Instrumenting Scenarios in a Model-Driven Development Environment

Wolfgang Grieskamp, Nikolai Tillmann and Margus Veanes

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

Abstract

SpecExplorer is an integrated environment for model-driven development of .NET software. In this paper we discuss how scenarios can be described in SpecExplorer's modeling language, Spec#, and how the SpecExplorer tool can be used to validate those scenarios by various means.

Keywords: Model-driven development, scenarios, use-cases, validation, verification, automated testing

1 Introduction

SpecExplorer is an integrated environment for model-driven development of .NET software which is being developed at Microsoft Research. It is a successor of a tool previously known as the AsmL test tool [5]. SpecExplorer supports authoring models in a C#-like modeling language, Spec#, exploring their behavior by various means, and relating them to implementations.

This paper shows how SpecExplorer and its language, Spec#, can be applied for scenario-oriented modeling and how these models can be instrumented for validation purposes. The paper refines and extends earlier work of the authors on scenarios [1,15].

Scenario-oriented modeling is understood as a *methodology* for describing the behavior of a system from a *global* perspective by looking at the observable interactions between components in the system. A use-case, in our understanding given by a set of scenarios, describes how the components interact for a certain purpose (or *service*). This view on scenarios and use-cases is not necessarily restricted to a particular notation like message sequence charts (MSCs). Rather, we describe scenarios in this paper in an algorithmic way, using the modeling language Spec#. Our ultimate goal is to have tailored notations like MSCs as front-ends to this kind of descriptions. However, the focus here is not on these user-level notations, but on the underlying logic and instrumentation which we propose can be faithfully implemented in Spec# and instrumented in SpecExplorer.

The paper is organized as follows. We start with a sketch of Spec#, and then describe how we encode in principle scenarios in Spec#. We then introduce, as a non-trivial example, a model for the weather control logic of CTAS, a flight control system, which has been selected as a common case study for the ICSE 2003 Workshop on Scenarios and State. We will then use the SpecExplorer tool for exploring the CTAS model by generating a finite state machine which gives a coherent view of the behavior modeled by the scenarios. Finally, we discuss test automation based on these results. The paper concludes with a discussion and comparison of related work.

2 A Sketch of Spec#

Since Spec# is an extension of C#, we are confident to rely on the readers' intuitive understanding of the language as used in the examples. [1]

Spec# integrates features known from modeling languages like VDM, Z and ASM into a C# notation style which is supposed to be easily comprehensible by users in an industrial development context. Spec# is a true superset of C#, providing all features of C# (objects, interfaces, inheritance, method overriding, properties, events, and so on), and adding a few constructs which support modeling. Among those the most important are:

- *Contracts*. One can define pre- and post-conditions, as well as invariants in Spec#.
- *High-Level data types.* There is library and language support for high-level data types such as finite sets, maps, bags, sequences and so on.
- *Comprehensions.* Spec# provides a generalized comprehension notation for denoting aggregate type comprehensions (e.g. the "set of all numbers which satisfy a condition"), as well as logical connectives, like universal and existential quantification.
- *Nondeterminism*. Spec# provides non-deterministic choice which is required in particular for supporting partial/loose models, avoiding over-specification.

[1] Spec# is the successor of an earlier modeling language developed at Microsoft Research, called AsmL. It takes over many concepts of AsmL, but in an appearance similar to C#.

Spec# documents can be authored as plain text, XML, or Microsoft Word documents and can be compiled from Visual Studio .NET or from Micorosft Word (which is integrated as an editor into SpecExplorer). Round tripping between XML and Word is possible. Note that this paper is itself a valid Spec# document which can be fed directly as a source file into SpecExplorer.

In addition to syntactic and semantic language features, Spec#'s implementation supports meta-programming and introspection that allows a systematic exploration of the model's behavior, e.g. for the purpose of model checking, test generation, and test evaluation. On the meta-level the state of a Spec# model is a first-class citizen, which enables us to realize various search strategies over the state space of a model, which can be written in Spec# itself or any other .NET-compliant language.

3 Scenarios in Spec#

We consider a *use case* to be semantically a set of *interaction sequences* consisting of *actions*, where each action in turn is an observable activity of a component of a system. Use cases are described by a set of *scenarios*, where scenarios can describe one or more instances of similar interaction sequences[2].

The scenarios of a use-case describe the collaboration of the components of the system for a particular purpose, also called *service*. The same components can be involved in different use-cases, collaborating on different purposes/services. A use-case thus is a *viewpoint* on the system's *integrated* behavior.

The power of scenario-oriented modeling stems from that it

– allows to model the feature interactions between components, in contrast to direct component-wise modeling, where the interactions only follow indirectly from the component interface models; and it
– allows to keep open details of component behavior, which is useful in particular during early requirement specification.

