Towards a Tool Environment for Model-Based Testing with AsmL

Mike Barnett, Wolfgang Grieskamp, Lev Nachmanson,
Wolfram Schulte, Nikolai Tillmann, and Margus Veanes

Microsoft Research, Foundations of Software Engineering, Redmond, USA
`fse@microsoft.com`

Abstract. We present work on a tool environment for model-based testing with the Abstract State Machine Language (AsmL). Our environment supports semi-automatic parameter generation, call sequence generation and conformance testing. We outline the usage of the environment by an example, discuss its underlying technologies, and report on some applications conducted in the Microsoft environment.

1 Introduction

Over the last two decades, the area of formal software modeling has been extensively explored, developing various methods, notations and tools. Formal specification languages like VDM, Z, B, CSP, ASM etc. have been developed and applied to numerous problems. Verification technology has had success in certain areas, in particular if based on model checking. However, in spite of promising results, a widely expected break-through of these technologies has not yet appeared.

The goal of our group at Microsoft Research is to bring rigorous, formal modeling to praxis, trying to avoid (suspected) obstacles of earlier approaches to formal modeling. We have developed the Abstract State Machine Language (AsmL), an executable modeling language based on the ASM paradigm [1] and fully integrated into the .NET framework and Microsoft development environment.

One important application we see for AsmL is automated testing. A huge amount of work is spent on testing in Microsoft's and other companies' product cycle today. Models not only enhance understanding what a product is supposed to do and how its architecture is designed, but enable one to semi-automatically derive test scenarios at an early development stage where coding has not yet finished. Given manually or automatically generated test scenarios, formal models can be used to automate the test oracle. A great advantage of model-based testing is seen in its adaptability: during the product cycle, various versions of the product are published at milestones, each of which requires thorough testing. Whereas manual test suites and harnesses are hard to adapt to the variations of the product, a model makes this work easier.

We have developed an integrated tool environment for model-based testing with AsmL. This environment comprehends the following technologies:

- *Parameter generation* for providing method calls with parameter sets;
- *FSM generation* for deriving a finite state machine from a (potentially infinite) abstract state machine;
- *Sequence generation* for deriving test sequences from the FSM;
- *Runtime Verification* for testing whether an implementation performs conforming to the model.

Our environment realizes a semi-automatic approach, requiring a user to annotate models with information for generating parameters and call sequences, and to configure bindings between model and implementation for conformance testing. This annotation process is supported by a GUI. The approach is novel, to the best of our knowledge, in its combination as well as in many of its ingredients. In this paper, we will discuss the environment's methodology and underlying implementation by a walk-through of an example.

2 The Abstract State Machine Language

Space constraints prevent us from giving a systematic introduction to AsmL; instead we rely on the readers' intuitive understanding of the language as used in the examples. AsmL is a fusion of the Abstract State Machine paradigm and the .NET common language runtime type system. From a specification language viewpoint, one finds the usual concepts of earlier specification languages like VDM or Z. The language has sets, finite mappings and other high level data types with convenient and mathematically-oriented notations (e.g., comprehensions). From the .NET integration viewpoint, AsmL has all the ingredients of a .NET language, namely interfaces, structures, classes, enumerations, methods, delegates, properties and events. The close embedding into .NET allows AsmL to interoperate with any other .NET language and the framework: AsmL models can call out into frameworks and AsmL models can be called and referred to from other .NET languages, up to the level that e.g. an AsmL interface (with specification parts) can be implemented by a .NET language, enabling checking that the interface contract is obeyed [3].

The most unique feature of AsmL is its foundation on Abstract State Machines (ASM) [1]. An ASM is a state machine that in each step computes a set of *updates* of the machine's variables. Upon the completion of a step, all updates are "fired" (committed) simultaneously; until that happens, updates are not visible, supporting a side-effect free view on the computation inside a step. The computation of an update set can be complex, and the numbers of updates calculated is not statically bound. Control flow of the ASM is described in AsmL in a programmatic, textual way: there are constructs for parallel composition, sequencing of steps, non-deterministic (more exactly, random) choice, loops, and exceptions. On an exception, all updates are rolled back, enabling atomic transactions to be built from many sub-steps.

AsmL supports meta-modeling which allows a programmatic exploration of the non-determinism in the model and dealing with state as a first-class citizen (i.e., the current state is accessible as a normal value that can be manipulated just as any other data value). This allows us to realize various state exploration algorithms for AsmL models, including explicit state model-checking and in particular test generation and test evaluation.

AsmL documents are given in XML and/or in Word and can be compiled from Visual Studio .NET or from Word; the AsmL source is embedded in special tags/styles. Conversion between XML and Word (for a well-defined subset of styles) is available. This paper is itself a valid AsmL document; it is fed directly into the AsmL system for executing the formal parts it contains or for working with the AsmL test environment.

