A Formal Model for Some Behavioural Features of Analysis Patterns

Agustina Buccella and Alejandra Cechich

Department of Informatics and Statistics - University of Comahue
Buenos Aires 1400, 8300 Neuquén, Argentina
E-mail: acechich@uncoma.edu.ar

Abstract

It is commonly said that a pattern has four essential parts: a statement of the context where the pattern is useful, the problem that the pattern addresses, the forces that play in forming a solution, and the solution that resolves that forces. This form underlies many published patterns, including analysis patterns. They show a number of highly generic processes that cut across traditional boundaries of system development and business engineering. However, patterns are invariably described informally in the literature, generally using natural language together some sort of graphical notation. A formal model of the semantic statements of analysis patterns has been proposed by reusing some of the properties formalised for GoF patterns. In this paper, we present a formal model of some behavioural properties of analysis patterns, and we illustrate using an example how an instantiation can be done. We also briefly discuss future work which will extend the model to include more behavioural properties.

Key words: Object-Oriented design patterns – Formal Methods – Analysis Patterns

Introduction

In object-oriented design methods, design patterns are becoming increasingly popular as a way of identifying and abstracting the key aspects of commonly occurring design structures, and thus as a basis for reusable object-oriented design. The GoF patterns [3] provide an infrastructure that defines the components to be included in each solution and how they should be interpreted. However, the GoF patterns and their properties are specified using a combination of graphical notation and natural language, together with sample code in some object-oriented programming language. The description of the patterns is thus largely informal, which makes it difficult to be certain that the patterns themselves are meaningful and contain no inconsistencies and, more importantly, that the pattern is being used correctly and consistently by developers. It is therefore extremely difficult to give any meaningful certification of the correctness of software developed using patterns.

In order to alleviate these problems, a more formal basis for patterns is needed. Our first work in this direction has presented a preliminary formal model of the essential elements of GoF patterns, which can serve as the basis for checking the internal consistency of pattern structures [2].

However, object-oriented modelling based on patterns is more than GoF design patterns. Patterns can be used in system analysis as well as in software design. In analysis, the type models are important. They define the "language of the business". These models thus provide a way of coming up with useful concepts that underlie a great deal of the process modelling. For example, the concept of accountability has proven very useful in modelling confidentiality policies in health care.

To accomplish component reuse for information systems, Fowler's patterns [1] describe alternative ways of modelling a situation. A small number of highly generic processes that cut across traditional boundaries of system development and business engineering constitute the analysis patterns introduced by Fowler – for example, the diagnosis and treatment model, or the accounting and inventory model. Many diverse business can use a set of very similar abstract process models, allowing frameworks being organised along abstract conceptual processes. But as GoF patterns, analysis patterns are depicted using an informal notation and a more formal model is needed. A first piece of work formalising the semantic statements of analysis patterns has been presented by reusing our formal model of design patterns [7].

However, object-oriented design also includes behavioural properties, as presented in [1], which cannot be expressed in our current model of analysis patterns – for example, events and triggers should be specified as part of the behaviour associated to a pattern.

In this paper we present an extension of our previous formal model of analysis patterns to allow us to specify some behavioural features. We have focused on event diagrams, as presented in [4], which generally abstract the main behavioural elements of a pattern. In Section 2 of the paper we introduce the basic notation of event diagrams of analysis patterns and we present an example. Section 3 presents our formal model (written in RSL [6]) and formalises constraints on various components applying them to the previous example. Future work will modify and extend the model to include other behavioural properties of analysis patterns. This is discussed briefly in the final section of this paper.

2. Techniques and Notations for Events in Analysis Patterns

Interaction diagrams show how several objects collaborate to get something done. An interaction diagram has a number of vertical lines that represent objects. Arrows between the lines represent messages sent between objects, with sequence indicated by progression down the paper. Interaction diagrams are widely used and simple to follow.

Event diagrams (Figure 1) are another form of behavioural model. Although they are more complex than interaction diagrams, they do allow complete control to be specified. They also are able to express parallel behaviour, which is very useful in business modelling. The boxes on an event diagram represent operations that complete by signalling an event. A trigger rule indicates that an event triggers an operation. Parallelism appears when an event type has more than one trigger rule defined on it. A label on the line indicates what collection is being iterated over.

If a trigger rule leads into an operation via a control condition, the operation is only invoked if the control condition (a Boolean expression) evaluates to true. The control condition is often used to synchronise parallel threads. Each thread triggers the condition, which is designed to be true only at the appropriate synchronisation point. Two common control conditions are the *and* condition and the *z* condition. The *and* condition is true only when all incoming trigger rules have fired once. It is shown by a *&* in the diamond. The *z* condition is true whenever there are no operations on the diagram that are triggered to run, that is, when all is quiet and the diagram has gone to sleep. It is shown by a z in the diamond (as in zzzzzz). A z condition is often used at the end of the diagram to synchronise the end of the diagram.

