Australian Journal of Basic and Applied Sciences, 4(12): 6346-6353, 2010
ISSN 1991-8178

Using ASMs and Spec# to Formal Modeling and Analysis Publish/Subscribe Architectures

Azadeh Esfandyari

School of Computer Engineering, Islamic Azad University- Gilan-e-Qarb Branch

Abstract: The Publish/Subscribe architecture has been proposed as a suitable architecture to develop highly dynamic systems. Although the structure of this architecture is easy to understand, unfortunately modeling and validating the whole system is complicated. In this paper, we present a formal approach based on Abstract State Machines (ASM) to model systems using this architecture. Then, to validate the designed models we use model-based testing. To do so, we propose a transformation from ASMs to Spec# language. The key feature of the proposed approach are new parametric dispatcher and the use of model-based testing for validation.

Key words: Publish/subscribe; component; modelling; validation; ASM

INTRODUCTION

In highly dynamic environment where systems composed of very expandable set of components Publish/Subscribe paradigm is an interesting solution for developing such systems. According to this paradigm, components interact through a special-purpose element called dispatcher. The components can publish events and listen to (subscribe to) events from other components. The dispatcher receives events and forwards them to subscribers, that is, to all components registered to listen to them David Garlan, (2003).

The main property of these systems is that a publisher doesn't know its subscribers. The consequent loose coupling between components in a Publish/Subscribe system makes it relatively easy to add or remove components in a system, introduce new events, register new listeners on existing events, or modify the set of announcers of a given event. Thus implicit invocation systems support the ability to compose and evolve large and complex software systems out of independent components David Garlan, (2003). Due to these loose coupling and flexibility, modeling and validation of Publish/Subscribe system is complicated. Existing efforts for developing formal foundation for specifying and reasoning about these systems are hard to use by practitioners and non-formalists and need a great deal of proofing for execution. To face this challenge this paper proposes a formal framework for software development using Publish/Subscribe architecture. The success of the Abstract State Machines (ASMs) as a system engineering method able to guide the development of hardware and software systems, from requirements capture to their implementation (David Garlan, 2003; Börger and Stärk 2003). Their abstract nature allows the system designer to focus on system concepts and not to be disturbed by the details. ASMs can specify a system at different abstract levels. The system designer can refine the system from a more abstract level to a less abstract level by providing more details and making more design decisions.

Even though the ASM method has a strictness scientific foundation Börger, (2002), the practitioner needs no special training to use it since Abstract State Machines are a simple extension of Finite State Machines, and can be understood correctly as pseudo-code or Virtual Machines working over abstract data structures. ASMs allow a nowadays widely-requested modeling technique which integrates dynamic and static descriptions. Analysis techniques that combine validation and verification can be performed at any desired level of detail. Hence in this paper we propose ASMs as a suitable formal method for modeling Publish/Subscribe systems. Based on the notion of ASM run, various tools have been built to mechanically execute ASM models for their experimental validation by simulation and testing. Based upon the mathematical character of ASMs, also any standard mathematical verification techniques can be applied to prove or disprove ASM model properties, implying precision at the desired level of rigour: from proof sketches over traditional or formalized mathematical proofs to tool supported proof checking or interactive or automatic theorem proving David Garlan, (2003).

Corresponding Author: Azadeh Esfandyari, School of Computer Engineering, Islamic Azad University- Gilan-e-Qarb Branch

In this paper we use Spec Explorer for validation, model-based testing and Model exploration. Spec Explorer is a Microsoft tool that provides modeling software behavior, analyzing that behavior by graphical visualization, model checking David Garlan, (2003). Spec Explorer powerfully explores models in order to discover all the potential behaviors they define, and present a graphical view of the result. So the models are transformed into Spec Explorer modeling language.

Related Work:

There are several efforts (David Garlan, 2003; Börger and Stärk 2003) that have concentrated on the problem of constructing Publish/subscribe systems that present desirable run time qualities instead of the problem of reasoning about the correctness of such systems. Also there are some research (David Garlan, 2003; Börger and Stärk 2003) on formal reasoning for such systems, but they can't been applied by non-formalism and practitioner and need a great deal of proofing.

