[Rule]
public static void StopEating(Philosophers
P)
  {
   ChopOwner.Remove(leftChop[P]);
   ChopOwner.Remove(rightChop[P]);
   Eating[P] = false;
   Console.WriteLine(P+" stops eating.");
  }
```

(b) Spec Explorer Rules

DiningPhilosopher.cs

```
[Rule]
public static void
PhilosopherProgram(Philosophers P)
  {
   if ((Hungry[P]) && !Eating[P])
     if (canPickBothChopsticks(P))
     {
       StartEatingEnabled = true;
       StartEating(P);
       StartEatingEnabled = false;
     }
     else
        Console.WriteLine(P + " is hungry but
can't eat !");
   if (!Hungry[P] && Eating[P])
     {
       StopEatingEnabled = true;
       StopEating(P);
       StopEatingEnabled = false;
     }
   Hungry[P] = flip();
 }
[Rule]
public static void StartEating(Philosophers P)
  {
    Condition.IsTrue(StartEatingEnabled);
    ...
  }
[Rule]
public static void StopEating(Philosophers P)
  {
   Condition.IsTrue(StopEatingEnabled);
    ...
  }
```

(c) Spec Explorer Rules Adjusted

Figure 14. CoreASM rules and their mappings in Spec Explorer

[7] Microsoft, "Spec Explorer 2010 Visual Studio Power Tool, Version 3.5," http://visualstudiogallery.msdn.microsoft.com/271d0904-f178-4ce9-956b-d9bfa4902745, 2010.

[8] R. Farahbod, V. Gervasi, and U. Glässer, "CoreASM: An Extensible ASM Execution Engine," *Fundamenta Informaticae*, vol. 77, pp. 71–103, January 2007.

[9] AsmL, "Microsoft Research: The Abstract State Machine Language," http://research.microsoft.com/en-us/projects/asml/, 2006.

[10] Microsoft, "Microsoft Research: Spec Explorer 2006 tool," http://research.microsoft.com/en-us/downloads/b33add8c-6172-444d-b1b1-6a91323ad7cc/default.aspx, 2006.

[11] W. Grieskamp and N. Kicillof, "A schema language for coordinating construction and composition of partial behavior descriptions," in *Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, ser. SCESM '06. New York, NY, USA: ACM, 2006, pp. 59–66.

[12] K. Winter, "Model checking for abstract state machines," *Journal of Universal Computer Science*, vol. 3, no. 5, pp. 689–701, 1997.

[13] P. Arcaini, A. Gargantini, and E. Riccobene, "AsmetaSMV: A Way to Link High-Level ASM Models to Low-Level NuSMV Specifications," in *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada*, 2010, pp. 61–74.

[14] A. Gargantini, E. Riccobene, and S. Rinzivillo, "Using Spin to generate tests from ASM specifications," in *Proceedings of the abstract state machines 10th international conference on Advances in theory and practice*, ser. ASM'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 263–277.

[15] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes, "Generating finite state machines from abstract state machines," in *Proceedings of the International Symposium on Software Testing and Analysis, July 22-24, Roma, Italy. ISSTA'02*, 2002, pp. 112–122.

[16] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes, "Towards a tool environment for model-based testing with AsmL," in *Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada*, 2003, pp. 252–266.

[17] M. Veanes, N. Bjørner, Y. Gurevich, and W. Schulte, "Symbolic bounded model checking of abstract state machines," *Int. J. Software and Informatics*, vol. 3, no. 2-3, pp. 149–170, 2009.

[18] E. Börger and R. F. Stark, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.

[19] G. Ma and R. Farahbod, "Dining Philosphers: A Sample Specification in CoreASM," 2006. [Online]. Available: http://coreasm.svn.sourceforge.net/viewvc/coreasm/engine-carma/trunk/sampleSpecs/DiningPhilosophers.coreasm

 ISBN: 978-1-61208-230-1