Scenario-oriented Modeling in AsmL and its Instrumentation for Testing

Mike Barnett, Wolfgang Grieskamp, Yuri Gurevich,
Wolfram Schulte, Nikolai Tillmann and Margus Veanes
Microsoft Research, Foundations of Software Engineering, REDMOND
wrwg@microsoft.com

Abstract

We present an approach for modeling use cases and scenarios in the Abstract State Machine Language and discuss how to use such models for validation and verification purposes.

1 Introduction

The Abstract State Machine Language (AsmL) is a novel, executable modeling language which is fully integrated in the .NET framework and Microsoft development tools. This position paper shows how AsmL can be used in a natural way for scenario-oriented modeling and how these models can be used for validation and verification. The paper refines and extends earlier work on AsmL and use cases [1].

The paper uses a fragment of the CTAS case study to demonstrate the approach. CTAS has been suggested by the organizers of the SCESM workshop as a common case study; the fragment looked at is the weather control logic of a traffic flight control system. We will show in particular how the problem of dealing with an arbitrary number of clients to the connection manager of the weather control logic can be adequately described with our approach.

The paper starts with a sketch of AsmL and then introduces the pattern for scenario-based modeling in AsmL. The major part of the paper is dedicated to the CTAS case study; we presume that the reader is familiar with the basic setup. We conclude with sketching how test cases can be derived from the model, using the AsmL System [2].

2 A Glimpse of AsmL

The space constraints prevent us from giving a systematic introduction into AsmL; instead we rely on the readers' intuitive understanding of the language as used in the examples. Conceptually, AsmL is a fusion of the Abstract State Machine paradigm and the .NET common language runtime type system. One finds the usual concepts of earlier modeling languages like VDM or Z. AsmL has sets, finite mappings and other high level data types with convenient and mathematically-oriented notations (e.g., comprehensions); it uses ASM update semantics and atomic transactions for dealing with state; and it has all the ingredients of a .NET language such as interfaces, structures, classes, enumerations, methods, delegates, properties, and events. The close embedding into .NET allows AsmL to interoperate with any other .NET language, and makes it a perfect choice for modeling under .NET.

The most unique feature of AsmL is its foundation on Abstract State Machines (ASMs) [3]. An ASM is a state machine which in each step computes a set of *updates* of the machines variables. Upon the completion of a step, all updates are "fired" (committed) simultaneously. The update semantics of AsmL is based on the theory of partial updates [6,7]. The computation of an update set can be complex, and the numbers of updates calculated may depend on the current state. Control flow of the ASM is described in AsmL in a programmatic, textual way: there are constructs for sequencing of steps, non-deterministic (more exactly, random) choice, loops, and exceptions. On an exception, all updates on state are unrolled, enabling atomic transactions to be built from many sub-steps.

AsmL supports meta-modeling based on reflection which allows a systematic exploration of the non-determinism in the model. On the meta-level, the state is a first-class citizen, which enables us to realize various search strategies over the state space of a model. This is important for the instrumentation of an AsmL model for test generation and the use as a test oracle.

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. Note that this paper is itself a valid AsmL document; it is fed directly into the AsmL system for animating the formal parts it contains or for working with the AsmL test generation tool.

3 Scenarios in AsmL

We consider a *use case* to be a set of *scenarios*; each scenario describes a sequence of *events*. As in [1], we do not explicitly attach actors and roles to the events, but regard this as an extra level of methodology which can be expressed for a particular model if required. Our goal is to describe scenarios programmatically by using the sequence notation of AsmL, as in:

```
  step DO( Event1 )
  step DO( Event2 )
```

Here **step** is a keyword introducing the next step of the abstract state machine in a sequence, and `DO` is a helper method which appends an event to the sequence of events associated with this scenario.

We collect the required auxiliary definitions in an abstract class `UseCase` which is extended for a concrete use case. This class provides an abstract .NET property defining the set of scenarios as defined by a sub-class, and contains an ASM variable holding a sequence of events. The `DO` helper method appends to this sequence. If a scenario is "played" within the use case, we can think of this variable holding what has been "played" so far.