Our goal is to describe scenarios programmatically by using the modeling features of Spec#. To this end, we first need to fix the understanding of an *action*. In an object-oriented programming world as it is provided by .NET, an action naturally amounts to a method call on a component class or interface, with given parameters and possible return values. Observing an action thus amounts to observing a method call. To describe such observations in scenarios, we extended Spec# by the so-called expect statement. This statement is introduced by the **expect** keyword, followed by a method call or an equality test over the result of a method call:

```
expect o.M(x1,...); // void action
expect y == o.M(x1,...); // non-void
```

As an illustrating example, let us consider a simple example, a keycard controlled door. The actors of this system consist of a user and the door, which are described by two interfaces:

```
interface IUser {
  void SwipeCard();
}
interface IDoor {
  void WaitForCard();
  void ReleaseLock();
  void SignalInvalidCard();
}
```

The use-case consists of a global state containing the data base of known users, and two scenarios, describing the behavior for the "good" path (a valid user) and the "bad" path (an invalid users):

```
Set<IUser> knownUsers;
IDoor door;
void GoodPath(User user) {
  require user in knownUsers;
  expect door.WaitForCard();
  expect user.WipeCard();
  expect door.ReleaseLock();
}
void BadPath(User user) {
  require user notin knownUsers;
  expect door.WaitForCard();
  expect user.WipeCard();
  expect door.SignalInvalidCard();
}
```

Note the use of pre-conditions for describing enabling of scenarios. A pre-condition should be read as a logical implication: we allow the specified behavior to happen only when the pre-condition holds.

An independent formal description of the semantcsi of such use-case descriptions can be given easily (e.g. by the set of traces of method calls they denote), but here we prefer to give our actual underlying implementation in Spec#, which turns out to be not complicated either. The expect statement is just syntactic sugar for firing an event which is passed *reflection* information about the method call. Thus, for the general case of expect with result, we translate

```
expect y == o.M(x1,...)
```

to the code:

[2] [2] Other authors identify uses cases and scenarios. We prefer to make a distinction like in Jacobson's original definition of use cases [7]. However, we don't restrict scenarios to describe just one interaction sequence; by parameterization or other means, they can describe many (similar) sequences.

```
  if (ExpectEvent != null)
    ExpectEvent(y,o,Info("M"),x1,...);
```

This event is declared (in Spec# or C#) as

```
  delegate void ExpectEventHandler(
        object expectedResult,
        object instance,
        MethodInfo method,
        params object[] args);
  event ExpectEventHandler ExpectEvent;
```

Subscribers to the expect event handler determine the actual meaning of the use-case. A simple case could be that we just want to "print" the expected method call, in order to see what kind of actions the use-cased induces. In this case we would define (in Spec# or C#) the following event handler:

```
  void PrintHandler(
      object expectedResult,
      object instance,
      MethodInfo method,
      params object[] args) {
    Console.WriteLine(
        "{0} == {0}.{1}({2})",
        expectedResult, instance,
        info.Name,args);
  }
```

A more sophisticated event handler could first perform an actual call to a method implementing the desired semantics, and then compare whether the actual result matches the one expected by the use case, thus realizing conformance testing. This will be discussed in greater detail later in this paper.

We could define at this point a simple Spec# model program which realizes the "playing" of the door control use-case, using Spec#'s modeling features like non-deterministic choice to provide parameters for the scenarios, and exploration features to see all possible behaviors for a given configuration of users and a door. However, this kind of application is much better done using the SpecExplorer tool. We will see how that works after we introduce a more interesting use-case example.

4 CTAS Weather Control Logic

CTAS weather control logic has been suggested by the organizers of the ICSE 2003 workshop on Scenarios and State Machines as a case study to compare tools and notations [3]. 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. Our approach to use-case modeling in Spec# allows us a nearly one-to-one translation from the original spec (we changed some identifiers and other small details for reasons of comprehensibility).

The interesting aspect of the weather updating process is that it has to guarantee atomicity: new weather information becomes effective only if all clients successfully receive the new weather information. Essentially, the logic realized between the connection manager and its clients is a two-phase transaction protocol.

We start by defining the types of the actors of the system, which are the connection manager and its clients. The connection manager provides methods for representing the connection attempt of a client, for receiving a new weather forecast, and for receiving whether a client has successfully got, used, or reverted a weather report which has been distributed to him:

```
  class ConnectionManager {
    void Connect(Client client);
    void NewForecast();
    void ReceivedGet
         (Client client, bool ok);
    void ReceivedUse
         (Client client, bool ok);
    void ReceivedRevert
         (Client client, bool ok);
  }
```

A client has methods for representing the get command of a weather report, the command to use the last received weather (only if all clients successfully received a weather report, they should start using it), the command to revert to a previous weather report, and finally the command to close the connection:

```
  class Client {
    void GetNewWeather();
    void UseNewWeather();
    void RevertWeather();
    void CloseConnection();
  }
```

Note that the methods of the classes above are unimplemented (and not abstract). Spec# allows us to define methods without a body, in which case the body is automatically generated to throw an "unimplemented method" exception. In general, our approach to scenario-modeling can refer directly to the classes of an existing implementation to obtain the vocabulary of actions, can use interfaces, or can use unimplemented classes as above which are later filled in with the implementation. To run a use-case for validation, we only need to be able to create objects of the given actor classes, which don't need to be implemented until we actually want to perform conformance checking, for example.

