Model-Based Testing with AsmL .NET

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

Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
{mbarnett,wrwg,levnach,schulte,nikolait,margus}@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] which is fully integrated in the .NET framework and Microsoft development tools. One important application we see for AsmL is automated testing. A huge amount of work is spent for testing in Microsoft's 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 thoroughly 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 (potential infinite) abstract state machine;
- *Sequence generation* for deriving test sequences from the FSM;
- *Runtime Verification* for testing whether an implementation performs confirming 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. In this paper, we will discuss the environments methodology and underlying implementation by a walkthrough to an example. A focus is put on the FSM generation.

2 The Abstract State Machine Language AsmL

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[1]. AsmL is a fusion of the Abstract

[1] At the time of this writing, there is no publication about the AsmL language available. However, the AsmL distribution [2] F. o. S. Engineering, "The AsmL Release,",, 2.1.5.9 ed. Redmond: Microsoft Research, 2000-2003.

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 callout into frameworks and AsmL models can be called and referred 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. 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 little 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. 6 for applications in the Microsoft environment).

The web shop's items and prices for items are introduced below. A *shopping cart* is a bag (multi-set) of items:

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

```
class Client
  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:

contains a tutorial and reference.

```
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 has no active session), can add an item to his cart (if he has a session), or remove an item (if he has a session and the item is on its cart). Finally, a client can checkout, obtaining the bill and ending his session:

```
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 Generating a FSM for the Web Shop

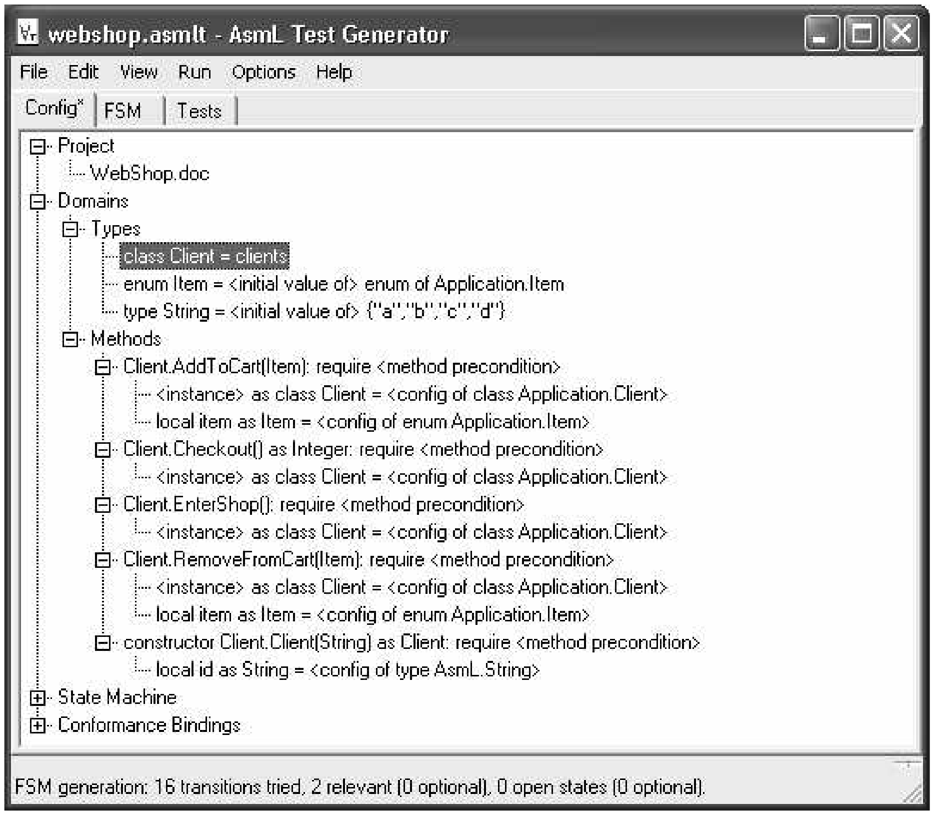
The AsmL tool environment allows generating a finite state machine from models like given for the web shop (and much larger models, indeed). The FSM is generated by exploring the state space of the model in a similar way an explicit-state model-checker works. Starting at the initial state, enabled actions are fired, leading to a set of successor states, from where the exploration is continued. An action hereby is a shared or instance based method; parameters to this method (including the instance object if necessary) are provided by a configurable parameter generator. An action is enabled if the methods 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. The AsmL environment provides a collection of different pruning techniques; the most important are:

- *State abstraction:* state abstractions map a concrete state in an abstract state. Exploration stops when a state is reached which abstract equivalent has been already 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. For the items, finally, we use the domain as given by the enumeration.

Our tool environment can take arbitrary expressions for parameter domains, in particular expressions which depend on the model state itself. The domains can be added as an annotation to the model. The screen shot below shows the domain configuration:


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