An event is described by an interface which just serves as a type tag on this level. A scenario is represented as a .NET delegate (similar to a first-class method). When executed it is supposed to record its events by calling `DO`. Playing the entire use case means, for some given number of iterations, non-deterministically choosing one of the enabled scenarios (i.e., its precondition holds), and finally returning the sequence of events produced:

```
abstract class UseCase
  interface Event
  delegate Scenario()
  abstract property Scenarios as Set of Scenario
    get // read-only property
  var events as Seq of Event = []
  DO(evnt as Event)          // fire event
    events := events + [evnt]
  Play(iters as Integer) as Seq of Event
    if iters = 0
      return events
    else
      var cands = Scenarios
      var found = false
      step while not found and cands <> {}
        choose scenario in cands
          try
            scenario() // calling the delegate
            found := true
          catch
            AssertionFailedException:
              // failed == scenario not enabled
              remove scenario from cands
      step
        if found
          return Play(iters-1)
        else // no scenario enabled
          return events
```

To give life to these definitions, let us consider a tiny example for a keycard controlled door. The use case for this defines structures (value types in AsmL) for the actions of the door and of the user, and gives scenarios for the normal behavior (the keycard is valid) and for the error behavior. Note that the "case" notation below is a convenient way to extend the enclosing class/structure in AsmL's OO type system, and corresponds to the sum-of-products or "free algebraic type" construct in other languages:

```
class KeycardControlledDoor extends UseCase
  structure DoorEvent implements Event
    case WaitForCard
    case ReleaseLock
    case SignalInvalidCard
  structure UserEvent implements Event
    case SwipeCard
  NormalScenario()
    step DO( DoorEvent.WaitForCard )
    step DO( UserEvent.SwipeCard )
    step DO( DoorEvent.ReleaseLock )
  InvalidCardScenario()
    step DO( DoorEvent.WaitForCard )
    step DO( UserEvent.SwipeCard )
    step DO( DoorEvent.SignalInvalidCard )
  override property Scenarios as Set of Scenario
    get
      return {new Scenario(NormalScenario),
              new Scenario(InvalidCardScenario)}
```

So far, we have approached a problem-oriented notation for scenarios in AsmL. The scenarios are type-checked and can be executed: evaluating the expression `new KeycardControlledDoor().Play(3)`[1] will result in a sequence of events, and due to the non-deterministic choice of the scenario, different ones over time. With the powerful meta-modeling facilities of AsmL we can actually do more via execution. Before we come to that, we will explore how the approach can be used for the CTAS case study.

4 The CTAS Weather Control Logic

The CTAS weather control logic is suggested by the organizers of the SCESM 2003 as a common case study. CTAS (Center TRACON Automation System) is a set of tools designed to help air traffic controllers. CTAS consists of a set of processes with one of them acting as the connection manager (CM) to which the other processes are clients. One task in the CTAS system is to synchronize weather information between a weather forecast provider and the variety of clients, which is safety critical since adverse weather conditions can grind an entire traffic control system to a halt. The weather control logic is given as a "real world" informal specification consisting of a set of axioms and scenarios. Here, we will model a fragment of the logic, more specifically, the updating of the weather information between the CM and its clients. The interesting aspect of the update phase is that it has to guarantee atomicity: it becomes effective only if all clients successfully receive the new weather information.

Our approach to scenarios in AsmL allows us a nearly one-to-one translation from the original spec. We start with modeling some data domains. The (simplified)

[1] Note that you can directly evaluate the expression from this document under Word XP by highlighting it and selecting the Quick Watch function of the AsmL tool bar. To that end, you will need to edit the configuration file and change the target to "library" and the output file name to end with ".dll".

STATUS of the CM as well of its clients is described by an enumeration distinguishing the states pre-updating, updating, post-updating, post-reverting, and done (idle). One interesting aspect of this example is that we deal with a variable number of clients; each client (CL) is identified by a unique CLIENTID, which is a number. We define structures describing the events (messages) of the client, of the connection manager, and environment related ones:

```
class CTASWeatherControl extends UseCase
  enum STATUS
    PREUPDATING
    UPDATING
    POSTUPDATING
    POSTREVERTING
    DONE
  type CLIENTID = Integer
  structure ENV implements Event
    case NEW_FORECAST
  structure CM implements Event
    destination as CLIENTID
    case CLOSE_CONNECTION
    case GET_NEW_WEATHER
    case USE_NEW_WEATHER
    case REVERT_WEATHER
  structure CL implements Event
    source as CLIENTID
    case CONNECT
    case RECEIVED_GET
      sucess as Boolean
    case RECEIVED_USE
      success as Boolean
    case RECEIVED_REVERT
      success as Boolean
```