3 Example: Web Shop

Throughout this paper, we will use as an example a simplified model of a web shop. Our web shop allows clients to order gifts like flowers or perfume using the common shopping cart metaphor. Real-world details are heavily abstracted in this example to make it comprehensible (we should emphasize at this point that our approach scales to richer examples; see Sect. 0 for applications in the Microsoft environment).

The web shop's items are introduced below:

```
enum Item
  Flowers
  Perfume
const prices as Map of Item to Integer =
  { Flowers -> 30, Perfume -> 20 }
```

A *shopping cart* is represented as a bag (multi-set) of items:

```
type Cart = Bag of Item
```

A client to the web shop is described by the class below. A client has an identifier and a session state, given by its shopping cart. If the client is not in a session the cart is null (The type `T?` in AsmL denotes a type where null is an allowed value; by default, types in AsmL do not contain null):

```
class Client
  const id as String
  var cart as Cart?   = null
  override ToString() as String?
    return id
```

The state of the web shop model is given by a set of clients:

```
var clients as Set of Client = {}
```

We now define the actions of the clients. A client can be constructed in which case he is added to the set `clients;` a client can enter the shop (if he is not in a session), can add an item to his cart (if he is in a session), or remove an item (if he is in a session and the item is on his cart). Finally, a client can checkout, obtaining the bill and ending his session (In this simplified model, the client can *only* leave the shop by paying):

```
class Client
  Client(id as String)
    require not exists client in clients
            where client.id = id
    add me to clients
  EnterShop()
    require cart = null
    cart := Bag of Item()
  AddToCart(item as Item)
    require cart <> null
    cart := cart.Include(item)
  RemoveFromCart(item as Item)
    require cart <> null and then item in cart
    cart := cart.Exclude(item)
  Checkout() as Integer
    require cart <> null and then cart.Size > 0
    var bill as Integer = 0
    step foreach item in cart
      bill := bill + prices(item)
    step
      cart := null
      return bill
```

4 FSM and Sequence Generation

The AsmL tool environment allows generating a finite state machine from models such as that for the web shop. From the FSM, call sequence can be generated using standard techniques [4]. The FSM is generated by exploring the state space of the model in a similar way an explicit-state model-checker works [5]. Starting at the initial state, enabled actions are fired, leading to a set of successor states, from where the exploration is continued. An action hereby is a shared or an instance based method; parameters to this method (including the instance object if necessary) are provided by a configurable parameter generator (see Sect. 0) An action is enabled if the method's precondition (`require`) is true in the current state.

Various ways are available to prune the exploration. Pruning is strictly necessary for infinite models (like the one for the web shop where a client could add items again and again to the cart). But pruning might be also required for large finite models in order to focus on certain test purposes. The AsmL environment provides a collection of different pruning techniques; the most important are:

- *State abstraction:* state abstractions map a concrete state to an abstract state. Exploration stops when a state is reached whose abstract equivalent has already been seen.
- *Filters:* a filter allows excluding certain states from exploration; only those states that pass the filter are considered for continuation during exploration.
- *Model coverage*: a percentage of model branch coverage can be given; exploration stops when this coverage is reached.

We illustrate the FSM generation for the web shop example. First we have to provide suitable definitions for the parameter domains of the actions. The actions of interest here are the client constructor and the instance methods of a client for entering a shop, adding and removing items, and checking out. The parameters required are identifiers for clients, client objects and items. For the first one we provide a given set of names like a, b, c and so on. For the client object domain it is natural to actually use the model variable `clients` itself: it provides in each state the set of clients created so far. For the items, finally, we use the domain as given by the enumeration. We discuss the configuration of parameter domains in greater detail in the next section. For now it is important to note that the configured parameter domains can depend on the *dynamic* state of the model. Thus, as clients are created, the domain for the instance parameter of client domains, given by the model variable `clients`, grows.

Once we have configured parameter domains, we define the variables and actions of the state machine, and add a so-called abstraction property for pruning the state exploration. The state abstraction properties group the concrete states into equivalence classes; exploration is stopped if we see a concrete state for which an equivalent one has been already seen before.

Finding the right abstraction property is a creative task and requires experience and trial and error. If the purpose of the generated FSM is to create scenarios for adding and removing two different items by just one client the following property does fine:

```
property SomeItemsInCart as Set of(Bag of Item)?
    get return
      { (if client.cart <> null then
          client.cart * Bag{Flowers,Perfume}
         else null) | client in clients }
```

This property maps the state space of the model into a set of carts, for each client one cart in the set; it does not distinguish from which client the cart comes. Each cart in turn is pruned to not contain more than one `Flowers` and one `Perfume` item (we use multi-set intersection for this purpose: for example, `{a,a,b} * {a} = {a}`).