The other conditional logic is that of the partition, as on operations. The event is subtyped depending on the outcome of the operation. A trigger rule can be placed on the supertype event to indicate a trigger that is fired whatever the outcome. The partition works the same way as in structural models. An event can have many partitions defined on it, a partition can have any number of events within it, and partitions can be defined on top of each other to any desired depth. Any event will be an instance of only one event type from each partition.

Event diagrams are conceptual in that they only say how some process works, not which objects carry out the process. Thus they complement interaction diagrams very well.

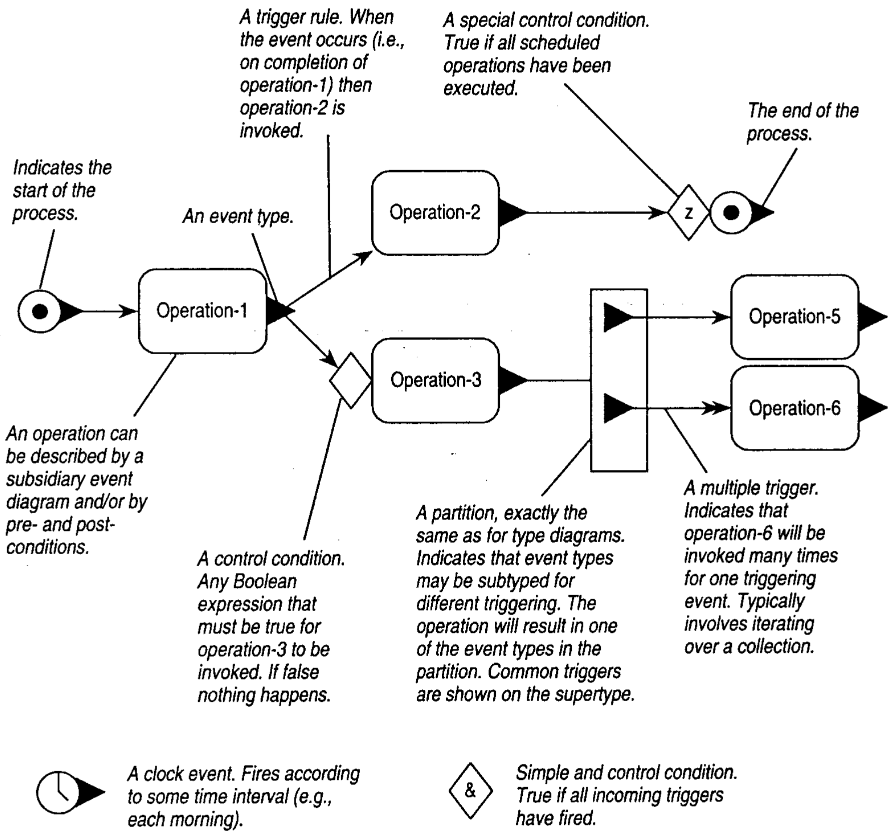
A trigger rule. When the event occurs (i.e., on completion of operation-1) then operation-2 is invoked.
A special control condition. True if all scheduled operations have been executed.
The end of the process.
Indicates the start of the process.
An event type.
Operation-2
z
Operation-1
Operation-3
Operation-5
Operation-6
An operation can be described by a subsidiary event diagram and/or by pre- and post-conditions.
A control condition. Any Boolean expression that must be true for operation-3 to be invoked. If false nothing happens.
A partition, exactly the same as for type diagrams. Indicates that event types may be subtyped for different triggering. The operation will result in one of the event types in the partition. Common triggers are shown on the supertype.
A multiple trigger. Indicates that operation-6 will be invoked many times for one triggering event. Typically involves iterating over a collection.
A clock event. Fires according to some time interval (e.g., each morning).
&
Simple and control condition. True if all incoming triggers have fired.

Figure 1: Event diagrams

The following example concentrates on the events of an observation as presented in [1]. Behaviours exist to create observations and to provide various ways of navigating associations to understand how those observations fit with other observations. Often a clinician has some path of observations that can be taken.

It is possible to sketch an outline of the process involved in observations. Whenever clinicians make observations, they consider the possibility of other associated observations. They use the associative functions they know to come up with a list of possible observation concepts that might be associated with the triggering observations. In Figure 2 the concurrent trigger rule is labeled “associated observation concept”. After the query, there is a control condition (evaluate proposal) before an observation is proposed. The query suggests possible observation concepts to look for based on the associative functions.