To represent a connection with a client, we extend the class CTASWeatherControl with a socket class which holds the id of the client and its status:

```
class CTASWeatherControl
  class SOCKET
    id         as CLIENTID
    var status as STATUS
    public override ToString() as String?
      return "#" + id
```

We can now define the data state of the use case. It consists of the current cycle status and a set of sockets representing the clients:

```
class CTASWeatherControl
  var status as STATUS = DONE
  var sockets as Set of SOCKET = {}
```

We start with a scenario for a client connecting with the CM. This scenario is parameterized over the client's id. (We will see later how to deal with this when registering the scenarios with the use case.) When the client connects, a new socket is created and the client's and CM's cycle status is set to DONE. (Note that in the original spec, we have an initialization protocol for the new client, which we skip here to save space.) We use the **require** construct of AsmL to ensure that a client connect can only happen when the CM is in cycle status DONE:

```
class CTASWeatherControl
  ConnectClient(id as CLIENTID)
    require status = DONE and
            not exists s in sockets
                where s.id = id
    DO( CL.CONNECT(id) )
    let s = new SOCKET(id,DONE)
    add s to sockets
```

The next scenario describes the situation where the CM enters the update weather information phase. This is triggered by the event ENV.NEW_FORECAST. The CM will send out a message to each client to get the new weather information; in reality, the message carries the weather information, which we omit here:

```
class CTASWeatherControl
  NewForecast()
    require status = DONE
    step DO( ENV.NEW_FORECAST )
         status := UPDATING
    step foreach s in sockets
           DO( CM.GET_NEW_WEATHER(s.id) )
           s.status := UPDATING
```

The next scenario handles incoming CL.RECEIVED_GET responses from the clients. It is parameterized over the client's socket and a boolean flag indicating whether the client has successfully received the new weather. It is enabled only if both the CM and the given client are in the status updating. If the client has successfully received the weather, its status is changed to post-updating. If the client failed, then the CM switches into status post-reverting and all clients are sent messages to revert:

```
class CTASWeatherControl
  ReceivedGet(s as SOCKET, success as Boolean)
    require status = UPDATING and
            s.status = UPDATING
    step DO( CL.RECEIVED_GET(s.id,success) )
    step if success
           s.status := POSTUPDATING
         else
           status := POSTREVERTING
           step foreach s' in sockets
                  DO( CM.REVERT_WEATHER(s'.id) )
                  s'.status := POSTREVERTING
```

The next scenario describes what to do when the CM is in status updating and all clients have successfully received the new weather information, i.e. are in state post-updating. The CM sends a message to all clients to actually use the new data:

```
class CTASWeatherControl
  AllReceivedGet()
    require status = UPDATING and
            (forall s in sockets
             holds s.status = POSTUPDATING)
    status := POSTUPDATING
    step foreach s in sockets
           DO( CM.USE_NEW_WEATHER(s.id) )
```

The next scenario describes incoming CL.RECEIVED_USE responses from the clients and is similar to the scenario ReceivedGet. However, if in

this state any of the clients fail when using the new weather, the system essentially resets, disconnecting all clients:

```
class CTASWeatherControl
  ReceivedUse(s as SOCKET, success as Boolean)
    require status = POSTUPDATING and
            s.status = POSTUPDATING
    step DO( CL.RECEIVED_USE(s.id,success) )
    step if success
           s.status := DONE
         else
           status := DONE
           step foreach s' in sockets
                  DO(CM.CLOSE_CONNECTION(s'.id))
                  remove s' from sockets
```

The next scenario describes the situation where all clients have successfully acknowledged usage of the new weather info. The CM returns to status `DONE`. In reality, more things happen (like logging the new weather info to a file) which we omit here:

```
class CTASWeatherControl
  AllReceivedUse()
    require status = POSTUPDATING and
     (forall s in sockets holds s.status = DONE)
    status := DONE
```