In order to define the use-case for CTAS, we next define some enumerations and global state variables which represent the status of the connection manager and the clients in the system.

```
enum CMStatus {
  Ready, Updating,
  PostUpdating, Reverting
};
enum CLStatus {
  Ready,
  Updating, PostUpdating, Reverting
};
bool initialized = false;
ConnectionManager cm;
CMStatus cms = CMStatus.Ready;
Map<Client,CLStatus> cls = Map{};
```

The boolean flag `initialized` indicates whether the configuration for the CTAS is initialized; the initialization includes creating a connection manager object `cm` and client objects. The initialization scenario will not be defined here but later on when we put things together.

We are now ready to define the scenarios. The first one describes the connection of a client to the connection manager (requiring that the use case is initialized). Connection is only possible for clients who are not yet connected and only if the status of the connection manager is `Ready`:

```
void Connect(Client c) {
  require initialized;
  require cms == CMStatus.Ready;
  require c notin cls;
  expect cm.Connect(c);
  cls[c] = CLStatus.Ready;
}
```

Next we describe the scenario for a new weather forecast, which is enabled only when the status of the connection manager is `Ready`. On a new forecast, the connection manager and all connected clients change their status to `Updating`, and all clients are expected to receive the new weather information:

```
void NewForecast() {
  require initialized;
  require cms == CMStatus.Ready;
  expect cm.NewForecast();
  cms = CMStatus.Updating;
  foreach (c->s in cls) {
    expect c.GetNewWeather();
    cls[c] = CLStatus.Updating;
  }
}
```

Next we describe the scenario where the connection manager gets the notification that a client has or has not successfully received weather information:

```
void Got(Client c, bool ok) {
  require initialized;
  require cms == CMStatus.Updating;
  require cls[c] == CLStatus.Updating;
  expect cm.ReceivedGet(c,ok);
  if (ok) GotOk(c);
  else GotNotOk(c);
}
```

If getting weather information was successful, the client status changes to `PostUpdating`. If *all* clients in the system are in this state, then the connection manager itself also changes its status accordingly, and clients are expected to receive the command to use the new weather information:

```
void GotOk(Client c) {
  cls[c] = CLStatus.PostUpdating;
  if (Forall{s==CLStatus.PostUpdating:
             c1->s in cls}) {
    cms = CMStatus.PostUpdating;
    foreach (c1->s in cls)
      expect c1.UseNewWeather();
  }
}
```

If a client was not successful in getting weather information, the system changes it status to `Reverting`. All clients are expected to receive the command to revert weather information:

```
void GotNotOk(Client c) {
  cms = CMStatus.Reverting;
  foreach (c1->s in cls) {
    expect c1.RevertWeather();
    cls[c1] = CLStatus.Reverting;
  }
}
```

We next describe the scenario where the connection manager gets the notification that a client has successfully used the new weather information. It is very similar to the `Got` scenario, except that in case of a failure, the system shuts down:

```
void Used(Client c, bool ok) {
  require initialized;
  require cms == CMStatus.PostUpdating;
  require
    cls[c] == CLStatus.PostUpdating;
  expect cm.ReceivedUsed(c,ok);
  if (ok) UsedOk(c);
  else Shutdown();
}
void UsedOk(Client c) {
  cls[c] = CLStatus.Ready;
  if (Forall{s == CLStatus.Ready:
             c1->s in cls})
    cms = CMStatus.Ready;
}
```

On shut-down, all connected clients are disconnected, and the connection manager resets its status to Ready:

```
void Shutdown() {
  foreach (c->s in cls)
    expect c.CloseConnection();
  cls = Map{};
  cms = CMStatus.Ready;
}
```

Finally, we model the reverting phase of the protocol, where the connection manager receives notification whether a client could successfully revert its weather information. It is very similar to the `Used`-scenario. If revert failed, the system shuts down:

```
void Reverted(Client c, bool ok) {
  require initialized;
  require cms == CMStatus.Reverting;
  require
   cls[c] == CLStatus.Reverting;
  expect cm.ReceivedRevert(c,ok);
  if (ok) RevertedOk(c);
  else Shutdown();
}
void RevertedOk(Client c) {
  cls[c] = CLStatus.Ready;
  if (Forall{s == CLStatus.Ready:
             c1->s in cls})
    cms = CMStatus.Ready;
}
```

This finishes the CTAS model. In the next sections, we discuss how to use such a model for validation under SpecExplorer.