Figure 2 also includes additional triggers that arise from projections and active observations. The triggers to propose intervention work in a similar way to the previous case. This reinforces the fact that although any observation can lead to further observations being made, only active observations or projections (not hypotheses) lead to interventions. An intervention is an action which either intends or risks a change in state of the patient. A final trigger in Figure 2 shows how the appearance of an active observation can contradict other observations and thus lead to those observations being rejected. Once an observation is rejected, further observations which were supported by this observation must be reconsidered.

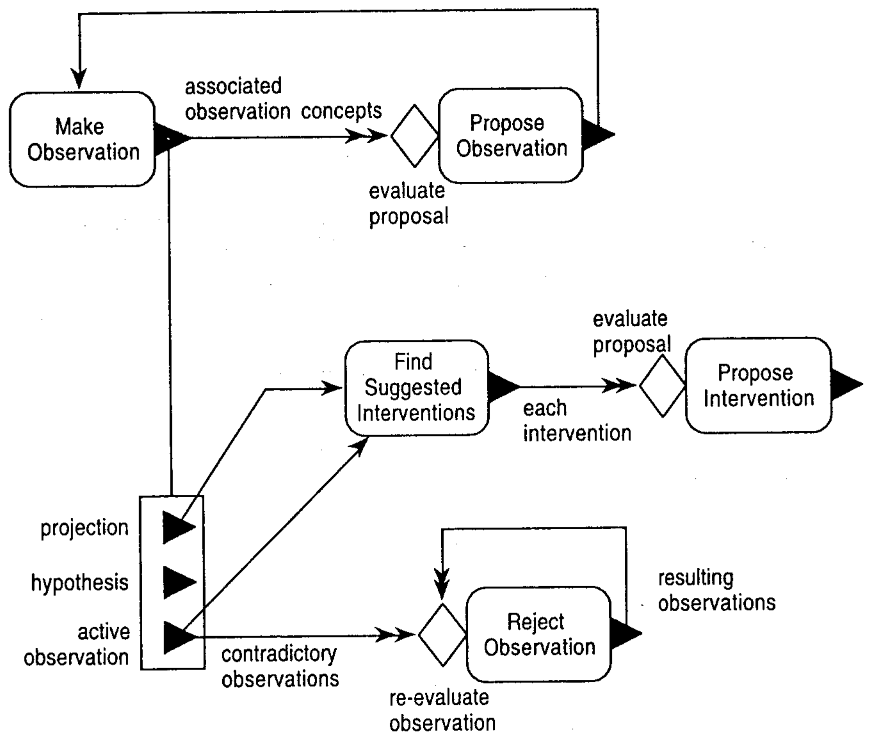

Figure 2: Event diagram for the process of working with observations

3. A Formal Model of Behavioural Properties in RSL

In the following specifications, knowledge about RSL language [5][6] or another similar formal language is assumed.

Our formal description is based on abstractions of the elements of an event diagram, which we deal with in Section 2. We give only an outline here and refer the reader to [8] for the full details.

The elements of an event diagram of an analysis pattern are defined in RSL as a product type composed of two elements: the event name, which is used to identify a particular event, and its structure. The structure is a product type composed of two elements: a set of well-formed operations, a set of well-formed inherited events.

```
Event_Diagram = G. Event_Name   x   Wf_Event_Structure
Event_Structure = E.Wf_Operation-set  x  I.Wf_Inherited_Event-set
```

An operation has a name and a set of associated events that are produced as a consequence of the actions performed by the operation. It also contains a type, which may be basic, clock or external clock, and a set of methods that represents the actions.

```
Operation ::
        operation_name: G.Name
        events: C.Wf_Event-set
        operation_type: G. Operation_Type
        operation_methods:   Operation_Methods-set

Operation_Methods::
        input_vble:G.Vble-set
        output_vble:G.Vble-set
        method_name: G.Name
```

Each operation method consists of a name and the variables it takes as input, and a set of output variables. Events are defined by its name, a time that abstractly represents the duration and synchronisation of the event, and a set of triggers. Finally, a trigger has a name, a type, which may be basic or multiple, and its related operation – sink operation – as well as a control condition.

```
Event ::
        event_name: G.Desc
        event_time: G.Time
        trigger: G.Trigger-set
```

```
Trigger ::
        trigger_mapping: G.Name
        trigger_type: G.Trigger_Type
        sink_operation: E.Wf_Operation
        control_cond: L.Wf_Control_Condition
```