We finally need to model the reverting phase, which happens when any of the clients fail to get the new weather data:

```
class CTASWeatherControl
  ReceivedRevert(s as SOCKET,success as Boolean)
    require status = POSTREVERTING and
            s.status = POSTREVERTING
    step DO( CL.RECEIVED_REVERT(s.id,success) )
    step if success
           s.status := DONE
         else
           status := DONE
           step foreach s' in sockets
                  DO(CM.CLOSE_CONNECTION(s'.id))
                  remove s' from sockets
  AllReceivedRevert()
    require status = POSTREVERTING and
     (forall s in sockets holds s.status = DONE)
    status := DONE
```

To set up the use case, we need to collect the set of scenarios. For each of the parameterized scenarios, we define a non-parameterized version which makes a choice selecting parameters:

```
class CTASWeatherControl
  ConnectClientChoice()
    choose id in {1,2,3}
      ConnectClient(id)
  ReceivedGetChoice()
    choose s in sockets, x in enum of Boolean
      ReceivedGet(s,x)
  ReceivedUseChoice()
    choose s in sockets, x in enum of Boolean
      ReceivedUse(s,x)
  ReceivedRevertChoice()
    choose s in sockets, x in enum of Boolean
      ReceivedRevert(s,x)
```

Now, we can override the property of the `UseCase` class which defines the scenarios:

```
class CTASWeatherControl
  override property Scenarios as Set of Scenario
    get
      return {
        new Scenario(ConnectClientChoice),
        new Scenario(NewForecast),
        new Scenario(ReceivedGetChoice),
        new Scenario(AllReceivedGet),
        new Scenario(ReceivedUseChoice),
        new Scenario(AllReceivedUse),
        new Scenario(ReceivedRevertChoice),
        new Scenario(AllReceivedRevert)}
```

This finishes the model. We can now "play" the use case: `new CTASWeatherControl().Play(11),` for example, produces for instance the following sequence:

```
[CL.CONNECT(client=2),
 CL.CONNECT(client=1),
 ENV.NEW_FORECAST(),
 CM.GET_NEW_WEATHER(client=2),
 CM.GET_NEW_WEATHER(client=1),
 CL.RECEIVED_GET(client=1,sucess=True),
 CL.RECEIVED_GET(client=2,sucess=False),
 CM.REVERT_WEATHER(client=1),
 CM.REVERT_WEATHER(client=2),
 CL.RECEIVED_REVERT(client=2,success=True),
 CL.RECEIVED_REVERT(client=1,success=False),
 CM.CLOSE_CONNECTION(client=1),
 CM.CLOSE_CONNECTION(client=2)]
```

Each additional execution would produce new variations of the choices of parameterized scenarios.

5 Generating Tests

The AsmL Test Tool generates test sequences from a scenario-oriented model like the one we gave for CTAS (and also from other kinds of AsmL specifications). Some concepts of the tool have been explored in [4]; the tool and its documentation are available as part of the AsmL distribution. Currently the test tool consists of the following components:

1. *Parameter Generator*. The generator is configured with a hierarchy of domain definitions. Definitions are given on type level, on method level, and on parameter level. Defaults are taken from a higher level if no definition is given on the lower level; for example, if a definition is not given for a parameter, it is taken from the type of the parameter. For the CTAS example, we just need to configure the type `CLIENTID` to take values from some final set like `{1,2}`, the type `SOCKET` to take elements from the current value of the `sockets` variable, and the type Boolean to take values from `{true,false}`.
2. *FSM Generator*. The finite state machine generator is configured with state variables and actions. For the CTAS, we take as state variables the `sockets` and `status` fields and as actions the scenarios (we can directly use the parameterized scenarios here). Using the parameter generator to find parameters for actions, the FSM generator exhaustively explores the

model's state space, by applying enabled actions starting at the initial state. This is a breadth-first exploration implemented on the meta-modeling level. The state-space exploration is terminated using several complementary techniques. *State abstraction properties* group states into equivalence classes; the number of visits to an equivalence class is bounded. *Filters* can be installed which stop the exploration at certain states. *Bounds,* for example the percentage of model path coverage, also terminate the exploration.

3. *Sequence Generator*. The sequence generator takes the FSM and applies known algorithms to find the least number of sequences covering all links of the generated FSM. In the CTAS case, these will be sequences of scenarios; the sequence of events can be easily extracted from each scenario sequence.
4. *Conformance Tester*. The conformance tester executes the model and an implementation together, ideally in lock step [5]. The conformance tester reads an arbitrary managed .NET assembly, binds methods of the model against the implementation, and executes the test sequences running the model and implementation. We have applied the conformance tester to various API specs. In the case of scenario-oriented specifications, the gap between requirements and design is larger but it can be bridged by providing more sophisticated binding code between the events of the scenarios and those of an underlying implementation.