```
⊟ Project
    WebShop.doc
⊟ Domains
    ⊟ Types
        class Client = clients
        enum Item = <initial value of> enum of Application.Item
        type String = <initial value of> {"a","b","c","d"}
    ⊟ Methods
        ⊟ Client.AddToCart(Item): require <method precondition>
            <instance> as class Client = <config of class Application.Client>
            local item as Item = <config of enum Application.Item>
        ⊟ Client.Checkout() as Integer: require <method precondition>
            <instance> as class Client = <config of class Application.Client>
        ⊟ Client.EnterShop(): require <method precondition>
            <instance> as class Client = <config of class Application.Client>
        ⊟ Client.RemoveFromCart(Item): require <method precondition>
            <instance> as class Client = <config of class Application.Client>
            local item as Item = <config of enum Application.Item>
        ⊟ constructor Client.Client(String) as Client: require <method precondition>
            local id as String = <config of type AsmL.String>
⊞ State Machine
⊞ Conformance Bindings
```

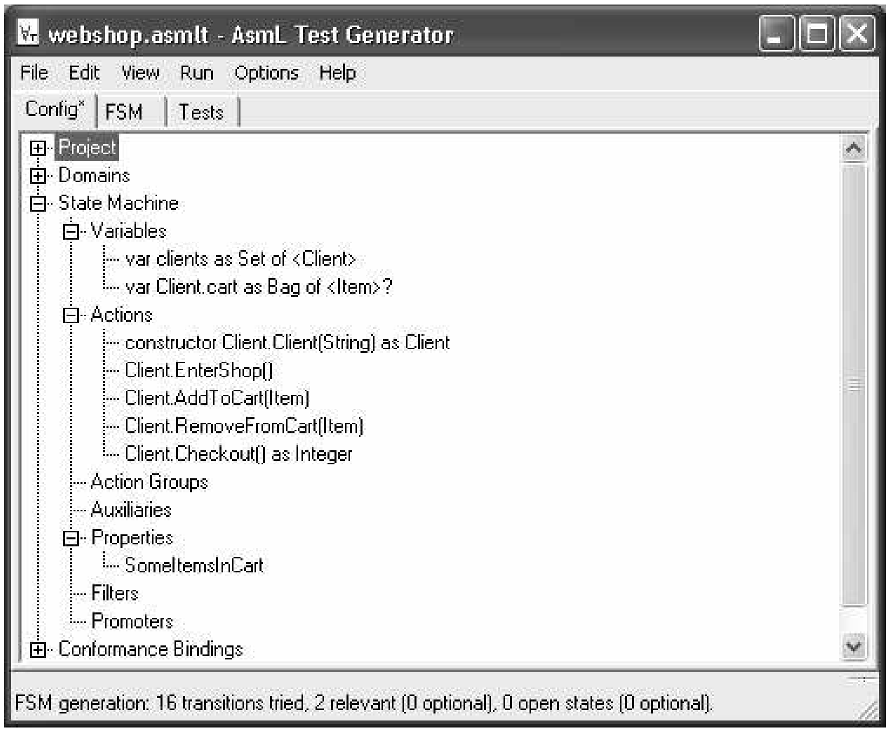
FSM generation: 16 transitions tried, 2 relevant (0 optional), 0 open states (0 optional).

Note that we can configure domains on a per-type and on a per-parameter base. For the example we only used per-type configuration, and the configuration of the parameters point to those of their according type. The parameter generator provides quite a bit more functionality, like automatically generating complex data structures [4], which are out of the scope of this abstract.

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. Finding the right abstraction property is a creative task and requires experience and trial and error. If the purpose of the generated FSM should be 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 { client.cart * Bag of Item([Flowers,Perfume])
             | client in clients where client.cart ne null }
```

