Validating Use-Cases with the AsmL Test Tool

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

Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
{mbarnett,wrwg,schulte,nikolait,margus}@microsoft.com

Abstract. The Abstract State Machine Language supports use-case oriented modeling in a faithful way. In this paper we discuss how the AsmL test tool, a new component of the AsmL tool environment, is used to generate finite state machines from use-case models which can be used for validation purposes or for testing.[1]

1 Introduction

The Abstract State Machine Language (AsmL) is an executable modeling language which is fully integrated in the .NET framework and Microsoft development tools. AsmL is designed to meet modeling needs arising in requirement and design specifications. This paper shows how AsmL can be used in a faithful way for use-case/scenario oriented modeling and how these models can be used for validation and verification purposes with the AsmL test tool. The paper refines and extends earlier work on AsmL and use cases [1].

The AsmL test tool is the newest component of the AsmL system [2]. It supports the generation of parameters, of call sequences, and the conduction of conformance tests. The tool realizes a semi-automatic approach, requiring a user to annotate models with information for generating tests. On the basis of the annotated model, parameter sets, a finite state machine, and call sequences are algorithmically derived. Some basic aspects of the AsmL test tool have been described in [4][2]. In this paper we investigate the use of the AsmL test tool for generating finite state machines from use-case models, which serve as a validation for the model and as a starting point for generating conformance tests.

The paper is organized as follows. We start with a sketch of AsmL. We then describe how we model use-cases in AsmL. We then introduce as a non-trivial example a model for the weather control logic of CTAS, a flight control system. We will use the AsmL test tool for deriving a finite state machine for the CTAS weather control logic which puts together the scenarios of the model into a coherent view of the behavior. The paper concludes with a discussion and comparison of related work.

2 A Glimpse of AsmL

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[3]. 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 [3]; and it has all the ingredients of a .NET language like 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 (ASM) [3]. An ASM is a state machine which 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. The computation of an update set can be complex, and the number of updates calculated may depend on the current state. The expressive power of AsmL in modeling is an extension of basic ASMs to that of *nondeterministic synchronous*

[1] Note to referees: this paper is an extended version of a paper accepted for presentation at the SCESM'03 workshop; the proceedings of the workshop do not have a formal publication status.

[2] We expect to have more references for the tool available at publication time of this paper.

[3] At the time of this writing, there is no publication about the AsmL language available. However, the AsmL distribution [1] contains a tutorial and reference.

parallel ASMs with submachines. AsmL uses the theory of partial updates [11,12] and the concept of state background for ASMs [10].

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. Upon an exception, all of the updates are rolled back, enabling atomic transactions to be built from many sub-steps.

AsmL supports meta-modeling and introspection that 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 executing the formal parts it contains or for working with the AsmL test tool.

3 Use-Cases 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. An event is described by an interface which just serves as a type tag. The class contains an ASM variable holding a sequence of events. The `DO` helper method appends to this sequence. If a use-case is "played" we can think of this variable as holding the history of what has happened so far.

```
interface Event
abstract class UseCase
  var events as Seq of Event = []
  DO(e as Event)
    events := events + [e]
```

To give life to these definitions, let us consider a simple example, 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 )
```

So far, we have a problem-oriented notation for use-cases in AsmL. The use-cases are type-checked and can be executed by calling the scenario methods. For example, the following top-level AsmL definition allows one to "play" the scenarios for the keycard controlled door:

```
PlayDoor(numberOfIters as Integer) as Seq of Event
  let door = new KeycardControlledDoor()
  step for i=1 to numberOfIters
    choose oracle in {true,false}
      if oracle
        door.NormalScenario()
      else
        door.InvalidCardScenario()
  step
    return door.events
```

`PlayDoor(3)`[4] will result in a sequence of events, and due to the non-deterministic choice of the scenario, different ones over time. With the expression **`explore`** `PlayDoor(3)` we can actually explore *all* the different choices taken, resulting in 8 sequences of events, covering the behavior described for the use-case with a chosen iteration depth of 3. (In general, the AsmL **`explore`** expression takes an arbitrary expression and delivers the sequence of the results of executing all possible combinations of choices in the expression.) Note that in [1] we needed a much more complicated setup to basically achieve the same functionality, which is now built into the AsmL language.

4 Example: CTAS Weather Control Logic

CTAS weather control logic is suggested by the organizers of the SCESM 2003 workshop as a case study for scenario oriented modeling [5]. 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 written by NASA. 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

[4] 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".

guarantee atomicity: new weather information becomes effective only if all clients successfully receive the new weather information. Our approach to use-cases in AsmL allows us a nearly one-to-one translation from the original spec (note that the choice of identifiers is also taken from the original spec and not invented by us).

4.1 Data Domains and State

We start with modeling some data domains. The (simplified) `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):