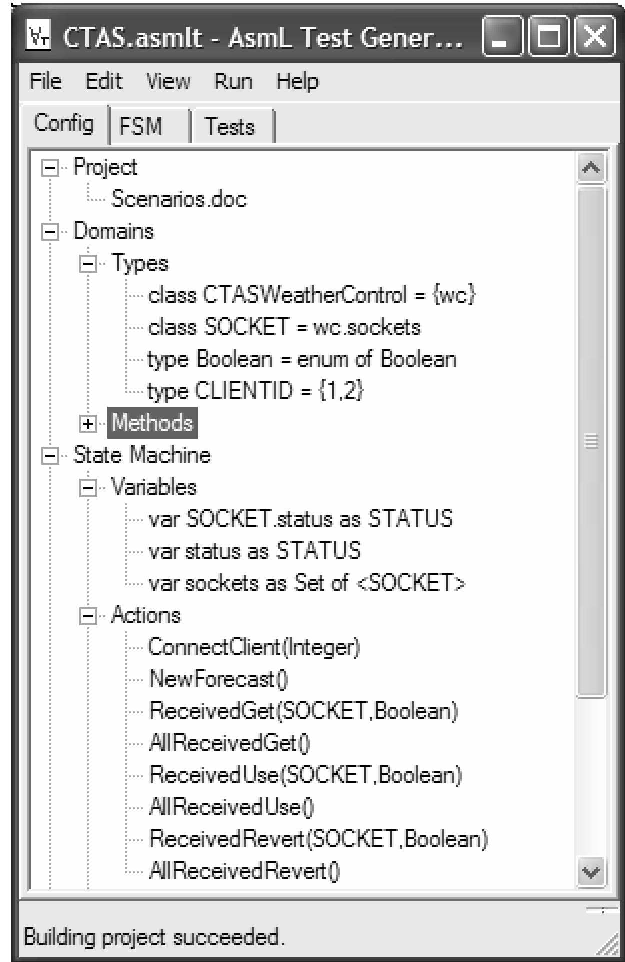
We describe how an FSM is generated for the CTAS weather control with the AsmL Test Tool. We need to give the tool an instance of the weather control use case, which is defined in the following paragraph. We also override the string conversion function for nicer output:

```
wc = new CTASWeatherControl()
class CTASWeatherControl
  override ToString() as String?
    return "C"
```

When the test tool starts up, we enter some basic configuration information:

1. The source of the model (this document);
2. The relevant state variables of the model (we choose that weather control cycle status, socket set, and individual socket);
3. The actions of the model (we choose all the original scenarios including parameterized ones);
4. Once we have added the actions, the tool will come up with defaults of the so-called *domain* configuration, which defines how parameters are generated. We assign to type `CTASWeatherControl` the singleton set containing the instance `wc`, to the type socket the value of `wc.sockets` as dynamically defined in the current state, to the type Boolean the set `{true,false}`, and to the type `CLIENTID` the set `{1,2}`.

A screenshot of the configuration after this input will look as shown below:



The most challenging step in configuring for test generation is to define when the exploration of the state space terminates. Recall that the test tool generates from the configured ASM an FSM by exhaustively executing the model, applying actions with supplied parameters starting from an initial state in a breadth-first search. Even if the state space is finite, the search space might be huge. The AsmL test tool supplies a variety of ways to control the search:

- *State Abstraction*. The user can supply a mapping from the concrete state into a more abstract state. This mapping serves to define *equivalence classes* between states. When state is visited which have been already seen in the abstract state space (which is in the same equivalence class) exploration is terminated.

