Object Synchronization in XMF

Xactium Ltd.

August 30, 2004

1 Introduction

XMF is an environment for language engineering and constructing executable models of systems. A key aspect of executable modelling is *dealing with change*. Systems can often be viewed in terms of events that occur during a life-cycle. A model should be able to represent a system life-cycle and its corresponding events; an engineer should be able to use a modelling environment to investigate the effects of injecting events into specific system states.

Given an imperative language, it is always possible to simulate system events by pre-plumbing event handlers throughout an executable model. Where an event occurs at the interface to a system, the event can be passed to the appropriate target via procedure calls. This approach flies in the face of modelling where we seek to abstract away from the detail of execution as far as possible.

At both the modelling and application development levels we would like to separate out the aspects of a system as far as possible. Many modern implementation platforms present a system in terms of business logic, distribution protocols, user interface, data management etc. In all aspects we would seek to identify and assimilate the key definition and execution features and thereby present the user with a *declarative* interface to these mechanisms.

With respect to modelling events there are two key areas where we can make significant abstractions. These are: data synchronization and condition-action rules. The requirements for the former occur widely where we have multiple representations for the same data (for example abstract and concrete syntax, external and internal databases, etc). The requirements for latter are well understood from the business rules community where expertise from the application domain can be captured succinctly in the form of human-readable chunks of knowledge.

XMF provides a language called XSync that can be used to construct event driven condition-action rules. XSync rules can be thought of as monitoring a collection of objects. XSync allows you to abstract away from the details of the monitoring and action execution mechanisms; XSync rules are separate from the application model and do not pollute the application execution model.

XSync rules can be used for a variety of modelling tasks. Rules may be used to maintain an invariant in a model, for example by synchronizing the state of multiple objects. Rules may be used to monitor invalid system states and provide warning messages or request the user to fix the problem before continuing. Rules may be used to model event-driven execution where events occur as a result of particular situations arising in the system state. Rules may be used to capture and independently model chunks of application knowledge.

This doument describes XSync and provides a number of examples of its use.

2 A Simple Synchronizer

Suppose that we have a class of objects that we want to monitor. For example, a water tank has a water level and a water temperature:

```
context Root
  @Class WaterTank
    @Attribute level : Integer end
    @Attribute temp : Integer end
    @Operation setLevel(l:Integer)
      self.level := l
    end
    @Operation setTemp(t:Integer)
      self.temp := t
    end
  end
```

We want to perform an action whenever the water tank changes state such that the state represents a dangerous situation. We could do this by requiring that all state changes to a water tank are performed through a given interface and perform the checks in the implementation of the state change operations in the interface. However, this mixes implementation concerns with the invariant that action should be taken whenever the dangerous situation occurs. We would like to separate out the actions that deal with state and the conditions and actions that represent the invariant. The separation is important because when we deploy the model (either by mechanical means or by human translation) the implementation mechanisms used to achieve the state implementation and the invariant implementation may be very different. By modelling them separately, they do not become confused and difficult to identify later on.

This separation can be achieved using XSync. The invariant is modelled as an object synchronizer with two rules:

```
@XSync autorun
  @Scope type WaterTank end
  @Rule checkLevel 1
    WaterTank[level=l] when l > 100
  do format(stdout,"Water level has gone dangerously high!~%")
  end
  @Rule checkTemp 1
    WaterTank[temp=t] when t < 10
  do format(stdout,"Water temp has gone dangerously low!~%")
  end
end
```

The rule `checkLevel` has a condition that is satisfied whenever the state of a water tank changes such that the level is greater than 100. In the example the action is to print out a warning message, however XSync permis any action to take place at this point. The rule `checkTemp` is similar and monitors the temperature going low.

A synchronizer has a `Scope` part which defines the collection of objects that is to be monitored. The declaration `type WaterTank` defines that all instances of the class `WaterTank` (existing and future) will be monitored.

You can either control the execution of a synchronizer yourself or let XMF execute the rules as they become enabled. In this case, the declaration `autorun` states that this synchronizer is immediately active and as soon as a rule becomes enabled its action part will be performed.

Consider the creation of a water tank and modification of its state:

```
w := WaterTank();
Water temp has gone dangerously low!
w.temp := 100;
w.level := 200;
Water level has gone dangerously high!
```

As you can see from above, the synchronizer has automatically executed the rules in response to the changes in state of the new object.

3 Controlling an Object Synchronizer

4 Synchronizing Objects

Models often present multiple overlapping views of the same information. It is therefore important to have modelling facilities that synchronize objects by propagating changes. A synchronization facility must detect changes and keep multiple representations up to date.

It is possible to synchronize objects by requiring that state change is performed through a standard interface; each update operation explicitly runs over all the synchronized objects and makes the necessary modifications. This is undesirable since it pollutes the logic of the object models with the details of synchronization technology.

XSync uses object synchronizers to achieve a separation of concerns between the logic of executable object models and technology used to keep the objects in sync. XSync rules are used to detect and propagate changes to objects in a synchronization collection. Typically, the synchronizer will be set to `autorun` so that the changes are propagated automatically.