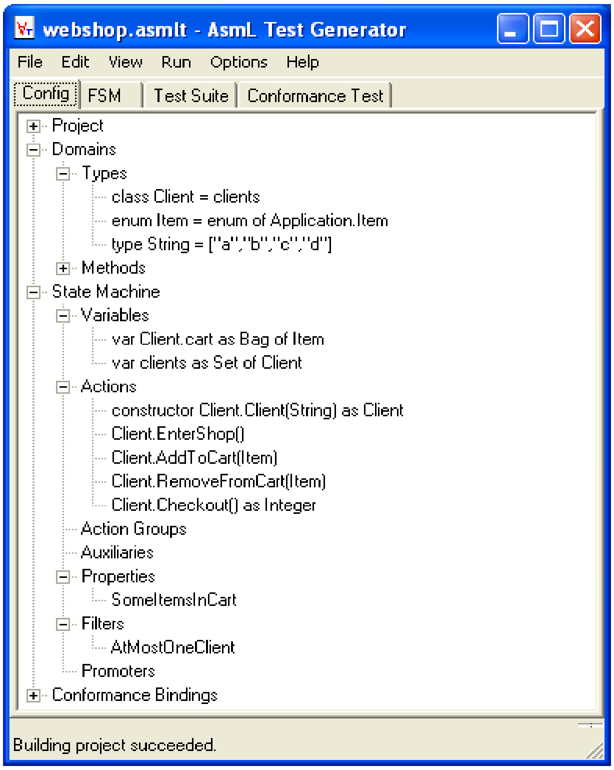
Here, we want to further prune to state space by filtering out states with more than one client. We use the following filter:

```
property AtMostOneClient as Boolean
  get return Size(clients)<=1
```

The complete configuration for the web shop is shown in the screenshot in Figure 1. The domains part of the configuration contains annotations of model elements for parameter domains. The state machine part contains annotations for variables and actions of the state machine as well as the abstraction property.

Given this configuration, we generate an FSM as shown in Figure 2. Only one client will be created in this FSM, since our abstraction property does not distinguish from which client a cart comes (and hence if a second client enters the shop, no difference is seen in the abstract state to the first client). In the FSM, S3 is associated with the state where the client's cart is empty, S4 where the client has `Perfume` on his cart, S5 where he has `Flowers` on his cart, and S6 where he has both. Among these states, various transitions exist, adding and removing items.

From an FSM as shown we generate test sequences using the well-known FSM traversal techniques (we use a variation of the transition tour method based on an algorithm from [6]). For the shown FSM we get a single traversal with 19 steps.


webshop.asmlt - AsmL Test Generator
File Edit View Run Options Help
Config FSM Test Suite Conformance Test

```
Project
Domains
  Types
    class Client = clients
    enum Item = enum of Application.Item
    type String = ["a","b","c","d"]
  Methods
State Machine
  Variables
    var Client.cart as Bag of Item
    var clients as Set of Client
  Actions
    constructor Client.Client(String) as Client
    Client.EnterShop()
    Client.AddToCart(Item)
    Client.RemoveFromCart(Item)
    Client.Checkout() as Integer
  Action Groups
  Auxiliaries
  Properties
    SomeItemsInCart
  Filters
    AtMostOneClient
  Promoters
Conformance Bindings
```

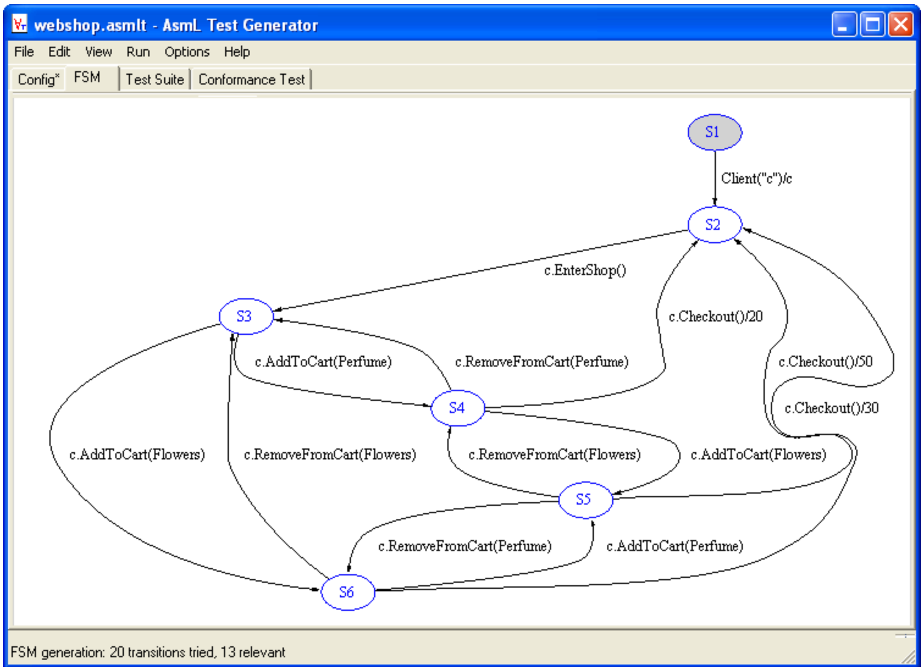
Building project succeeded.