```
class CTASWeatherControl extends UseCase
  enum STATUS
    PREUPDATING
    UPDATING
    POSTUPDATING
    POSTREVERTING
    DONE
```

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 events related to the environment; the former both are parameterized by a client id:

```
class CTASWeatherControl
  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 add a socket class to the class `CTASWeatherControl`. It holds the id of the client and its status:

```
class CTASWeatherControl
  class SOCKET
    id         as CLIENTID
    var status as STATUS
```

We can now define the data state of the use case. It consists of the current cycle status of the CM and a set of sockets representing the clients with their status. Note that this is the state of the *entire* system, not of an actor like the CM or a client.

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

4.2 Scenarios

We start with a scenario for a client connecting with the CM. This scenario is *parameterized* over the client's id. 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 happen only 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 technique of parameterized scenarios will be used in our approach whenever we need to invent some data to synthesize events.

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, 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
```

This finishes the CTAS model. As with the keycard controlled door, we could give now definitions which play the scenarios of the CTAS. However, a more powerful approach to analyze the behavior is provided by the AsmL test tool.

5 The AsmL Test Tool

The AsmL test tool supports exploring a model's behavior by various means. The main purpose of the tool is to generate test suites and conduct conformance tests on the basis of a model, but the tool is also useful to

understand the behavior of a model, which is the main focus of this paper. The technologies currently used in the tool are the followings:

- *Parameter generation*: given annotations on types and/or methods providing domain information, and a precondition or invariant on the parameters, parameter tuples are automatically generated. Conceptually, the product of the domains of each parameter is generated (including the inductive generation of terms for nested/recursive types like trees and general graphs), filtered by the precondition/invariant. In fact, filter promotion is used to optimize the process. (In this paper, we won't use much of the powerful facilities for parameter generation found in the AsmL test tool but focus on call sequence generation.)
- *Call sequence generation:* our approach to call sequence generation is divided into two phases. First we generate a *finite state machine* (FSM) from the model [4]. This is done as follows: starting from the initial state, the state space is transitively explored by executing all enabled actions (those whose precondition holds). By defining so-called state abstraction properties and filters, the user can control when the exploration is terminated (we discuss this in more detail below). Once the FSM is generated, we use standard techniques to generate a set of sequences covering all paths of the FSM in an optimal way (we use a version of the algorithm found in [6]).
- *Conformance testing*: given a model-to-implementation binding that relates types and methods, the model is used to verify whether the implementation conforms to the specified behavior, running the test sequences generated in a previous step. To achieve this, we do not need the source of the implementation; instead we modify it at the binary level in order to monitor all API method calls. (In this paper, we won't use the facilities for conformance testing.)

Here, we will focus on the FSM generation technique to understand the behavior of the CTAS model.

6 Generating an FSM for CTAS

The first step in preparing for FSM generation is configuring variables and actions of the abstract state machine to explore. The variables constitute the relevant state of the ASM. During exploration, states which are identical regarding these variables are identified. The actions are methods which shall be used for exploration. A variable can either be shared or instance based; in the last case, a domain for the instance type needs to be configured to provide values for the instances. If a variable v is instance based, and $i_1,...,i_n$ is the domain for the instance type, then the tuple $(i_1.v,...,i_n.v)$ will be part of the relevant state.

For the CTAS example, as variables we use the CM cycle status, the set of client sockets and the client cycle status per socket; all these variables are instance based. As actions we use the scenario methods. Note that each scenario actually describes a sequence of use-case events, though it is an atomic action of the ASM.

Once we have configured the ASM we need to provide domains for the types of instance variables and parameters of methods. The tool allows defining these domains as arbitrary AsmL expressions which depend on the current state.

For example, we need to tell the tool the domain of the socket type since it is required to obtain instances for the client cycle status variables and for parameters of scenarios like `ReceivedGet`. We can use the current value of the variable `sockets` of the CTAS use case. Naturally, this variable presents those sockets in each step of the ASM whose client cycle state is relevant and which should be considered as a parameter for the scenarios. (In general, we have found that the domains needed for object types naturally arise from the model.)

To define the domain of the `CTASWeatherControl` type itself we introduce a constant which represents the use case; the domain is then the singleton set containing this constant:

```
const CTAS = new CTASWeatherControl()
```

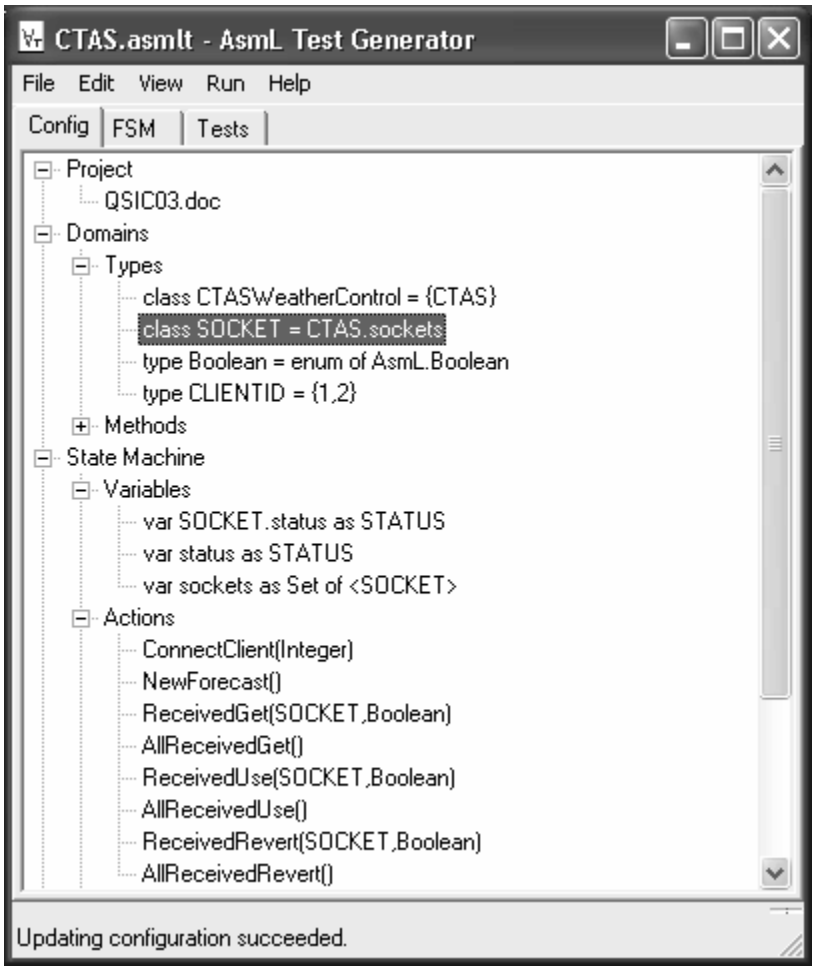

Figure 1: Configuration for the CTAS

The complete configuration for the CTAS is shown in the screen shot given in Fig. 1. In addition to the ones mentioned, we have defined the domains for Boolean to be the enumeration of this type (true and false), and for client ids to be a set containing two numbers (thus we will have only two clients which connect to the CTAS in this configuration).

The next step is the configuration for FSM generation: to define state abstraction properties and other means to control the state exploration. Our state exploration algorithm works by applying enabled ASM actions from the initial state with the provided parameters in a breadth-first way; actions are enabled if their precondition (`require` form) is true in the current state. This exploration potentially does not terminate in feasible time if the state space is not finite or of a huge size; but even if the exploration space is feasible, we might want to reduce it to get a more comprehensive picture.

The state abstraction properties allow us to group states into equivalence classes; when we encounter a state for which we have already seen an equivalent one according to the state abstraction we stop exploration at this point. For the CTAS configuration we actually have a finite state space (since there are only two clients). However, there are symmetrical behaviors we do not want to distinguish, for example, the order in which clients connect, or perform other actions. This is achieved by the following abstraction:

```
property CTASAbstraction as (STATUS, Map of STATUS to Integer)
  get return (CTAS.status,
              { st -> [st | so in CTAS.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 order (`{DONE->2,...}`, since clients are in state DONE after connection.)

A further means to control the state exploration are *filters*. If filters are given, then only those states which pass the filter are considered for exploration. For the CTAS example, we want to restrict the exploration to those states where a non-trivial number of clients is connected to the CM. We can express this as follows: whenever the CM is not in status DONE, there must be more then one client connected:

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

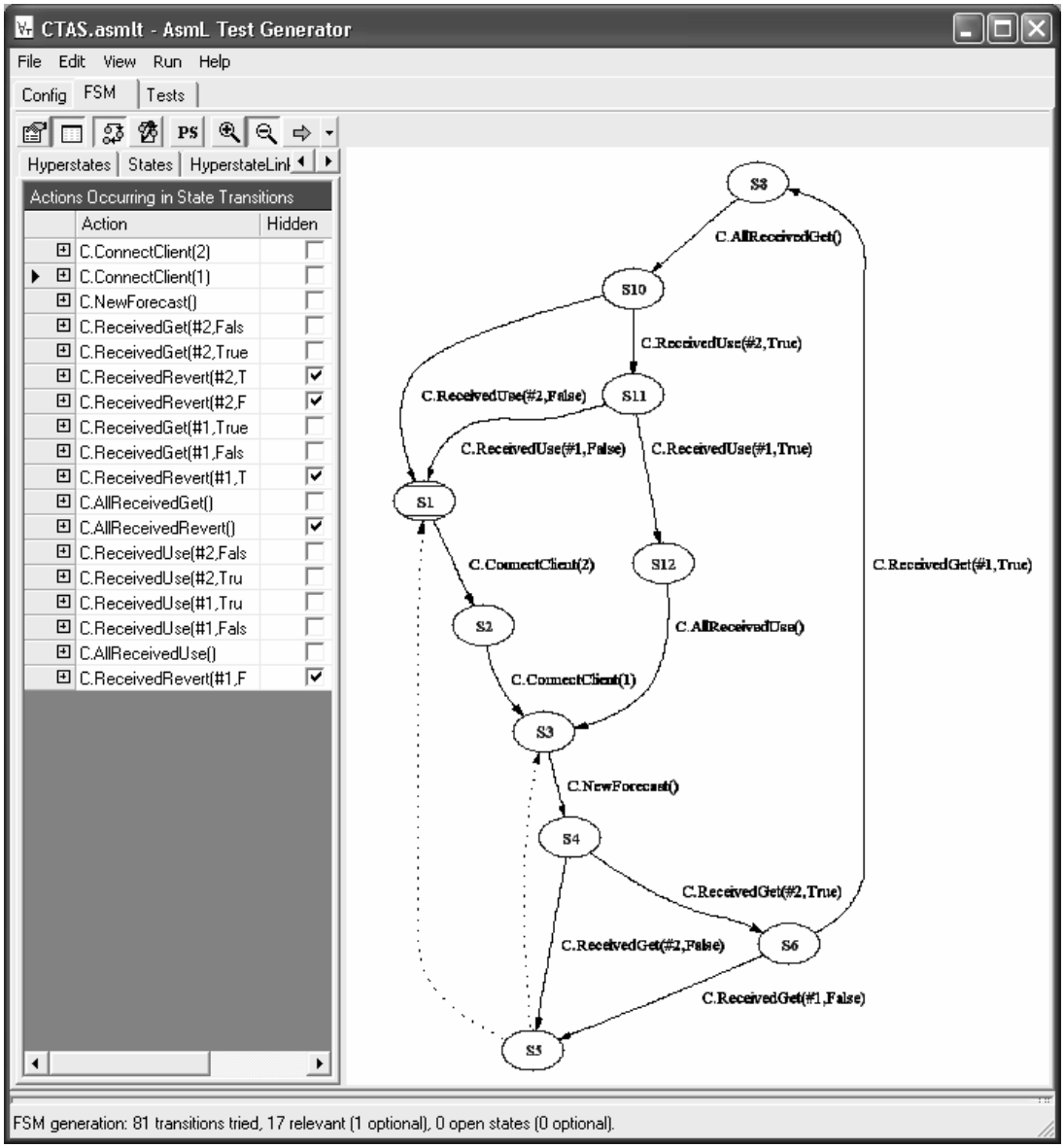

Figure 2: FSM for CTAS (revert actions hidden)

The result of the FSM generation with this property and filter is shown in Fig. 2. The visible FSM shows the behavior we expected. The screenshot shows a view of the FSMs 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.

For sake of completeness of this document as a formal input to the AsmL test tool, we provide two further auxiliary definitions for the printout of sockets and the CTAS use case object (this representation is seen in the screenshot):

```
class CTASWeatherControl
  class SOCKET
    override ToString() as String?
      return "#" + id
class CTASWeatherControl
  override ToString() as String?
    return "C"
```

7 Discussion and Conclusion

In this paper we showed with a non-trivial example the application of AsmL for use-case/scenario oriented modeling and how the AsmL test tool can be used to visualize the behavior of the use-case model as a finite state machine. The visualization of the FSM serves at least as a validation of the model. But we can do more. The AsmL test tool 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 when running the AsmL test tool. 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. This sequence contains all combination of behaviors where two clients are connected to the CM and where updating the weather succeeds or fails in various ways, including the reverting phase on failure. It is easy to generate longer sequences by increasing e.g. the number of clients which can connect to the CM.