Garlan *et al.* (2003) provide a set of pluggable modules that allow a modeler to choose one possible instance out of a set of possible choices. However, the available models are far from fully capturing the different characteristics of existing Publish-Subscribe systems.

David Garlan, (2003) characterize the Publish-Subscribe infrastructure in terms of reliability, event delivery order and subscription delay. Still, it does not consider several characters. Finally, there have been several previous efforts at providing formal, checkable models for software architectures. Some of these efforts use model checkers to check properties of event-based systems. But, none has been adapted to the specific needs of publish/subscribe systems development.

Modeling:

In this phase all components must be modeled. First the behavior of parametric dispatcher is brought up and then how ASMs can be used to model components are shown.

Dispatcher:

The predefined dispatcher is instantiated by providing parameters that customize the dispatcher's behavior to reflect the different peculiarities of existing publish/subscribe middleware platforms David Garlan, (2003). On the one hand, many parameters would allow us to specify the very details of the behavior, but, on the other hand, they would complicate the model unnecessarily. So the identification of the minimal set of parameters is essential David Garlan, (2003). In additional to requisite parameters provided in David Garlan, (2003) such as delivery, notification, un/subscription and filtering policy, for fully capturing the different characteristics of existing Publish/Subscribe middleware platforms specifying following parameters are necessary. Table 1 shows the required parameters.

Tabel 1: Extra Required parameters

Subscription to the same evens	all components subscribed to the same events are notified in the same order. all components subscribed to the same events are notified according to specific priorities. all components subscribed to the same events are notified in random order
Adding components	new components are added through the subscription to an event. New components are added through the publishing an event. both
Removing components	components are removed by unsubscribing from all events. components are removed if they didn't publish any event in T time unite and unsubscribed from all event. The real dispatcher is created choosing one option for the parameters.

Components:

Either environment or components that consist of data and operation can publish events. Events have a structure including a name, parameters, and other attributes such as event priorities, timestamps, etc Börger and Stärk, (2003). So events are modeled by concept of Location Borger, (2005) in ASM. A function signature and its indicated parameters and the value of the function with those parameters construct a location. With particular parameter values (t1, . . ., tn), the location for the function f(x1,. . ., xn) is (f(x1, . . ., xn), (t1, . . ., tn)), or more concisely f(t1, . . ., tn). The value of the location is the value of f(t1, . . ., tn).
For modeling components following dimensions must be specified correctly:

Binding:

in this approach each component has an event pool. The dispatcher send event into that event pool. So the event pool consists of events that trigger the methods of component in other words it includes events that the component subscribed to them. The event pool support two operations Store(event) and Remove(event). Each component has only one event pool for incoming event. There is no outgoing event pool. A event pool is an abstract data structure. It may be a queue or a stack or other data structure.

Event announcement is specified by an ASM rule :

Announceto (A event pool: EventPoolName ,Event: Eventname)

This rule checks the destination by event pool name. Since the dispatcher is a destination of all events, an update rule is used for announcing event:

```
Rule EventAnnouncement program
    Eventname(t1 ... tn)
```

When an event is announced the dispatcher adds it to appropriate event pool data structures (subscriber).

Subscription/ un subscription to/from events

```
Rule un/subscription program
Case SubSignal of
     Subscription→
            SubscribeTo(anEvent)
    Unsubscription→
             UnsubscribeFrom(anEvent)
```

If a component subscribe for any events, no events are added to its event pool data structure.

Event invocation*:* choosing mechanism from event pool isn't defined and a *choose rule* is used for describing it.

Choose x with do R

is a choosing mechanism that can be defined by designer.

```
    Rule EventInvocation program
 Choos event with interested priority do EventRule
```

Some times choosing an appropriate data structure for event pool satisfies the choosing mechanism and there is no need to additional choosing mechanism. Invocation of an event in a method will be blocked until the event pool contains matching event.

Invocation acknowledgment is specified by shared binary state variables that are written by component methods and read by event pool.