Fig. 1. Configuration

The simple example of the web shop can produce much richer FSMs. The following property allows for more items: each client can buy up to two flowers and two perfume sets:

```
property MoreItemsInCart as Set of(Bag of Item)?
    get let maxItems = Bag{
              Flowers,Flowers,Perfume,Perfume}
        return { if client.cart <> null
                   client.cart * maxItems
                 else null
               | client in clients }
```

The FSM generated from this abstraction property and a filter that restricts the number of clients to 4 consists of around 900 relevant transitions (transitions leading to a new state under the abstraction); 6000 transitions have been tried out to find these transitions, and the construction time was around 4 minutes with a maximal memory footprint of 110 MB. Indeed, such an FSM is not feasible to visualize as a whole; however, with the methodology we are proposing one first tries out with a smaller abstract state space to understand the abstraction and then scale up parameters for the actual generated FSM and test suite.


webshop.asmlt - AsmL Test Generator
File Edit View Run Options Help
Config* FSM Test Suite Conformance Test
S1
Client("c")/c
S2
c.EnterShop()
S3
c.Checkout()/20
c.AddToCart(Perfume)
c.RemoveFromCart(Perfume)
c.Checkout()/50
S4
c.Checkout()/30
c.AddToCart(Flowers)
c.RemoveFromCart(Flowers)
c.RemoveFromCart(Flowers)
c.AddToCart(Flowers)
S5
c.RemoveFromCart(Perfume)
c.AddToCart(Perfume)
S6

FSM generation: 20 transitions tried, 13 relevant

Fig. 2. Generated FSM

5 Parameter Generation

The AsmL test environment uses a parameter generator based on access driven filtering (ADF) which is an enhancement of an existing framework called Korat [7]. ADF can generate values of recursive value types and object graphs. Given a predicate and a domain configuration, ADF generates all non-isomorphic valid inputs whereby an input is regarded as valid if the predicate holds. The domain configuration contains descriptions of finite sets as the domains of basic types, and information about how to generate objects of class types and elements of value types, and imposes bounds on the size of the generated input. The domain configuration classifies the domain of each type into one of the following three categories.

– *Defined Domain*: A defined domain is given by an arbitrary AsmL expression which is evaluated in the scope of the model. It can depend on the dynamic state of the model.
– *Inherited Domain*: An inherited domain is composed of domains as they are for other types. That is, an inherited domain just refers to one or more types, and the union of the domains of these types constitutes the inherited domain. A typical application of inherited domains is abstract types. The domain of those types is naturally the union of the domains of all of its subtypes.

- *Generated Domain*: The domain of a class or value type will be generated by ADF. ADF must be given a domain configuration for each field of the type in one of the three ways described here. A bound on the maximal number of objects/elements of the type in a single input can be imposed. Finally, each field is assigned a cost; all assignments to fields in a given input are summed up to compute the cost of this input. The predicate is configured to have a maximal cost.

ADF exhaustively finds all valid inputs which are within the bounds imposed by the domain configuration. To this end, ADF considers the parameters of the predicate and the fields of generated domains as free variables. ADF executes the predicate with an input which initially only consists of the parameters of the predicate as free variables. Whenever the execution of the predicate accesses a free variable, then ADF will instantiate this variable by choosing an object or value that is allowed by the domain configuration (thus the name access driven filtering). If the bounds imposed by the domain configuration are exceeded or the predicate returns false, then this assignment of the free variables is discarded. Otherwise, if the predicate returns true, any instantiations for the remaining free variables can be chosen to create a valid input. By exhaustively exploring all choices that are possible when instantiating free variables ADF will find all valid inputs within the given bounds.