5 A Sketch of SpecExplorer

The SpecExplorer tool is an integrated environment for model-driven development of .NET software. It provides a platform for tool developers to integrate various techniques which leverage models in the development process. The tool, developed on base of experiences with its predecessor, the so-called AsmL test tool, is rather new. By the time of this writing it encompasses the following functionality:

- Authoring of (literate) models in various editors, in particular Microsoft Word.
- Annotating the model with information about its use in execution and exploration: what are the main actions of the model (the main entry points), which parameters to provide to these actions, predicates on states to be omitted in exploration, and so on.
- Exploring the model, continuously or in single-stepping mode, in various modes: random execution, state reachability checking, exhaustive exploration pruned by state partitioning, and so on.
- Displaying results of exploration by means of a state machine diagram.
- Generating test suites from state machines generated by exploration.
- Running conformance checks of test suites against actual implementations.

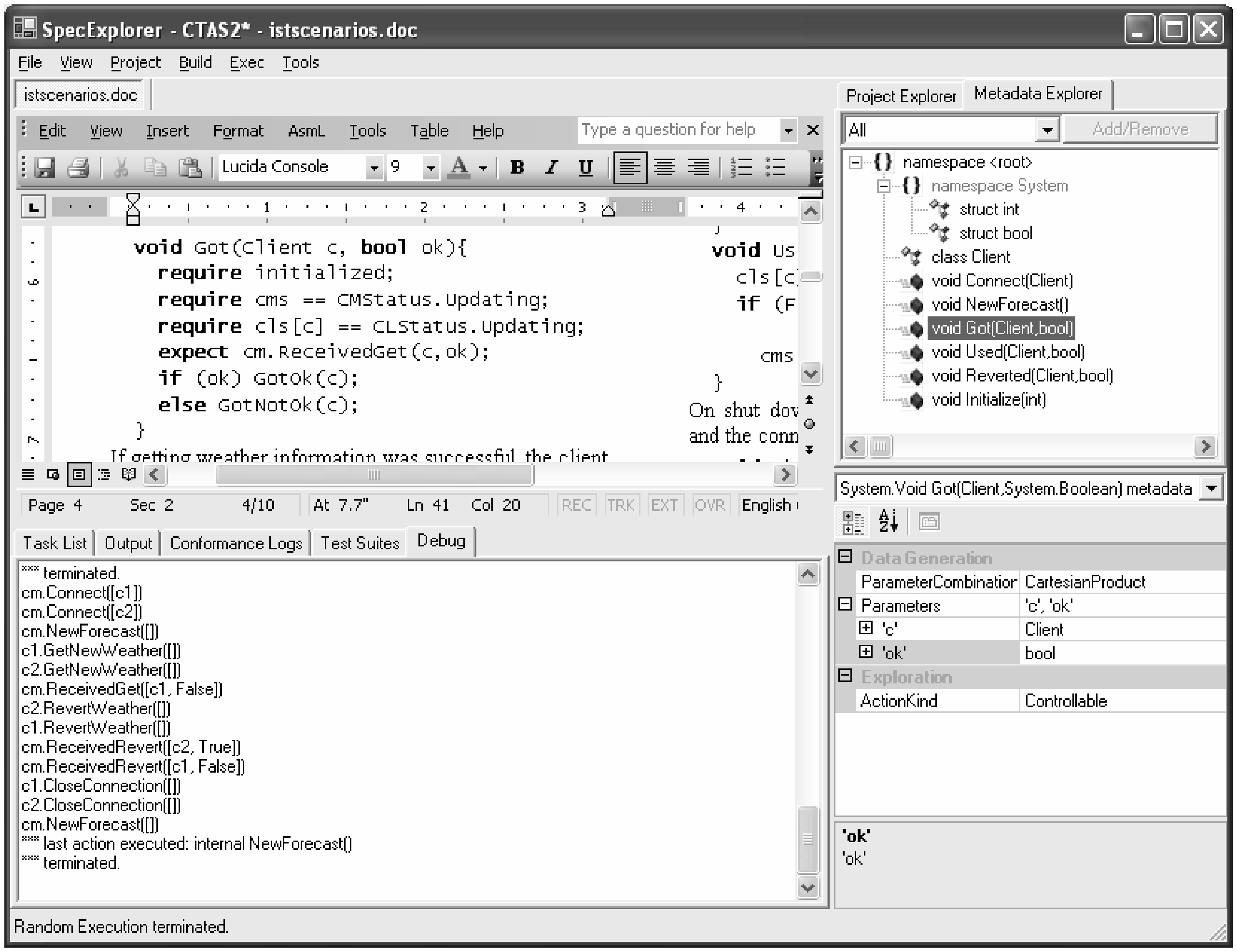
In this paper, we will only use some of those features, namely random execution and state machine generation. We also sketch a conformance checking algorithm which is tailored for the particular scenario-oriented modeling style we present, and which is currently being implemented for SpecExplorer.

6 Random Execution of CTAS

It is straightforward to employ a model like CTAS under SpecExplorer for random execution. An `ExpectEventHandler`, which prints out expected calls (as given in Sect. 3) can be used to get first insights into the behavior of the model.