For example, the following operation can be used to synchronize the names of two named elements. It shows how an object synchronizer can be created *in a context* containing bound variables (in this case the parameters of the operation):

```
@Operation sameName(n1:NamedElement,n2:NamedElement)
  @XSync autorun
    @Scope
      Set{n1,n2}
    end
    @Rule SetName 1
      x1 = NamedElement[name = name1] when x1 = n1;
      x2 = NamedElement[name = name2] when x2 = n2 and name1 <> name2
    do if timestamp(x1,"name") > timestamp(x2,"name")
       then x2.name := x1.name
       else x1.name := x2.name
       end
    end
  end
end
```

The scope of the synchronizer is the pair of named elements supplied as arguments to the operation. The rule `SetName` detects changes to either of the two objects. Since rule patterns are general with respect to objects of a given type, the side condition on each rule pattern fixes the matched object to either `n1` or `n2`.

The rule matches when either object state has changed and the names are not the same. In the body of a rule the operation `timestamp` may be used to compare the relative time at which slots have been updated. In the case of `SetName` we check to see which of the objects have been changed most recently and modify the name of the other object.

The following is a transcript of an XMF session:

```
[1] XMF> x1 := Class("A");
<NameSpace Root>
[1] XMF> x2 := Class("B");
<NameSpace Root>
[1] XMF> x3 := Class("C");
<NameSpace Root>
[1] XMF> sameName(x1,x2);
<a Net>
[1] XMF> sameName(x3,x2);
<a Net>
[1] XMF> c1;
<Class W>
```

```
[1] XMF> x1;
<Class B>
[1] XMF> x2;
<Class B>
[1] XMF> x3;
<Class B>
[1] XMF> x3.setName("O");
false
[1] XMF> x1;
<Class O>
[1] XMF> x2;
<Class O>
[1] XMF> x3;
<Class O>
[1] XMF>
```

5 Event Driven Modelling

Systems are often driven by externally generated events. For example, *we start to process the raw materials when the customer order is received.* In addition to external events, knowledge about how a system executes is often conveniently expressed in terms of internal events that occur when the system enters particular composite states. For example *if the contents of tank X reaches the safety limit and valve Y is open the redirect the flow to tank Z.*

When modelling such systems we do not wish to be bothered with all the machinery that detects situations when they arise and that ensures events are received by their intended targets. We simply want to state the information and let the executable models do the rest. This section provides a simple example of how XSync can be used to construct an event driven model. Once the model is created it is used to simulate the system.

Consider a factory that manufactures products. A product consists of a collection of sub-components (each of which is a product) that must be constructed and assembled in a specified order. Each product is allocated to a department of the factory that specializes in constructing and assembling the product from its constituent components. Each product takes a given time to construct. Orders arrive at the factory, are immediately placed on the input queue of the appropriate department; assembled products are placed on the delivery queue of the factory.

Time is important to the simulation of the factory. We are not interested in *real-time*, just in a simple clock that can be used to synchronize the start and end times of construction and assembly:

```
context Root
  @Class Clock
    @Attribute time : Integer end
    @Constructor(time) ! end
    @Operation tick() self.time := time + 1 end
  end
```

Each product is of a given type. The type tells us how long the product takes to manufacture (let's assume that assembly is modelled as the time taken to manufacture a non-atomic component).

```
context Root
  @Class ProductType
    @Attribute name : String end
    @Attribute manufacture : Integer end
    @Attribute components : Set(ProductType) (+) end
    @Constructor(name,manufacture) ! end
  end
```

Each product has a type, an ordered collection of sub-components, a start and completion time, and a status. The status indicates whether the product is waiting to be constructed and assembled, whether it is being processed or whether it is completed.

A product is scheduled by supplying its `start` operation with the time at which its manufacture will start. Since each of the sub-products must be constructed before a composite product can be assembled, `start` schedules each sub-component and increments the earliest time at which the composite component can start by the time taken for each sub-component. Finally, each component is scheduled by setting its start and completion time (note `start` returns the completion time so that parent products know when a sub-component will finish):

```
context Root
  @Class Product
    @Attribute type : ProductType end
    @Attribute components : Seq(Product) (+) end
    @Attribute status : String = "WAITING" end
    @Attribute start : Integer end
    @Attribute completion : Integer end
    @Constructor(type) end
    @Constructor(type,components) end
    @Constructor(type,status,start,completion,components) ! end
    @Operation completed()
      self.status := "COMPLETED"
    end
    @Operation start(time:Integer):Integer
      self.status := "PROCESSING";
      @For c in components do
        time := c.start(time)
      end;
      self.start := time;
      self.completion := time + type.manufacture;
      time + type.manufacture
    end
  end
```

Each factory department specializes in manufacturing a specific type of product. A department has a queue of products waiting to be processed:

```
context Root
  @Class Department
    @Attribute type : ProductType end
    @Attribute queue : Seq(Product) (+,-) end
    @Constructor(type) end
    @Constructor(type,queue) ! end
    @Operation processOrder(o:Order)
      if o.type = type
      then self.addToQueue(o)
      end
    end
  end
```