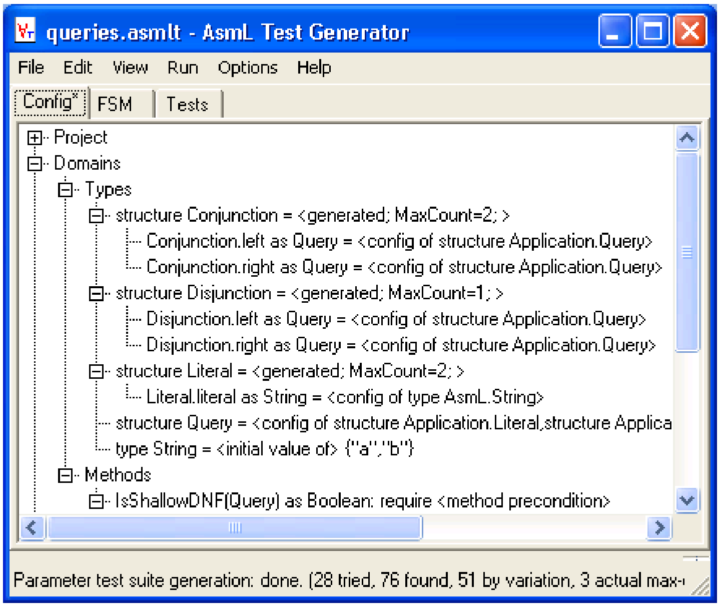
Two kinds of bounds are imposed on generated domains by the domain configuration.

- A maximal number of objects/elements of a single type: No input will contain more objects/different elements of a single type. This is an effective bound if only a small number of generated domains are involved.
- As an extension to Korat, a maximal accumulated cost along field accesses is maintained. The intuition behind this bound is that one often wants to generate asymmetric inputs which tend to be more complex only in certain areas. In this case, one would assign low costs to fields which lead to the desired complex areas, and high costs to fields which lead to areas which should not be considered. ADF stops the generation of bigger inputs when the accumulated costs exceed a given maximal cost.

As an example of ADF's usage, suppose our web shop allows inputting search queries which are simple boolean expressions over string literals. These can be defined in AsmL as below (where an AsmL structure is a value type which allows recursion):

```
abstract structure Query
structure Literal extends Query
  literal as String
structure Conjunction extends Query
  left  as Query
  right as Query
structure Disjunction extends Query
  left  as Query
  right as Query
```

Suppose we want to generate those queries as parameter inputs which are in disjunctive normal form. We define a filter predicate as below:


queries.asmlt - AsmL Test Generator
File Edit View Run Options Help
Config FSM Tests

```
Project
Domains
  Types
    structure Conjunction = <generated; MaxCount=2; >
      Conjunction.left as Query = <config of structure Application.Query>
      Conjunction.right as Query = <config of structure Application.Query>
    structure Disjunction = <generated; MaxCount=1; >
      Disjunction.left as Query = <config of structure Application.Query>
      Disjunction.right as Query = <config of structure Application.Query>
    structure Literal = <generated; MaxCount=2; >
      Literal.literal as String = <config of type AsmL.String>
    structure Query = <config of structure Application.Literal,structure Applica
    type String = <initial value of> {"a","b"}
  Methods
    IsShallowDNF(Query) as Boolean: require <method precondition>
Parameter test suite generation: done. (28 tried, 76 found, 51 by variation, 3 actual max-
```

Fig. 3. Domain Definitions

```
IsShallowDNF(query as Query) as Boolean
  match query
    q as Conjunction:
      return not q.left is Disjunction and then
             not q.right is Disjunction
    q as Disjunction:
      return IsShallowDNF(q.left) and then
             IsShallowDNF(q.right)
    q as Literal:
      return true
```

Since ADF inductively generates input by instantiating free variables in already generated input, the shallow DNF test as above is sufficient for generating a tree which is in full DNF.

Our test environment allows annotating the configuration for parameter generation with a GUI. Domains can be defined on a per-type base, per-field/parameter base, or per-method base. For the query example, the configuration is given in Figure 3.

The super-type `Query` inherits its domain from the union of the configuration of its sub-types (the full definition is not displayed because of window size). The sub-types are generated using ADF, where the recursive fields point back to the configuration of the `Query` super-type. The recursion causes no problem because the input size is bounded; we allow 2 instances for literals and conjunctions, and 1 instance for disjunction. The run of the parameter generator results in 76 parameter combinations which are of the obvious shape.

6 Conformance Testing

The AsmL test environment allows interactively configuring bindings between a model and an implementation, instrumenting the model as a test oracle. The implementation can be given as any managed .NET assembly, written in any of the .NET languages. A wizard supports the binding of model classes and methods with implementation classes and methods by signature matching.