In order to enable random execution, we need to annotate the CTAS model with information about the top-level actions[3] and their parameters. The actions in our case are the scenarios of the CTAS model, plus one further pseudo-scenario which sets up a configuration of a connection manager and its installed clients. This scenario is parameterized over the number of clients we want to install:

[3] Please note the overloading of notions of actions in scenarios and in SpecExplorer. An action in a scenario is an method call on the interface of a component which we observe using the expect-statement. An action in SpecExplorer is a top-level entry point of the model.


SpecExplorer - CTAS2* - istscenarios.doc
File View Project Build Exec Tools
istscenarios.doc
Edit View Insert Format AsmL Tools Table Help
Type a question for help
Lucida Console 9
Page 4 Sec 2 4/10 At 7.7" Ln 41 Col 20 REC TRK EXT OVR English
Task List Output Conformance Logs Test Suites Debug
Project Explorer Metadata Explorer
All Add/Remove
System.Void Got(Client,System.Boolean) metadata

```
void Got(Client c, bool ok){
  require initialized;
  require cms == CMStatus.Updating;
  require cls[c] == CLStatus.Updating;
  expect cm.ReceivedGet(c,ok);
  if (ok) GotOk(c);
  else GotNotOk(c);
}
```

```
*** terminated.
cm.Connect([c1])
cm.Connect([c2])
cm.NewForecast([])
c1.GetNewWeather([])
c2.GetNewWeather([])
cm.ReceivedGet([c1, False])
c2.RevertWeather([])
c1.RevertWeather([])
cm.ReceivedRevert([c2, True])
cm.ReceivedRevert([c1, False])
c1.CloseConnection([])
c2.CloseConnection([])
cm.NewForecast([])
*** last action executed: internal NewForecast()
*** terminated.
```

```
namespace <root>
  namespace System
    struct int
    struct bool
  class Client
  void Connect(Client)
  void NewForecast()
  void Got(Client,bool)
  void Used(Client,bool)
  void Reverted(Client,bool)
  void Initialize(int)
```

Data Generation	
ParameterCombination	CartesianProduct
Parameters	'c', 'ok'
'c'	Client
'ok'	bool
Exploration	
ActionKind	Controllable

'ok'
'ok'

Random Execution terminated.

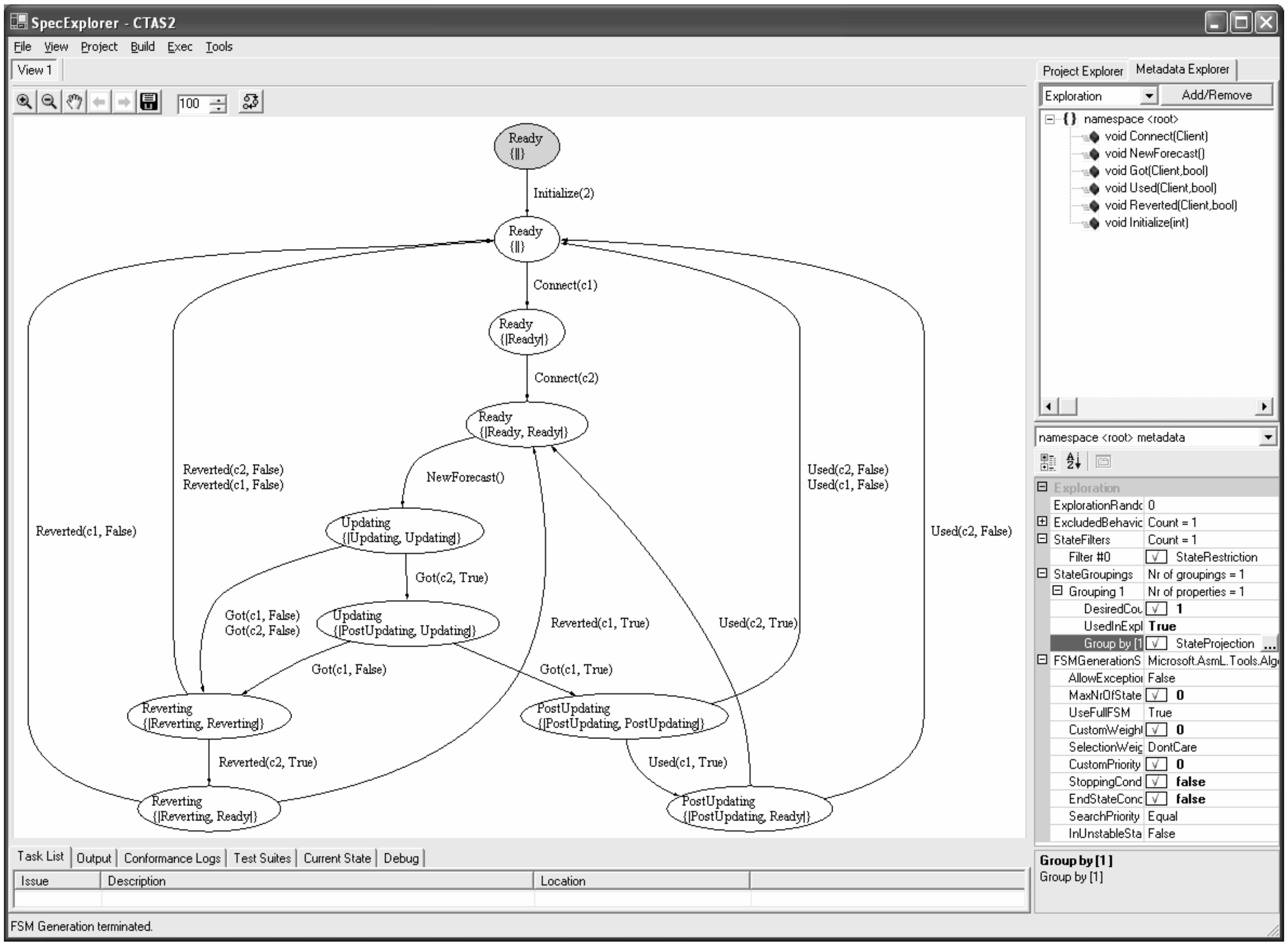
```
Set<Client> installedClients = Set{};
void Initialize(int numOfClients) {
  require !initialized;
  cm = new ConnectionManager();
  installedClients =
    Set{new Client():
        i in Set{1..numOfClients}};
  initialized = true;
}
```