Ack(event name)=true/false

All events in event pool with true Ack are removed.

```
Rule Remove program
  Forall event with Ack=true do Remove(event)
```

Shared variables that are supported by most event systems, are used for relations between methods. A shared variable can be updated by both the ASM machines and its environment, and can be read by both.

Finally transition between two abstract states is specified by transition rules. A transition may arise from a component method or environment.

A state is called disabled if it has no transitions and doesn't change.

```
Component program =
    Un/subscription program
    EventAnnouncment program
    EventInvocation program
    Remove program
```

Components rules execute in parallel. An execution is a finite sequence of transitions that the final state is disabled.

Validation:

In this paper we use Spec Explorer for validation, model-based testing and Model exploration. Spec Explorer is a Microsoft tool that powerfully explores models in order to discover all the potential behaviors they define, and present a graphical view of the result.

The main functionality of Spec Explorer from user's perspective is to provide an integrated tool environment to develop models, to explore and validate models, to generate tests from models, and to execute tests against an implementation under test. A central part of the functionality of Spec Explorer is to visualize finite state machines generated from models as graphs. This is a very effective way to validate models and to understand their behavior, prior to test case generation David Garlan, (2003). Spec Explorer handles non determinism by distinguishing between *controllable* actions invoked by the tester and *observable* actions that are outside of the tester's control. It also handles infinite states spaces by separating the description of the *model* state space which may be infinite and finiteness provided by *user scenarios* and *test cases* Campbell *et al.,* (2005). A model program defines the state variables and update rules of an abstract state machine. The states of the machine are first-order structures that present variable values in each step. The transitions between states are invocations of the model program's methods that satisfy the given state-based *preconditions*. The tool *explores* the machine's states and transitions with techniques similar to those of explicit state model checkers. This process results in a finite graph that is a representative subset of model states and transitions Börger and Stärk, (2002). Finally, Spec Explorer produces test cases for the explored behavior that may be against the system under test to check the adaptation of real and predicted behavior.

A model program declares a finite set of action methods and a set of state variables. A state is given by the values (or interpretations) of the state vocabulary symbols that occur in the model program. The value of a state variable may change as a result of program execution.

A nullary state variable is a normal program variable that may be updated. A unary state variable represents either an instance field of a class (by mapping from object identities to the value of that field) or a dynamic universe of objects that have been created during program execution.

Each action method *m*, with variables **x** as its formal input parameters, is associated with a state based Boolean expression *Prem*(**x**) called the *precondition* of *m*. The execution of *m* in a given state *s* and with given actual parameters **v**, produces a result state where some of the state variables have changed.

A model program can be written in a high level specification language such as AsmL Börger, (2002) or Spec (http://msdn.microsoft.com).

The model automaton defined by a model program include states, transitions and concept of accepting states, but they extend traditional model-based testing by admitting open systems whose transitions are not just a subset of the specification's transitions and by treating states as first-order structures of mathematical logic.

We say that a model and an implementation under test (IUT) *conform* if the following conditions are met: The IUT must be able to perform all transitions outgoing from an active state. The IUT must produce no transitions other than those outgoing from a passive state. Every test must terminate in an accepting state Campbell *et al.,* (2005).

Case Study:

To demonstrate our proposed approach we use set&counter example as a case study.
The system includes two loosely coupled components: a set and counter. Elements are added/removed from it.

The set upon the receipt of insert(element) or delete(element) events from the environment; when the insertion or deletion is successful (insertion fails if set is full; deletion fails if element is not in the set), the set announces update(ins) or update(del) events.

Asm model

Set Component program

Subscribe(ins(element),del(element))
Update_ins()
Update_del()
*R1:**Choos** event **with** fifo priority **do** event rule*
*R2:**Forall** event **with** ack==true **do** remove(event)*

```
Counter Component program
Subscribe(updat_ins(),update_del())
R1:Choos event with No priority  do event rule
R2:Forall event with ack==true do remove(event)
User Component program
ins(element)
del(element)
Transition Rule :
Ins(element)=
    If  not setfull() then
    Insert element in the set seq update_ins()
    Else
    Insert fail()
    Ack(ins(element))=true;
Del(element)=
    If exist(element) then
    Delet( element) seq update_del()
    Else  delete fail()
    Ack(del(element))
Update_ins()=
    Let counter+=1 then ack(update_ins)=true
Update_del()=
    Let counter-=1 then ack(update_del)=true
Main rule :
    do in-parallel
                                set.R1 seq set.R2
                                counter.R1 seq counter.R2
    main rule ;
```

The Model Program:

The set is implemented by a set of integers and several methods on the set, which inserts, and deletes elements. The system is un-initialized at start-up. After initialization, several threads and an empty set are created. Each thread has a unique id and can be in a mode of Inactive, Adding or Deleting. The elements of the set are integers.