To enable conformance testing, the implementation assemblies are rewritten on the intermediate code level inserting callbacks for monitored methods to the runtime verification engine. This engine is able to deal with non-determinism in the model by maintaining a set of admissible model behaviors. Each time a monitored method is called in the implementation, its parameters and output result will be propagated to the conformance test manager. On each of the currently possible model states, the according model method will be called. If the method contains nondeterministic behavior, several resulting states can arise. The resulting states of those calls which produce a conformant output constitute the set of next model states. If this set becomes empty, the conformance test fails. In addition to comparing just method results, a predicate which relates the model and implementation state can be employed, which may prune the state-space evolution earlier than by just observing method return values.

The problem of relating object identities is dealt with as follows. A mapping from model to associated implementation objects is maintained. Whenever a monitored implementation returns an object, the according model method's returned object must either map to exactly that object in the mapping, or no entry in the mapping exists, in which case one is created. One can think of this mechanism as letting object identities in the model being distinct logical variables which are "bound" with the associated object identities of the implementation.

7 Discussion and Conclusion

We presented aspects of a first version of an integrated environment for model-based testing with AsmL and illustrated its use by an example. The environment combines and refines the techniques for parameter generation, FSM generation, call sequence generation, and conformance testing in a novel way. We conclude with discussing applications, related work, and future work.

7.1 Applications

Though the AsmL test environment is still in a prototypical stage, it has been applied in several non-trivial projects at Microsoft.

– The parameter generator has been used for testing an implementation of the XPath language. The stateless model of XPath used for that purpose consists of around 33 pages. More than a million tests have been generated, out of which the system identified 120 test cases which already resulted in 90% model code branch cover-

age. To achieve full model code branch coverage the test engineer added 10 tests manually. The recovery of the manual test cases was easy since the system tracks branches which haven't been covered.
- The FSM and sequence generator has been used for testing web-services protocols, among them reliable messaging (RM). The model for RM consists of around 40 pages. The FSM generator produces a machine with around 1500 transitions out of 30000 possible in a couple of minutes, simulating various kinds of wire failure and recovery operations.
- Within the few months of its introduction, the AsmL test environment has gained considerable interest in the model-based testing community at Microsoft. Model-based testing using finite state machine models are in use at Microsoft for quite some time (a couple of hundred people are registered for the internal mailing list, to give an impression). The more powerful approach provided by AsmL is investigated by many of these users, and we expect a couple of new applications in the near future.

7.2 Related Work

Our approach to parameter generation is based on and extends the work found in [7], which is a later branch of the work of the authors of [8]. Whereas the authors use a Java data type to describe what they call the "finitization", we use a richer interactive method for what we call "domain configuration". Other extensions of our approach include a cost function for the generation of recursive domains and detection of isomorphisms for value types.

The conformance testing conducted by our tool environment can be classified as *grey box* testing. Traditional FSM based testing techniques with either Mealy or Moore machines typically amount to *black box* testing, where the actual states of the implementation are unobservable. In contrast, our testing approach *allows* the user to specify *conformance relations* connecting the model state to the state of the implementation, in addition to pure input/output behavior reflected at the API level. In other words, the tool may be used to perform a *limited* form of *white box* testing, where the limitations depend on what part of the implementation state is accessible (if no state is accessible, the tool works as well, but may not be able to detect errors as early as they occur.) This approach is possible due to the intermediate language platform provided by the .NET runtime, which the tool architecture is based on, that allows binary level access to the state of the implementation.

The basic FSM generation algorithm that is implemented in the test tool has been significantly extended since its first description in [5]. Test case generation that is performed on the basis of the generated FSM can be classified as a T-method [13]. We have not considered utilizing more powerful methods, such as U- D- or W-methods [13] used in pure black box testing.

One of the first automated techniques for extracting FSMs from model-based specifications for the purpose of test case generation, introduced in [9], is based on a finite partitioning of the state space of the model using full disjunctive normal forms. While our partition of the state space is similar to that of the DNF approach, the two approaches are quite different. Most importantly, the DNF approach employs symbolic techniques while we build the FSM by executing the specification. This enables us to support the full spectrum of AsmL, including call-outs from the model into framework code.

In [10] projections on state machines are used to restrict them for a certain test purpose; also filters on states are used. This is related to our pruning technique of state exploration, though we never look at the larger FSM but generate the projected one from the beginning.

In model checking, data abstraction is used to cope with state explosion when the original model M is too large. Data abstraction groups states of M and produces a reduced model M_r which is analogous to the FSM produced in our test environment by using properties. However, whereas in model checking operations need to be lifted to the abstract domain as well, which is the fundamental difficulty there, we still work with the operations on the concrete data, which can be realized using full AsmL. Due to efficiency considerations, the standard data abstraction algorithms of model-checking may yield an *over-approximation* of M_r; see [11]. In contrast, our approach may yield an *under-approximation* of the true abstraction, in other words some transitions may be missing, but there are no false transitions, which is important for using the FSM for test case generation.