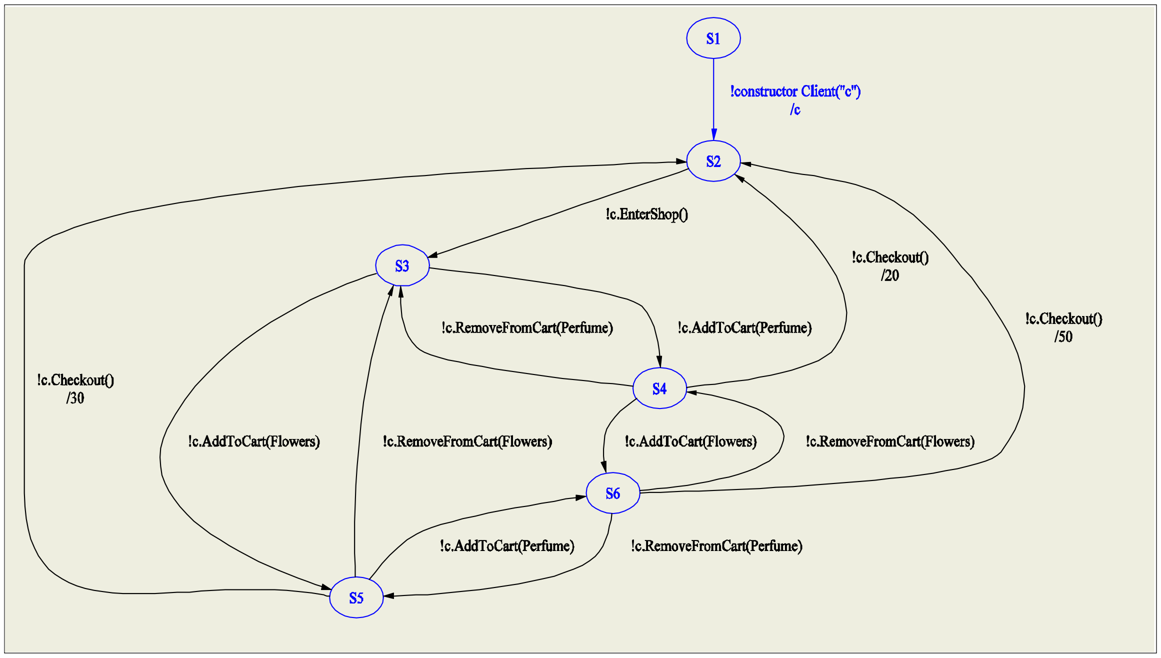
This property maps the state space of the model into a set of carts, for each client one cart in the set; it does not distinguish from which client the cart comes. Each cart in turn is pruned to not contain more than one `Flowers` and one `Perfume` item (we use multi-set intersection for this purpose: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



```
⊞ Project
⊞ Domains
⊟ State Machine
    ⊟ Variables
        var clients as Set of <Client>
        var Client.cart as Bag of <Item>?
    ⊟ 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
      Promoters
⊞ Conformance Bindings
```

FSM generation: 16 transitions tried, 2 relevant (0 optional), 0 open states (0 optional).

Given this configuration, we generate an FSM as depicted below. Only one client will be created in this FSM, since our abstraction property does not distinguishes from which client a cart comes (and hence if a second client enters the shop, this looks not different than for the first client). Below, S3 is associated with the state where the clients cart is empty, S4 where the client has `Perfume` on his cart, S5 where he has `Flowers` on his cart, and state S6 where he has both. Among these states, various transitions exist, adding and removing items.



From an FSM as shown above we generate test sequences using the well-known traversal techniques (we use a variation of [5]). For the shown FSM we get a single traversal with 19 steps.

The simple example of the web shop can produce much richer FSMs. The following property makes a distinction of clients by delivering a sequence of carts instead of a set; the sequence is pruned to length 3 such that not more than 3 clients will be represented in the FSM. Each client can buy up to two flowers and two perfume sets:

```
property MoreItemsInCart as Seq of (Bag of Item)?
  get return Subseq([ if client.cart <> null
                        client.cart * Bag of Item([Flowers,Flowers,Perfume,Perfume])
                      else null
                    | client in clients ],
                    0,3)
```

The FSM generated from this abstraction property consists of around 2800 relevant transitions (transitions leading to a new state under the abstraction); 12000 transitions have been tried out to find these transitions, and the construction time was 6 minutes with a maximal memory footprint of 128 MB. Indeed, such an FSM is not feasible to visualize as a whole; however, with the methodology we envisage 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.

5 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 test manager. On each of the currently possible model states, the according model method will be called. 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 by 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.

6 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.

6.1 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. About 120 test cases have been generated from the model resulting in around 90% code coverage; about 10 test cases had to be manually defined to achieve full coverage. The recovery of the manual test cases was easy once knowing the non-covered code.

- 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.

6.2 Related Work

Our approach to parameter generation is based on and extends the work found in [4]. 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 basic FSM generation algorithm that is implemented in the test environment is described in [6]. One of the first automated techniques for extracting FSMs from model-based specifications for the purpose of test case generation, introduced in [7], 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 spec.

In [8] projections on state machines are used to restrict them for a certain test purpose; also filters on states are used. Our pruning technique of state exploration is related, 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 here, we still work with the operations on the concrete data. Due to efficiency considerations, the standard data abstraction algorithms of model-checking may yield an *over-approximation* of M_r; see [9]. 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 [10] the counter examples of SPIN are considered as test cases generated from a given ASM and a given property. The approach in [10] does not consider approximations. The model checking approaches for test generation are inherently restricted by what parts of the modeling language can be mapped into the used model checker; our approach has no restrictions at all regarding the available modeling constructs.

6.3 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* [11], 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 to see the tests as data in their process. For these applications, we look at generating DAGs (directed acyclic graphs) instead of sequences. 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. Gurevich, "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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*: MIT Press, 1999.

[10] A. Gargantini, E. Riccobene, and S. Rinzivillo, "Using Spin to Generate tests from ASM Specifications," presented at Proc. Abstract State Machines 2003, 2003.

[11] 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.