A factory has a sequence of orders waiting to be allocated to departments, has a sequence of completed products and has a clock that is used to synchronize the manufacturing process. A factory processes the orders in its input queue by removing them from the input queue and broadcasting them to each of the departments. Assuming that there is only one department specializing in constructing each product type then each order will be added to the input queue for the appropriate department:

```
context Root
  @Class Factory
    @Attribute orders : Seq(Product) (+,-) end
    @Attribute completed : Seq(Product) (+,-) end
    @Attribute departments : Set(Department) end
    @Attribute clock : Clock = Clock(0) end
    @Constructor(clock,orders,departments,completed) ! end
    @Constructor(departments) end
    @Operation processOrders()
      @For o in orders do
        self.deleteFromOrders(o);
        @For d in departments do
          d.processOrder(o)
        end
      end
    end
  end
```

The execution of the factory is determined by events occurring: external events place orders on the factory input queue and various internal events occur when the factory enters appropriate system states. The execution can be distilled into a collection of rules. These rules are defined as an object synchronizer as follows. Comments on the definition are included:

```
factoryController :=
@XSync
  @Scope
    type Factory,Department,Clock,Product
  end
```

An object synchronizer must specify its scope. The factory controller is scoped over several types. Each type is a class (and may be specified as a path to the type). the synchronizer applies to any instances of the classes that are currently available an also applies to any instances of the classes that are subsequently constructed.

In the case of the factory controller, the rules will involve conditions in terms of the factory (when orders arrive), departments (when orders are started and completed), the factory clock (controlling when products are scheduled) and products (changing state as they are processed).

```
  @Rule ProcessOrders 1
    f = Factory[orders = O] when not O->isEmpty
  do f.processOrders()
  end
```

The rule named `ProcessOrders` detects when orders are received at the factory. It will be enabled by any event that changes the state of the factory such that the collection of orders becomes non empty. When an enabled instance of the rule is activated, the result is to process the orders via the `processOrders` operation defined by `Factory`.

```
  @Rule ProductCompleted 2
    Clock[time=t];
    p = Product[completion=t,status="PROCESSING"]
  do p.completed()
  end
```

The rule named `ProductCompleted` detects when the factory time matches the completion time of any product that is currently being processed. When an enabled instance of the rule is activated. The `completed` operation of `Product` updates the status of the product. Note that the precedence of the rule is `2` which causes it to be activated in preference to other enabled rules with lower precedences. This ensures that products are completed.

```
  @Rule Tick 1
    Factory[orders = O] when O->isEmpty;
    Product[status="PROCESSING"];
    c = Clock[ ]
  do c.tick()
  end
```

The `Tick` rule advances the factory clock. Note that the precedence of this rule is tt 1 ensuring that it does not become activated instead of the `ProductCompleted` rule.

```
  @Rule MoveToWarehouse 1
    f = Factory[ ];
    d = Department[queue=Q] when not Q->isEmpty;
    p = Product[status="COMPLETED"] when Q->includes(p)
  do
    f.addToCompleted(p);
    d.deleteFromQueue(p)
  end
```

The `MoveToWarehouse` rule detects when a product at the head of a department's processing queue becomes completed. The product is removed from the department and placed in the factory warehouse (modelled as the collection of `completed` products in `Factory`).

```
  @Rule StartProcessing 1
    Department[queue=Q] when not Q->isEmpty;
    p = Product[status="WAITING"];
    Clock[time=t]
  do if p = Q->head then p.start(t) end
  end
end;
```

The `StartProcessing` rule detects when a department has a product that is ready for processing. The process at the head of the departments queue is selected for processing and started. The current time `t` is used to set the start and completed time for the product and its components.

The following operation `runFactory` simulates the operation of the factory with two products and departments. Since the factory controller is not defined to run automatically (simulations are usually performed explicitly) the operation `run` is used to start the simulation.

```
context Root
  @Operation runFactory()
    let X = ProductType("X",10);
        Y = ProductType("Y",20);
        Z = ProductType("Z",15)
    in Root::f :=Factory(Set{Department(X),Department(Y)});
       f.addToOrders(Product(X,Seq{Product(X,Seq{Product(Y)})}));
       f.addToOrders(Product(Y,Seq{Product(Z)}));
       factoryController.run()
    end
  end
```

A XSync Syntax

```
XSync ::= [ 'autorun' ] Exp*.
Scope ::= Dec (';' Dec)*.
Dec ::= 'type' ScopedTypes | LogicalExp.
ScopedTypes ::= LogicalExp (',' LogicalExp)*.
Rule ::= Name Int SyncPatterns 'do' Exp.
SyncPatterns ::= SyncPattern (';' SyncPattern)*.
SyncPattern ::= ObjectPattern ['when' LogicalExp].
ObjectPattern ::= BindingObjectPattern | BasicObjectPattern.
BindingObjectPattern ::= ['~'] Name '=' ObjectPath '[' Slots ']'.
BasicObjectPattern ::= ObjectPath '[' Slots ']'.
ObjectPath ::= Name ('::' Name)*.
Slots ::= Slot (',' Slot)*.
Slot ::= Name '=' Value.
Value ::= Name | Str | Int | Bool.
```