A Use Case Interpretation of B AMN

Richard Shore

University of Teesside

Middlesbrough, Cleaveland TS1 3BA, UK

Abstract

In this paper we suggest methods to aid the development of specifications using the B method. We propose an object-oriented approach to specification, developing the structure through refinement. We start with a use case analysis to find the functionality of the system, and then use a number of objects to specify the use case in greater detail. We separate data, interface and behavioural aspects of the specification in order to minimise changes in the problem domain. We illustrate the method using a simple case study, and discuss other object-oriented concepts such as components and design patterns.

1 Introduction

Formal specification is often seen as a complex and time consuming process. Although the benefits of a correct specification are seen later in the development life cycle [Lin94], if industry is to use formal methods, then it is important that the specifications produced are both reusable and maintainable. Object-oriented methods [Boo94, R⁺91] have received a lot of recent interest for these reasons, and it is reasonable to suppose that we may be able to borrow ideas from this paradigm to influence our formal specifications. In particular, in this paper we consider how to integrate the B method [Abr96] with object-oriented methods. We employ a use case analysis to derive the functionality of the system, and we then refine this functionality using a number of objects. In particular, we use the three object types (entity, interface, and control objects) of the OOSE [J⁺96] method. We should note at this point that it is not our intention to formalise OOSE, or indeed any object-oriented method, as is the case with many other types of method integration [Lan96b, Smi95, HS95]. Rather, we are attempting to help formalist build developments which are maintainable in the face of changing requirements and operating environments, and which can be reused with the minimum of effort. We do not suppose that all object-oriented concepts will be of use in a specification, since the B method is not object-oriented in nature. Instead, we will use only those aspects that we find useful when specifying. It is our intention to ease the process of specification, and attempting to use all object-oriented concepts may cause more problems than it solves, since imposing an “artificial” structure on a specification can blur the content of that specification.

There has been much interest in the integration of B and object-oriented methods [Lan96a, F⁺96]. These approaches show how object-oriented concepts can be modelled as AMN machines and machine structures, but our work here differs. In past work [Sho96] we showed how a class hierarchy can be “flattened”, with the objects being introduced during the refinement process. Here, we show how this flattening can be introduced as part of the object-oriented method.

In section 2 we introduce the idea of a use case, and consider how we may refine it into a number of communicating objects, and in section 3 we use a simple case study to show our ideas. In section 4 we discuss some of the considerations that we will have for our implementation, and in section 5 we draw our conclusions, and discuss future work.

Figure 1: An analysis showing two use cases and two roles

2 An AMN Specification

In this section we consider how we will decide what it is we are meant to be specifying, and how we will go about specifying that system in a maintainable and reusable way. The first part is achieved using a use case analysis, which will define a number of different ways in which the users of the system will interact with the system. The second part is carried out as a number of refinements of the use case. It is at this point that we introduce an object structure into our specification. As with most object-oriented methods, we relay heavily on diagrams. We then translate the diagrams into AMN machines and operations.

2.1 Specifying the Functionality

An AMN development traditionally takes place in a layered fashion. A specification is written, which is then implemented using a number of finer grain machines, and this continues until all of the machines can be expressed as library machines. The top layer expresses the functionality of system, and as such it is called the interface layer.

In [J⁺96], a use case is defined as

“a specific way of using the system by performing some part of the functionality.”

That is, it defines the behaviour of the system which is visible to the user of the system. This is very much like our definition of the interface layer of an AMN development, and it seems reasonable that we could formalize the use case analysis as a B machine. In fact, we will define a single machine with a number of operations, each operation representing one of the identified use cases. We can show the use case analysis diagrammatically as in figure 1, where we use the notation of [J⁺96]. The use case analysis reveals two use cases, and two *roles*. A role is a way of partitioning the environment, it is a class in the environment. An instance of a role is called an actor, and each actor can participate in multiple roles, and each role can be filled by a number of actors. Roles are important, since they help us define the interface to the system, as we will see in section 2. A role can be thought of as a class in the environment, with an actor being an instance of that class.