Consistency conditions on an operation, for example that there must be at least one event associated to the operation are incorporated into the model by defining them as Boolean-valued functions (e.g. *always_one_event*) then constructing an RSL subtype which satisfies the conjunction of all such predicates (*is_wf_operation*).

```
always_one_event: Operation → Bool
always_one_event(o) ≡ events(o) ≠ {}

Wf_Operation = {| o: Operation • is_wf_operation |}
```

Other functions constrain some properties of the events inside an operation. For example, many events associated to an operation are executed at the same time (simultaneous events); events are identified by its name so there is no duplicated event names, and the events inside an operation are classified as “normal” meaning that an event has no partitions.

```
have_many_events: Operation → Bool
have_many_events(o) ≡
        (∀ e1: C.Wf_Event-set • e1 ⊆ events(o) ⇒ is_ simultaneous(e1))

no_duplicated_event: Operation → Bool
no_duplicated_event(o) ≡
        (∀ e1,e2: C.Wf_Event • e1 ∈ events(o) ∧ e2 ∈ events(o) ∧
                        event_name(e1) = event_name(e2) ⇒ e1= e2)

always_normal: Operation →  Bool
always_normal(o) ≡   (∀ e1: C.Wf_Event • e1 ∈ events(o) ⇒ is_event_normal (e1)),
```

Finally, a inherited event with partitions is defined by a pair of events that represents the supertype and the subtype respectively. Again we impose some well-formedness constraints on inherited events by defining a well-formed event as a subtype of the type *Inherited_Event*. The details of this specification are omitted for brevity.

There are also constraints which the operations and the inherited events in the structure must satisfy in order for the whole structure to be well-defined. For example, the pair of inherited events should invoke some operation of the structure.

There are other constraints on the model based on the particular responsibilities and roles of the various elements in the pattern. In analysing these, however, we notice that different elements often have responsibilities which are very similar at an abstract level. For example, the connection between operations of event diagrams all have essentially the same basic form: two operations connected by a relation, with a control statement having the responsibility of evaluating the triggering conditions.

In our model, we abstract these common responsibilities, which we call *common features*, in order to obtain a more general and reusable specification. We illustrate the abstraction by specifying some of the properties of the participants in the previous example. A module describing diagram's participants embodies an instantiation of each value definition applied on it. The instantiation of the example introduced in Section 2 is defined in RSL by the object EvObs, where some variant declarations specifying name types and description types are used to parameterise the behavioural statements.

```
Object :
 EvObs ::  class
   Name_Type = = MakeObservation | ProposeObservation | RejectObservation |
                 FindSuggestedInterventions | ProposeIntervention
   Desc_Type = = ObservationPartition | Projection | Hypothesis |  ActiveObservation

   end
```

One of the properties related to the behaviour constrains the relation between the *MakeObservation* operation and the *ProposeObservation* operation, as we see in Figure 2. To apply this property, a name instantiation and a description instantiation are needed. So, the general property *like_relation* constrains the relation between two operations to have a trigger associated to a special condition.

```
observation_relation: Wf_Event_Structure → Bool
observation_relation(s)≡
  like_relation(ED.MakeObservation, ED.ProposeObservation, G.EvaluateProposal,
  G.Normal)(s),


like_relation:
  Name_Type x Name_Type x  Name x Condition_Type → Wf_Event_Structure → Bool
like_relation(n1,n2,cn,ct)(o,e,i,r) ≡
              (∀o1,o2: Wf_Operation • o1 ∈ o ∧ o2 ∈ o ∧ name_type(operation_name(o1)) =
              n1 ∧ name_type(operation_name(o2)) = n2 ⇒ (∃r1: Wf_Trigger • r1 ∈ r ∧
              R.source_operation = o1 ∧ R.sink_operation = o2 ∧ R.condition_name = cn ∧
              R.condition_type=ct))
```

In similar way, the function *like_relation* can be reused to constrain another relations in the pattern. For example, the relation between *FindSuggestedInterventions* and *ProposeIntervention* has the same features as the previous instantiation: two connected operations with a triggering condition. The application of the instantiation in this case is the following:

```
intervention_relation: Wf_Event_Structure → Bool
intervention_relation(s)≡
  like_relation(ED.FindSuggestedInterventions, ED.ProposeIntervention, G.EvaluateProposal,
  G.Normal)(s),
```