Model Program: Definitions& Thread Structure & Initialize function

```
enum SystemMode {Uninitialized, Initialized};
SystemMode systemMode = SystemMode.Uninitialized;
structure ThreadMode{
  virtual public string Kind(){return "";}
  case Inactive
  {
    override public string Kind(){return "Inactive";}
  }
  case Adding
  {
    public int elem;
    override public string Kind(){return "Adding";}
  }
  case Deleting
  {
    public int elem;
    override public string Kind(){return "Deleting";}
  }
}
```

```
type ThreadId = int;
Map<ThreadId,ThreadMode> threads = Map{};
SET<int> content = SET{};
int maxContentSize = 0;
int counter = 0;
type Element = int;
void Initialize(int maxNrOfThreads, int maxNrOfElements)
  requires systemMode == SystemMode.Uninitialized;
{
  threads = Map{tid in Seq{0..maxNrOfThreads-1} ; <tid,ThreadMode.Inactive()>};
  systemMode = SystemMode.Initialized;
  maxContentSize = maxNrOfElements;
}
```

Adding an element to the set is initiated by the Add function, which can create a slave thread to actually commit the action. The slave thread in turn indicates the success or failure of adding the element by calling either Update-Add or AddFailed, respectively, depending on whether the set's capacity has been reached.

Model Program: Add functions

```
void Add(ThreadId tid, Element elem)
 requires systemMode == SystemMode.Initialized;
 requires threads[tid] is ThreadMode.Inactive;
{
   threads[tid] = ThreadMode.Adding(elem);
}
[Action]
void Update-add(ThreadId tid)
  requires systemMode == SystemMode.Initialized;
  requires threads[tid] is ThreadMode.Adding;
  requires ContentSize < maxContentSize;
{
  content += SET{(threads[tid] as    ThreadMode.Adding).elem};
  threads[tid] = ThreadMode.Inactive();
  counter+=1
}
void AddFailed(ThreadId tid)
  requires systemMode == SystemMode.Initialized;
  requires threads[tid] is ThreadMode.Adding;
  requires ContentSize >= maxContentSize;
{
  threads[tid] = ThreadMode.Inactive();
}
```

The following functions delete only one element in the set with value elem, so that there might be other elements with the same value left in the set after calling the functions. The master thread creates a slave thread in the Delete function, and the slave thread calls the update_del function when the actual deletion happens.

Model Program: Delete Functions

```
void Delete(ThreadId tid, Element elem)
  requires systemMode == SystemMode.Initialized;
  requires threads[tid] is ThreadMode.Inactive;
{
  threads[tid] = ThreadMode.Deleting(elem);
}

void Update_del(ThreadId tid)
  requires systemMode == SystemMode.Initialized;
  requires threads[tid] is ThreadMode.Deleting;
  requires exists {int k in (0:maxcontentSize);content[k]==elem };
```

```
{
  ThreadMode.Deleting a = threads[tid] as ThreadMode.Deleting;
  content -= SET{a.elem};
  threads[tid] =  ThreadMode.Inactive();
  cuonter-=1
}
```

GetState returns the contents in the set (in a string) only in the states where no adding/deleting/looking up actions are called. ***ContentSize*** is a helper function used to avoid a bug in the set implementation.