Domain annotations can be done in SpecExplorer on a per-type and per-method base. If a parameter has no domain annotation, then its domain defaults to that of the type of the parameter. For CTAS, it is sufficient to annotate domains on a per-type base:

– The type `int`, which is used as an parameter for the `Initialize` scenario, is assigned the expression `Set{2}` (so we configure the CTAS to execute with two clients; other values are easily possible);
– The type `Client` is annotated with the expression `installedClients` (so all scenarios which expect a client parameter are tried with all installed clients; note, however, the pre-condition of the scenario might filter out clients for which it is not enabled);
– The type **`bool`**, used in various scenario parameters for representing acknowledge of a message, is annotated with the expression `Set{`**`true`**`,`**`false`**`}`.

Note that domain annotations are arbitrary expressions which are evaluated in the current state of the model (consider the case of the `installedClients` domain annotation).

Fig. 1 shows a screenshot of SpecExplorer after a random execution experiment. In the upper left corner, Microsoft Word is running as an embedded control, editing the CTAS model. On the upper right corner, the metadata explorer is shown, which allows annotating actions and domains of the model. The lower right corner displays information about the annotations for the method `Got`, which is configured to be an action of a certain kind, with parameter domains to be taken over from the annotation of the parameter types. The lower left corner shows the printout of the expect event handler on one random execution run. Different executions will result in different runs, determined by



namespace <root>
- void Connect(Client)
- void NewForecast()
- void Got(Client,bool)
- void Used(Client,bool)
- void Reverted(Client,bool)
- void Initialize(int)

Exploration	
ExplorationRandc	0
ExcludedBehavic	Count = 1
StateFilters	Count = 1
Filter #0	√ StateRestriction
StateGroupings	Nr of groupings = 1
Grouping 1	Nr of properties = 1
DesiredCou	√ **1**
UsedInExpl	**True**
Group by [1	√ StateProjection ...
FSMGenerationS	Microsoft.AsmL.Tools.Alg
AllowExceptio	False
MaxNrOfState	√ **0**
UseFullFSM	True
CustomWeigh	√ **0**
SelectionWeig	DontCare
CustomPriority	√ **0**
StoppingCond	√ **false**
EndStateConc	√ **false**
SearchPriority	Equal
InUnstableSta	False

Group by [1]
Group by [1]

FSM Generation terminated.

the random selection of parameters from the given parameter domains.

7 Generating an FSM for CTAS

Random execution is a simple application of the underlying powerful exploration engine of SpecExplorer. In general, this engine allows us to explore a model by various means, using techniques similar to that of an explicit state model checker. Another application of the exploration engine is finite state machine generation, which is used in the SpecExplorer tool primarily for test case generation, but is also useful on its own for validating the design of a model. We will show how to generate an FSM for CTAS.

The FSM generator does an exhaustive exploration of the model's state space. If directly applied to the annotations we have given for random execution (where two clients are involved), it produces an FSM with 48 transitions and 22 states. However, this FSM is already too large to be comprehensible. SpecExplorer provides a collection of techniques to prune exploration, which is essential if the model's state space is infinite (which is not the case for the CTAS as configured), and useful if one wants to get a coherent picture of the behavior. The pruning techniques are the followings:

– *Filters*: predicates over the state, which characterize the states to be included in the exploration. If a state does not satisfy all filters, it will be excluded.
– *State Partitioning*: one can define a projection function on the state which partitions the state space into state groups, where a group is the set of states areequal under the projection. If during exploration a state is visited, for which a configurable number of representatives in its class hasgroup already been explored, then this state will not be considered for further exploration [2].
– *Bounds*: one can define an upper bound of the number of states to visit.
– *Coverage*: one can define model coverage goals which, if reached, will prune exploration.