In general, model checking techniques have been considered in the context of ASM based test case generation; in [12] the counter examples of SPIN are considered as test cases generated from a given ASM and a given property. The technique of using a model-checker with negated goal-states, and then letting the model-checker produce a counter-example which can be interpreted as a test sequence to reach this state, has been proposed by many other authors for automatic test generation. We believe this approach is highly restricted, on the one hand by the input language restrictions most model-checkers have to obey, on the other hand because tailored search machines for finding tests can be more efficient. For example, a model checker used to generate tests finds just one test sequence per exploration whereas our approach finds all test sequences in one exploration of the ASM.

Currently our tool supports the Rural Chinese Postman Tour method to traverse the generated FSM. For an efficient implementation of the postman tour the tool uses the algorithm for Maximal Weight Bipartite Matching given in [6]. In general, the test methodology of the tool is an extension of the FSM approach. The bulk of the work in this area has dealt with deterministic FSMs. See [4, 13] for comprehensive surveys and [14] for an overview of the literature. The *Extended* Finite State Machine (EFSM) approach has been introduced mainly to cope with the state explosion problem of the FSM approach. Typically the problem arises when the system to be modeled has variables with values in large, even infinite, domains, for example integers. In an EFSM, such variables are allowed, and the transitions may depend on and update their values; see [15] [16]. In EFSMs, the control part is finite and is separated from the data part, which distinguishes them from ASMs. An interesting problem in our FSM generation algorithm is to fiddle with the properties in order to avoid nondeterminism. This problem is related to the stabilization problem of EFSMs [16]. The use of input/output FSMs for fault coverage based test case generation is studied in

[17]. The specification FSMs in [17] are (possibly non-deterministic) Mealy machines.

Conformance testing plays a central role in testing communication protocols where it is important to have a precise model of the observable behavior of the system. This has lead to a testing theory based on labeled transition systems. See an overview of the approach in [18] and an overview of related literature in [19]. Labeled transition systems are in general nondeterministic. In the LTS approach, verification techniques can be used to deal with state explosion and to generate test cases. TGV [20] is an industrial tool that utilizes the LTS approach to generate test cases from SDL specifications. Fault model based FSM testing methodology has been recently considered for labeled transition systems as well [21,31].

There are many different groups doing work related to runtime verification. Perhaps the closest is the JML runtime assertion checking provided for components written in Java [22]. Eiffel [23] also provides for the checking of pre- and post-conditions, but only for components written in Eiffel. There are many similar design-by-contract tools for Java, such as JMSAssert [24], iContract [25], Handshake [26], Jass [27], and JContract [28]. However, all lack any facility for maintaining the state-space separation between the specification and the implementation. More general component-oriented work has been done by Edwards [29] to generate wrapper components for checking pre- and post-conditions, but cannot handle more general synchronization issues that require model programs.

7.3 Future Work

Several extensions of the AsmL test environment are on the way. High priority on our agenda is dealing with non-determinism in the model. Though we can handle non-determinism on the level of runtime verification, the test generator can not deal with it, not at least because its output, sequences, is not a suitable representation. We are looking at two different approaches. One promising approach is *on-the-fly testing* [30], which in our setting amounts to fusing the FSM generation with conformance testing. This approach has the advantage that non-determinism of the model is immediately pruned by the decisions of the implementation. However, in our experience some user groups require the tests as data in their development process. For these applications, we look at generating DAGs (directed acyclic graphs) instead of sequences. Test cases for non-deterministic systems are usually tree structures (as one form of DAG). A further topic of future work is employing symbolic computation by means of constraint resolution, lifting restrictions of our approach implied by computing with ground data.

References

1. Y. Gurevrich, "Evolving Algebras 1993: Lipari Guide," in *Specification and Validation Methods*, E. Boerger, Ed.: Oxford University Press, 1995, pp. 9–36.
2. F. o. S. Engineering, "The AsmL Release,",, 2.1.5.9 ed. Redmond: Microsoft Research, 2000–2003.