Model Program: GetState() & ContentSize Functions

```
SET<int> GetState()
requires systemMode == SystemMode.Initialized;
requires Forall{tid in threads;
                threads[tid] is ThreadMode.Inactive};
{
  return content;
}

int ContentSize
{
  get{
    int res = 0;
    foreach (x in content.Keys) res += content[x];
    return res;
  }
}
```

Test Configuration Setting:

Nr of threads. Threads are given ids 0 ... NrOfThreads - 1.

```
Seq<int> threadIds { get {return Seq{0..threads.Size-1};}}
```

Convert the implementation state, given as a collection returned by Test.GetContent(), into a set. At stable states this set has to coincide with the content of the model set.

```
SET<int> GetImplState()
{
  return SET{x in Test.GetContent(); (int)x};
}
```

Accepting states are all states where the SET is nonempty and all threads are inactive.

```
bool IsAccepting
{get
{return content.Size > 0 &&
Forall{t in threads; threads[t] is ThreadMode.Inactive};}
}
```

Must wait for events if no thread is inactive.

```
bool MustWaitForEvents
{
get
{return Forall{t in threads; ! (threads[t] is ThreadMode.Inactive)};
}
}
```

Scenario:

Initialize and keep adding elements to the set so that the set is full. The Initialize method should be disabled if this scenario action is enabled.

```
void FillSET(int maxNrOfThreads, int maxNrOfElements)
  requires systemMode == SystemMode.Uninitialized;
{
  Initialize(maxNrOfThreads,maxNrOfElements);
  for (int i = 1; i <= maxNrOfElements; i++)
  {
     Add(0, i*10); //let thread 0 add the element
     Update-add(0);      //must be observed before adding the next element
  }
}
```

After running the model Spec Explore presents a graphical view of the result as a graph. Trough the generating an running test case we can validate and undrastand the the model behavior and we will realiz that a model and an implementation under test (in C# orVB) conform or not.

Conclusion:

The approach that is proposed for developing software using publish/subscribe architecture applies ASMs for modeling and Spec Explorer for validation. ASMs provide a mathematical framework for system modeling, while Spec Explorer is a model-based testing tool. Together, they form a powerful tool for checking systems. Also using this combination is uncomplicated for practitioner.

REFERENCES

Barrett, D.J., L.A. Clarke, P.L. Tarr and A.E. Wise, 1996. A framework for event-based software integration. ACM Transactions on Software Engineering and Methodology, 5(4): 378-421.

Barnett, M., R. Leino and W. Schulte, 2005. The Spec# programming system: An overview. In M. Huisman, editor, Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004, 3362: 49-69.

Baresi, L., C. Ghezzi, L. Mottola, 2006. Towards Fine-Grained Automated Verification of Publish-Subscribe Architectures. FORTE pp: 131-135.

Börger, E. and R. Stärk, 2003. Abstract State Machines: A Method for High-Level System Design and Analysis. Springer Verlag,

Börger, E., 2002.The Origins and the Development of the ASM Method for High Level System Design and Analysis. J.UCS(Journal of Universal Computer Science), 8(1):2-74.

Borger, E., 2005. The ASM Method for System Design and Analysis. A Tutorial Introduction. In Proceedings of FroCos' pp: 264-283.

Campbell, C., W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes, 2005. Testing Concurrent Object-Oriented Systems with Spec Explorer, in FM, Springer.

David Garlan, 2003.Serge Khersonsky and Jung Soo Kim. Model Checking Publish-Subscribe Systems. In Proceedings of The 10th International SPIN Workshop on Model Checking of Software (SPIN 03), Portland, OR,

Dingel, J., D. Garlan, S. Jha and D. Notkin, 1998. Reasoning about Implicit Invocation. In Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6), Lake Buena Vista, Florida, ACM.

http://msdn.microsoft.com

Gurevich, Y., B. Rossman and W. Schulte, 2005. Semantic essence of AsmL. TheoreticalComputer Science, 343(3): 370-412.

Sun Microsystems. JavaBeans. http://java.sun.com/products/javabeans.

Taylor, R.N., N. Medvidovic, K.M. Anderson, J.E. James Whitehead, J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow, 1996. A component- and message-based architectural style for GUI software. IEEE Transactions on Software Engineering, 22(6): 390-406.

Veanes, M., C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, 2008. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer, in Formal Methods and Testing, Springer Verlag,