For the CTAS example, we will make use of filters and state partitioning. First, we define a filter which excludes the trivial behavior of a cycle of the CTAS with zero or one connected client. The filter reads as follows:

```
bool StateRestriction {
  get {
    return cms != CMStatus.Ready ==>
                        cls.Size > 1; }
}
```

This filter demands that in any state where the connection manager's status is not `Ready`, the number of connected clients should be greater than one.

We next define a state projection which abstracts in which order clients are interacting with the connection manager, effectively reducing the number of interleavings explored. The state abstraction delivers a pair of values, where the first element is the connection manager's status, and the second is a bag (multi-set) of the status of the clients:

```
<CMStatus,Bag<CLStatus>>
  StateProjection {
    get {
      return <cms,Bag{s: c->s in cls}>;
    }
  }
```

Using this filter and partitioning an FSM will be generated as visualized in the screenshot in Fig. 2. Note that the transitions of the FSM are the *scenarios* of the model, and not the expected actions of the actors of the use-case. The FSM clearly visualizes the intended behavior of the CTAS protocol, thus serving for validating the adequacy of the model.

8 Scenario Conformance Checking

The SpecExplorer tool contains an engine for checking conformance of implementations against models. This engine takes test suites generated by traversal algorithms on the FSM and executes them on an implementation. It supports the automatic binding of model actions against implementation actions, instruments the implementation by inserting callbacks for bounded implementation methods into the conformance engine, and checks the implementation by lock-step execution of model and implementation. Actions of SpecExplorer are thereby distinguished to be either *controllable* or *observable*. A controllable action is an input to the system under test, whereas an observable action is an output (in .NET, usually an event or callback).

This engine cannot be directly applied to our scenario-oriented modeling approach, since the actions of the implementation do not correspond to the actions explored by SpecExplorer, which are the scenarios. We have developed a simple tailored version of conformance checking for scenario models which is currently implemented.

The scenario checker works as follows. As SpecExplorer's actions, scenarios are distinguished to be either controllable or observable. By the nature of scenario-modeling, which describes observations over an autonomous system, scenarios will be configured to be observable most of the time. Exclusion of the rule is for example the `Initialize` scenario of CTAS, which controls setting up a system configuration; all other CTAS scenarios are observable.

For checking the scenarios, we define the expect event handler (conceptually) as follows:

```
bool isControllable;
void CheckHandler(
      object inst, object result,
      MethodInfo info,
      params object[] pars) {
  if (isControllable) {
    let implResult =
      Trigger(inst,info,pars);
    if (!implResult.Equals(result))
      throw new ConformanceFailure();
  } else {
    let timeout =
        Await(inst,info,pars,result);
    if (timeout)
      throw new ConformanceTryNext();
  }
}
```

The flag `isControllable` determines whether currently an observable or controllable scenario is checked. The methods `Trigger` and `Await` do the obvious thing, i.e. calling the according method in the implementation, or waiting for a call to happen with exactly the given parameters and result. Since SpecExplorer already supports instrumenting of implementations with callbacks, `Await` can be easily realized.

The scenario checking engine combines exploration and checking "on-the-fly". From a frontier of admissible model states, it determines the enabled scenarios, and partitions them into observable and controllable. It first tries to execute the observable scenarios; if any of those throws `ConformanceTryNext`, it is skipped. It then tries the controllable scenarios; if any of those fails, or if there is no scenario overall to apply, the conformance check fails.

Note that this technique supports only external choice in the implementation. Once a controllable scenario succeeds, there will be no point of return, i.e. backtracking to another choice if checking fails. Techniques are possible which also support internal choice; those are based on re-execution of the implementation. However, in our experience with applications of the SpecExplorer technology and its predecessor, these are rarely needed in praxis.

There are various heuristic extensions of this basic model which have been investigated in the realm of on-the-fly testing (e.g. [8]) and which are addressing smart selection of the next action to execute when several are enabled in a given state; we expect these techniques to be applicable to the scenario checker.

9 Discussion and Conclusion

In this paper we showed by employing a non-trivial example the application of Spec# and SpecExplorer for use-case/scenario oriented modeling. We demonstrated how we are able to explore a scenario model by execution, visualize its coherent behavior, and instrument it for conformance checking of an implementation. Since scenarios are represented programmatically, our approach is very powerful, because it gives us all the features of a general modeling and programming language like Spec#, which allows as describing the state of the system, the interfaces of the actors, and the scenarios in dependency to that. Consistency comes for free in our approach, since we do not introduce redundancy. We conclude with a discussion of related work and restrictions of our approach as well as future work implied by this.

9.1 Related Work

We have presented earlier work on use-cases and scenarios in [1] and [6]. This paper presents a much straightened approach compared to [1], and fills the gap we left in [6] regarding conformance checking. Also, whereas in [6] actions of system components needed to be represented as data values, here we can represent them more natural by directly using the methods provided by actor types, using the newly introduced expect-statement.