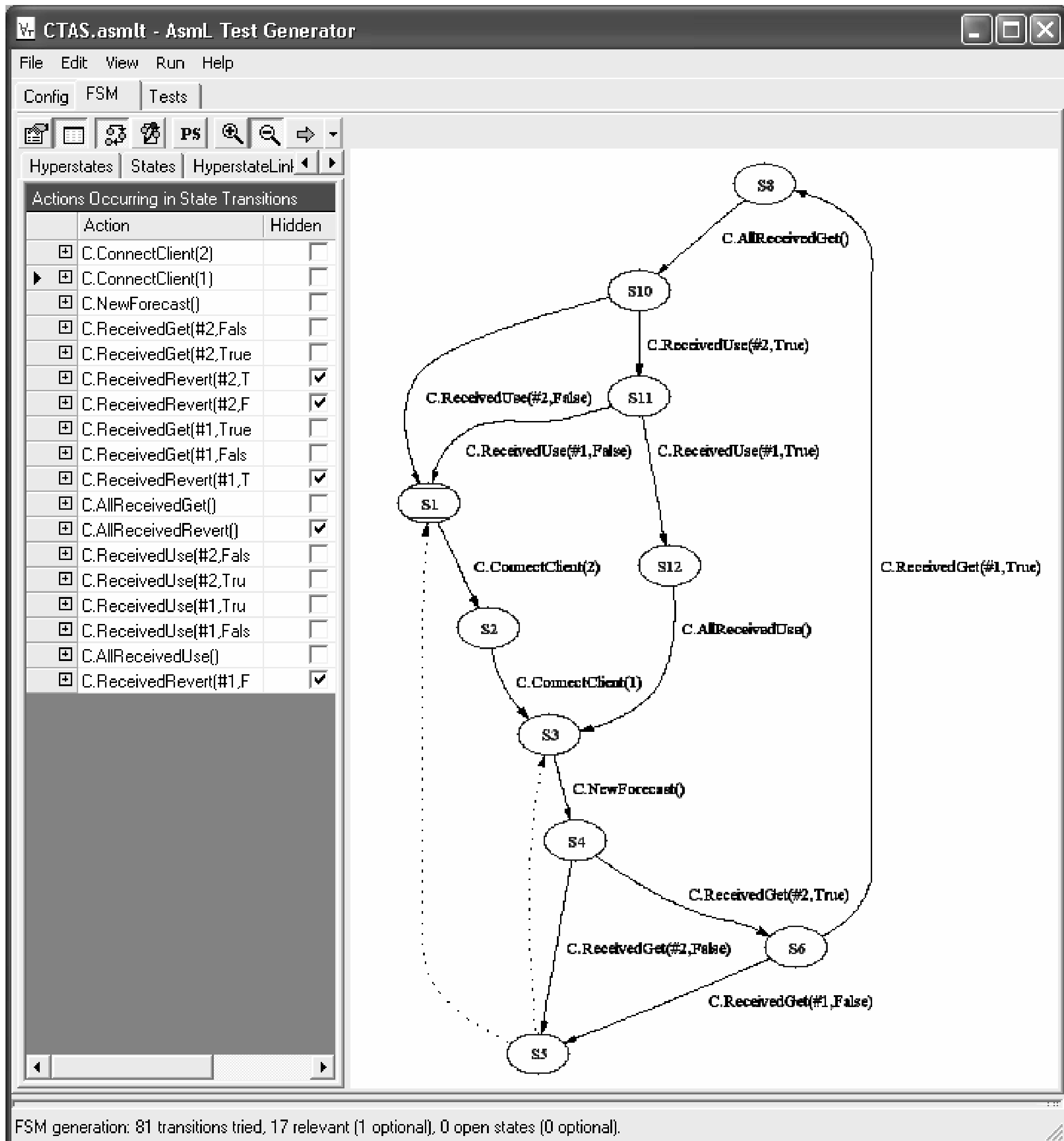
- *Filters*. The use can supply predicates on the concrete state. Only those states which pass the filter are considered for exploration; exploration is stopped at a state which does not pass the filter.
- *Model Coverage*. The user can supply a bound for model branch coverage; when the coverage exceeds that bound, exploration stops.

For the CTAS we have a finite state space since we configured only two clients which can ever connect. However, it still makes sense to not explore the state exhaustively, in order to get a more compact understanding of the system's behavior.