We have presented earlier work on use-cases in AsmL in [1]. In contrast, this paper presents a much simplified technical approach which is enabled by meta-modeling facilities of AsmL which have been recently added, and by the AsmL Test Tool which is based on these facilities. Though we haven't discussed it in this paper, we nevertheless believe one general message of [1] is still very true: use-case modeling in the style we presented in this paper has to augment existing techniques, e.g. by means of annotation of informal use-cases with AsmL fragments, as we showed in [1].

The basic FSM generation algorithm that is implemented in the AsmL test tool is described in [4]. 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 related 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 model. Heuristics are used differently in the two approaches: in the DNF approach, heuristics are used as part of theorem proving, whereas we use heuristics to prune the search space.

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 tool by using properties. Due to efficiency considerations, the standard data abstraction algorithms may yield an *over-approximation* of M_r; see [8]. 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 [9] the counter examples of SPIN are considered as test cases generated from a given ASM and a given property.

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].

References

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

[2] AsmL for Microsoft .NET (version 2.1.5.7 or higher), Software Distribution. Containing Tools, Samples and Documentation. Downloadable at http://www.research.microsoft.com/foundations/asml.

[3] Y. Gurevich: Evolving Algebra 1993: 3 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, *ISSTA 02, Software Engineering Notes* 27(4) 112-122, ACM, 2002.

[5] CTAS case study, http://www.doc.ic.ac.uk/~su2/SCESM/CS/.

[6] H. Bast, K. Mehlhorn, G. Schäfer, and H.Tamaki. A heuristic for Dijkstra's algorithm with many targets and its use in weighted matching algorithms. In *ESA*, Lecture Notes in Computer Science, pages 242-253, 2001.

[7] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In Proc. *FME'93*, LNCS 670, p. 268-284, Springer, 1993.

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

[9] A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to Generate Tests from ASM Specifications. In *Proc. Abstract State Machines 2003*, *LNCS*, Vol 2589, pages 263-277, Springer, 2003.

[10] A. Blass and Y. Gurevich. Background, reserve, and Gandy machines, in *Proc. Computer Science Logic, Lecture Notes in Computer Science*, Vol. 1862, pages 1-17, Springer, 2000.

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

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