There are no other approaches we are aware of which closely integrate scenarios in a general purpose modeling language, and which are also directly connected to the world of implementation. A large collection of works exist which explores the formal semantics of scenarios, and their most common representation, message sequence charts (e.g. [10] or [11]). Our focus is not so much on formalizing the semantics, which by now is well-understood, but on instrumenting scenarios in model-driven development environment. Naturally, our approach is more tool-oriented then others.

Various authors presented work on synthesizing state machines from scenarios, e.g. [12] (by means of synthesizing Statecharts) or [9] (by means of synthesizing Markov chains). From the state machine representation, test suites can be derived, using the known techniques based on testing of finite state machines, which are also build into SpecExplorer [4]. Our approach for scenarios does not actually intends to use the FSM as a device for testing, but just as an intermediate result of the modeling process where it serves to visualize and validate the design of a use case. Instead, we propose to use on-the-fly testing [13] for scenario-based conformance testing, which is better capable to deal with the huge amounts of interleavings and behaviors which can be generated from a typically very loose scenario model.

The basic state exploration and FSM extraction algorithm that is implemented in the SpecExplorer tool is described in [2], and the predecessor of SpecExplorer, the AsmL test tool, in [5]. To our knowledge, there are no other model-driven development environments which provide a close integration of all the aspects of authoring, execution, exploration and conformance testing. Some tools implements parts of this functionality, for example TGV, TorX, or UniTesK. None of those tools, however, provides support for scenario-oriented modeling.

9.2 Future Work

Currently, there is a growing user base for Spec# and SpecExplorer at Microsoft, and those users are asking for scenario-oriented modeling features. However, one advantage of the approach presented in this paper is also one of its drawbacks for those users: by employing a full-fledged modeling language like Spec# for scenario-oriented modeling, the ease of access of scenario-oriented modeling by non-expert users may get lost. We regard it hence as essential to add diagram notations to our approach, possibly based on MSCs or LSCs, which recover some of the accessibility. However, our ambition is to provide diagram notations as a *view* on (a subset of) the programmatic representation of scenarios, so that a user can switch between those views, and has the programmatic description available for those cases where MSCs or LSCs are not sufficient powerful enough. Whether this approach works in practice has to be explored.

Our current approach to scenario-modeling has (at least) one severe restriction which needs to be addressed in the future: all execution and exploration of scenarios is done on ground data. This is actually a restriction of SpecExplorer's exploration framework itself, which does not support "unknowns" as values. In scenario-oriented modeling, this restriction is particularly hindering, since unknowns can be very handy here to avoid over-specification. Also, the restriction to ground data induces an efficiency problem for exploration, since variable bindings need to be unnecessarily varied over ground data, even if their value is never accessed.

Our approach does not currently incorporate concepts for composing scenarios, like defined in High-Level MSCs. Future work needs to look at this subject, which we expect to be a smooth extension to our approach.

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] 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.

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

[4] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann and W. Grieskamp. Optimal Strategies for Testing Nondeterministic Systems. Submitted to ISSTA'04.

[5] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Towards a tool environment for model-based testing with AsmL. In Petrenko and Ulrich, editors, *Formal Approaches to Software Testing, FATES 2003*, volume 2931 of *LNCS*, pages 264--280. Springer, 2004.

[6] Mike Barnett, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, Margus Veanes. Validating Use-Cases with the AsmL Test Tool. In Proceedings of *Third Internationl Conference On Quality Software (QSIC'03)*, IEEE, 2003.

[7] Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G. *Object-Oriented Software Engineering – A Use Case Driven Approach*. Reading, Mass. Addison-Wesley, 1992.

[8] Loe M.G. Feijs, Nicolae Goga, Sjouke Mauw, Jan Tretmans. Test Selection, Trace Distance, and Heuristics. In Proceedings of IFIP 14th International Conference on Testing Communicating Systems - TestCom 2002. Kluwer, 2002.

[9] Winfried Dulz, Fenhua Zhen. MaTeLo -- Statistical Usage Testing by Annotated Sequence Diagrams, Markov Chains, and TTCN-3. In Proceedings of *Third Internationl Conference On Quality Software (QSIC'03)*, IEEE, 2003.

[10] Martin Glinz. An Integrated Formal Method of Scenarios Based on Statecharts. In *Software Engineering -- ESEC'95. Proceedings of the 5th European Software Engineering Conference*. Springer, Berlin, 1995.

[11] Manfred Broy and Ingolf Krueger. Interaction Interfaces - Towards a scientific foundation of a methodological usage of Message Sequence Charts. In *Formal Engineering Methods (ICFEM'98)*. IEEE, 1998.

[12] Ingolf Krueger, Radu Grosu, Peter Scholz and Manfred Broy. From MSCs to statecharts. In *Distributed and Parallel Embedded Systems*. Kluwer, 1999.

[13] 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.