Other properties can be specified in similar way. For example, the fact that there is a partition relating different conditions to a particular operation is defined using the general function *has_parent* applied to the roles *Projection* and *ObservationPartition* (the name assigned to the inherited event).

```
has_partition: S.Wf_Event_ Structure→Bool
has_partition(s)≡ has_parent(ED.Projection, ED.ObservationPartition)(s)

has_partition: S.Wf_Event_ Structure→Bool
has_partition(s)≡ has_parent(ED.Hypothesis, ED.ObservationPartition)(s)


has_partition: S.Wf_Event_ Structure→Bool
has_partition(s)≡ has_parent(ED.ActiveObservation., ED.ObservationPartition)(s)
```

```
has_parent: PN.Desc_Type x PN.Desc_Type → S.Wf_Event _Structure → Bool
has_parent (n1,n2)(o,e,i,r)≡
            (∀r1: C.Wf_Event • r1 ∈ e ∧ desc_type(C.event_name(r1)) = n1 ⇒
                  (∃i1: I.Wf_Inherited_Event • i1 ∈ i ∧
                        desc_type(C.event_name(I.supertype_event(i1))) = n2 ∧
                        I.subtype_event(i1) = r1 ∧ is_event_father(n2) ∧ is_event_children(n1)))
```

The *is_event_parent* and *is_event_children* functions are used to identify the pair of inherited events using a the values “parent” and “child” as a type.

A multiple invocation is shown for example in Figure 2 between *Make Observation* and *Propose Observation* and between *Find Suggested Interventions* and *Propose Intervention*. The following function *can_invoke_once* abstracts this common feature: when an operation whose role is parameterised by n1 (*MakeObservation* or *FindSuggestedInterventions*) evaluates its triggering condition true, there is a multiple trigger invoking the operation whose role is n2 (*ProposeIntervention* or *ProposeObservation*).

```
observation_invoke: S.Wf_Event_ Structure → Bool
observation_invoke(s) ≡
                  can_invoke_more(ED.MakeObservation, ED.ProposeObservation,
                  G.AssociatedObservationConcepts)(s),

observation_partition_invoke: S.Wf_Event _Structure → Bool
observation_partition_invoke(s)≡
                  can_invoke_more(ED.FindSuggestedInterventions, ED.ProposeIntervention,
                  G.EachIntervention)(s),

  can_invoke_more:PN.Name_Type x PN.Name_Type → S.Wf_Event _Structure → Bool
   can_invoke_more(n1,n2)(o,e,i,r)≡ (∀r1: R.Wf_Trigger • r1 ∈ r ∧
                name_type(E.operation_name(R.source_operation(r1))) = n1 ∧
                evaluate_condition(R.control_cond(L.condition_name(r1))) ⇒
                name_type(E.operation_name(R.sink_operation(r1))) = n2 ∧
                is_multiple_trigger(r1))
```

Other properties and full details of the instantiation can be found in [8].

Conclusions

In this paper, we have presented a formal model of some behavioural elements of Fowler's patterns written in RSL, and we have formalised various constraints related to its structure. We have modified and extended our previous formal model of analysis patterns to allow us to specify behavioural properties, and we have illustrated using one Fowler's example how this can be done.
This is another step towards formally verifying the consistency and correctness of the development of an object-oriented design using patterns. However, object-oriented design also includes other behavioural properties, as presented in [1], which cannot be expressed in our current model – for example, interaction diagrams.

In the next stage of our work, we are modifying our formal model to allow us to specify other behavioural properties of analysis patterns. Also including dynamic and concurrent executions will be considered.

Acknowledgements

The authors wish to thank the United Nations University / International Institute for Software Technology, for its support and guidance in this work.

References

[1] Fowler M.:
Analysis Patterns,
Addison-Wesley (1997)
[2] Cechich A. and Moore R.:
A Formal Basis for Object-Oriented Patterns:
6th Asia-Pacific Software Engineering Conference, Takamatsu, Japan, December 1999 (284-291)
[3] Gamma E., Helm R., Johnson R., and Vlissides J.:
Design Patterns - Elements of Reusable Object-Oriented Software,
Addison-Wesley (1995)
[4] James Martin and James Odell:
Object-Oriented Methods: A Foundation,
Prentice-Hall (1995)
[5] The RAISE Method Group:
The RAISE DEVELOPMENT METHOD,
Prentice Hall (1995)
[6] The RAISE Language Group:
The RAISE SPECIFICATION LANGUAGE
Prentice Hall (1992)
[7] Cechich A.:
A Formal Model for Semantic Statements of Analysis Patterns:
Submitted to CIC2000 – Congreso Internacional de Computación, México D.F.
[8] Buccella A. and Cechich A..:
A Formal Model for Analysis Patterns,
UNC/AIS Technical Report No 5, July 2000