3. M. Barnett and W. Schulte, "Runtime Verification of .NET Contracts," *The Journal of Systems and Software*, pp. 199–208, 2002.
4. D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines - a survey," *Proceedings of the IEEE*, vol. 84, pp. 1090–1123, 1996.
5. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes, "Generating Finite State Machines from Abstract State Machines," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2002)*. Rome, Italy: IEEE, 2002, pp. 112–22.
6. K. Mehlhorn and G. Schaefer, "A Heuristic for Dijkstra's Algorithm with Many Targets and Its Use in Weighted Matching Algorithms," *Lecture Notes in Computer Science*, vol. 2161, pp. 242–253, 2001.
7. C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated Testing Based on Java Predicates," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*. Rome, Italy: IEEE, 2002.
8. D. Marinov and S. Khurshid, "TestEra: A Novel Framework for Automated Testing of Java Programs," presented at Proceedings of ASE-2001: The 16th IEEE Conference on Automated Software Engineering, Coronado, CA, USA, 2001.
9. J. Dick and A. Faivre, "Automating the Generation and Sequencing of Test Cases from Model-Based Specfications," in *FME '93: Industrial Strength, Formal Methods*, vol. 670, *Lecture Notes in Computer Science*, J. C. P. Woodcock and P. G. Larsen, Eds.: Springer-Verlag, 1993, pp. 268–284.
10. G. Friedman, A. Hartman, K. Nagin, and T. Shiran, "Projected State Machine Coverage for Software Testing," in *Proceedings of the ACM SIGSOFT 2002 International Symposium on Software Testing and Analysis (ISSTA-02)*, vol. 27, 4, *SOFTWARE ENGINEERING NOTES*, P. G. Frankl, Ed. New York: ACM Press, 2002, pp. 134–143.
11. E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*: MIT Press, 1999.
12. Gargantini, E. Riccobene, and S. Rinzivillo, "Using Spin to Generate tests from ASM Specifications," presented at Proc. Abstract State Machines 2003, 2003.
13. D. P. Sidhu and T.-K. Leung, "Formal Methods for Protocol Testing: A Detailed Study," *IEEE Transactions on Software Engineering*, vol. 15, pp. 413–426, 1989.
14. Petrenko, "Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography," *Lecture Notes in Computer Science*, vol. 2067, pp. 196–?, 2001.
15. Bourhr, R. Dssouli, and E. M. Aboulhamid, "Automatic test generation for EFSM-based systems," Publication departementale 1043, Departement IRO, Universite de Montreal August 1996 1996.
16. K.-T. Cheng and A. S. Krishnakumar, "Automatic generation of functional vectors using the extended finite state machine model," *ACM Transactions on Design Automation of Electronic Systems.*, vol. 1, pp. 57–79, 1996.
17. Petrenko, N. Yevtushenko, and G. v. Bochmann., "Testing deterministic implementations from nondeterministic FSM specifications," presented at Proc. IFIP TC6 9th International Workshop on Testing of Communicating Systems, Darmstadt, Germany, 1996.
18. J. Tretmans and A. Belinfante, "Automatic testing with formal methods," presented at EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review, Barcelona, Spain, 1999.
19. E. Brinksma and J. Tretmans, "Testing Transition Systems: An Annotated Bibliography," *Lecture Notes in Computer Science*, vol. 2067, pp. 187–?, 2001.
20. J. C. Fernandez, C. Jard, T. Jéron, and C. Viho, "An experiment in automatic generation of test suites for protocols with verification technology," *Science of Computer Programming - Special Issue on COST247, Verification and Validation Methods for Formal Descriptions*, vol. 29, pp. 123–146, 1997.

21. Petrenko, N. Yevtushenko, and J. L. Huo, "Testing Transition Systems with Input and Output Testers," presented at Proc. IFIP TC6 9th International Workshop on Testing of Communicating Systems, Sophia Antipolis, France, 2003.
22. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, "How the design of JML accomodates both runtime assertion checking and formal verification," Technical Report TR #03-04 Department of Computer Science, Iowa State University March 2003 2003.
23. B. Meyer, *Eiffel: The Language*. New York, NY: Prentice Hall, 1992.
24. M. M. Systems, "JMSAssert,"., 2002.
25. R. Kramer, "iContract -- the Java Designs by Contract tool. In Proc. Technology of Object-Oriented Languages and Systems," presented at Proc. Technology of Object-Oriented Languages and Systems, TOOLS 26, Santa Barbara, CA, 1998.
26. Duncan and U. Hoelzle, "Adding Contracts to Java with Handshake," University of California, Santa Barbara. Computer Science., Technical Report TRCS98-32, December 9, 1998 1998.
27. D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim, "Jass - Java with assertions," in Workshop on Runtime Verification held in conjunction with the 13th onference on Computer Aided Verification, CAV'01, 2001.
28. P. Corporation, "Using design by contract to automate Java software and component testing,", February 2003 2003.
29. S. H. Edwards, "A framework for practical, automated black-box testing of component-based software," *Software Testing, Verification and Reliability*, vol. 11, 2001.
30. R. G. d. Vries and J. Tretmans, "On-the-fly conformance testing using Spin," *Software Tools for Technology Transfer*, vol. 2, pp. 382–393, 2000.
31. Q.M. Tan and A. Petrenko, "Test Generation for Specifications Modeled by Input/Output Automata", in Proc. IFIP 11th Int. Conf. on Testing of Communicating Systems, IWTCS'98, Russia, 1998.