First, we want to abstract the order in which clients do something: whether first client #1 and then #2 connects, receives a weather report, etc., or vice versa, is not of interest to us. This is achieved by the following abstraction:

```
property CTASAbstr as (STATUS,
                       Map of STATUS to Integer)
  get return (wc.status,
              { st ->
                  [st|so in wc.sockets
                     where so.status = st].Length
              | st in enum of STATUS })
```

The domain of the state abstraction is a pair of the status of the CM and a multi-set of the status of connected



clients (where the multi-set is presented as a mapping from a status into occurrences). For example, the sequence of events where first client #1 connects and then client #2 will lead to the same multi-set as in the opposite way ({DONE->2,...}, since clients are in state DONE after connection.

Another behavior we are not interested to see is a weather forecast event in a trivial configuration where zero or one client is connected. To exclude this behavior we use a filter:

```
property CTASFilter as Boolean
  get return wc.status <> STATUS.DONE
                 implies Size(wc.sockets) > 1
```

This predicate filters out CM cycles with less then 2 clients connected.

Adding the abstraction and filter to the CTAS configuration and then generating the FSM leads to the output shown in the above screen shot. The visible FSM shows the behavior we expected. The screenshot shows a view of the FSM's automatic layout where actions belonging to the reverting phase of the CTAS are hidden (for reasons of space). These actions are collapsed into the transitions with dotted lines: successful reverting leads us from S5 back to S3 from where a new forecast can be handled, failing revert shuts down the CTAS and leads to the initial state where no client is connected.

The AsmL tester allows generating sequences of actions from the FSM which cover all branches. Since the CTAS example is a cyclic system where all states are connected, we get a single sequence from the FSM consisting of 44 actions. The value of the `events` variable of the use case in the last step of this sequence gives us a corresponding sequence of events which can be used for conformance testing of an implementation of the CTAS weather control logic.

6 Discussion and Conclusion

The transparent integration of use cases in a full-scaled formal modeling language like AsmL provides opportunities far beyond those of informal models. We can execute the model to validate the design and we can instrument it for test generation and conformance testing. This is in principle not a new message to the research community — but we are making it happen in reality.

The approach of embedding use cases in a host language is powerful. In our case, it enables us to use all the facilities provided by AsmL — like parameterization, the programmatic control structures, and the wealth of support for data. The drawback of this approach is that the user needs to master the host language. AsmL was carefully designed to avoid some of the complexity issues associated with earlier specification languages, such as Z, VDM, or algebraic specifications. For example, the language avoids using special math symbols and adopts a simple notational style similar to pseudo-code and languages like Visual Basic. Nevertheless, some complexity is inherent. On the other hand, if a user masters one application of AsmL, it becomes easy for her to master other applications.

In reality, the way use cases are written depends on guidelines which change from organization to organization. The original CTAS specification shows that. Our embedding approach has the advantage of being flexible enough to act as an augmentation to existing approaches instead of replacing them.

Graphical notations like those of UML are not yet commonplace in industry, but they probably improve communicating the requirements and the design with the customer. A graphical model is not necessarily easier to write, but it can be easier to read, in particular allowing the reader to poke into the model on different levels of detail. We believe it is a feasible approach to have graphical notations like Statecharts or message sequence diagrams as a front-end to a general modeling language like AsmL, and plan to explore tool support for this in the future.

References

[1] W. Grieskamp, M. Lepper, W. Schulte, and N. Tillmann: **Testable Use Cases in the Abstract State Machine Language** in *Proceedings of Asia-Pacific Conference on Quality Software* (APAQS'01), December 2001.

[2] Foundations of Software Engineering, Microsoft Research: **AsmL System** (software package). http://www.research.microsoft.com/foundations/asml.

[3] Y. Gurevich: **Evolving Algebra 1993: Lipari Guide**, *in Specification and Validation Methods*, Ed. E. Börger, Oxford University Press, 1995.

[4] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes: **Generating Finite State Machines from Abstract State Machines** in *ISSTA 2002, International Symposium on Software Testing and Analysis* , July 2002.

[5] M. Barnett and W. Schulte: **Runtime Verification of .NET Contracts** in *The Journal of Systems and Software*, Elsevier, 2002.

[6] Y. Gurevich and N. Tillmann. **Partial Updates: Exploration**. *Journal of Universal Computer Science*, 11 (7): 917-951, Springer Pub. Co, 2001.

[7] Y. Gurevich and N. Tillmann. **Partial Updates Exploration II**. In *Proc. Abstract State Machines 2003*, *LNCS*, Vol 2589, pages 57-86, Springer, 2003.