As we stated above, the use case analysis is formalized as a machine with operations representing the use cases.

```

MACHINE Example
VARIABLES x
INVARIANT y
INITIALISATION z
OPERATIONS
  UseCase1 ≐
  UseCase2 ≐
END

```

At this point we are as abstract as possible. Indeed, the interface layer of an AMN development is often defined as skip on an empty state space - the reason being that the most important thing for interface layer to show is termination, and such an operation is the simplest way of doing this.

Figure 2: The interface of the system identifies three interfaces, one of which involves an abstract role.

2.2 Defining the Objects of the Refinement

Once we have decided the functionality that the system must provide, we can start to consider the various ways in which we can do this. We start by defining a number of objects, each of which will be either an *interface*, *entity*, or *control* object, which we discuss these objects further in the following sections. These three types of objects are used in order to decouple the analysis - we do not want changes to the interface to cause major redesign of the system. A simple example of this is a printer driver - when we change printer, we only want to change the driver (the interface), not large parts of the program.

We may find that we will carry out a number of refinements of the use case and of the objects used in the use case. Every time we carry out a refinement we will use finer grain objects, and the object structure will change. Indeed, some of the objects we use at the top level of the refinement will not be present in the implementation of the system. This is because we use the objects as a convenient way of partitioning a certain level of abstraction in the specification, and as the level of abstraction changes, so the objects required to specify the behaviour will change. The changing structure of the specification is something which is central to the Catalysis method [DW96], where refinement is seen as an important way of partitioning a system. It is also an important part of the traditional approach to specifying in AMN, the layered approach, where the structure of a specification and its implementation are usually very different.

2.2.1 Interface Objects

Interface objects interact with the environment in order to receive and send data. By assigning each interface an object of its own we are able to decouple the actual interface from the object that provides the functionality. For example, we may have a system which requires an order code to be entered. We can define a object which is responsible for this, with an operation *GetOrder*, and any object requiring entry of an order simply uses this operation. It does not matter how the object actually retrieves the order code (it may be from a keyboard or a bar code reader). In fact, the retrieval method can be changed by replacing the interface object with another object which also provides the *GetOrder* operation. We shall use this as our definition of type - any two machines A and B which offer the same set of operations $\mathcal{O}$ can be considered to have the same type (and therefore be interchangeable) from the point of view of a third machine C which uses only those operations in $\mathcal{O}$. The importance of defining type in such a way is that any two objects which provide the same operations can replace each other - this is not always the case with the traditional definition of type in object oriented language [Sny86].

We show the interface as in figure 2.

When we have identified a number of roles in the environment, we may find that a number of our interface objects have common functionality. For example, two different roles may each require an interface which reports errors. We could introduce a single interface object, *ErrorReporter*, and an abstract role, *ErrorReceiver*, which both of the two existing roles inherit from. The new interface object may have a number of instances in the system, or it may be shared by the two roles.

We will represent each of the interface objects as AMN machines with single operations. For example, consider *Interface1* (which is an input operation) and *Interface3* (which is an output operation) from figure 2

MACHINE Interface1

Figure 3: Each entity object consists of a number of attributes.

OPERATIONS

```

  res ← Interface1Op ≐
    BEGIN res :∈ X END
END

```

MACHINE Interface3

OPERATIONS

```

  Interface3Op (param) ≐
    PRE param ∈ Y THEN skip END
END

```

There is little we can say about the objects at the abstract level. Input objects are defined as operations which return a nondeterministic value of the correct type. Output objects are defined as operations which may have an input (eg the message to be output), but we say nothing about their functionality.

2.2.2 Entity Objects

Entity objects store the data required by the system, and the behaviour associated with that data. Each entity object consists of a number of attributes, and a number of operations to manipulate those attributes, as shown by figure 3. We can also show the inter-relationships of entity objects in an ERD type diagram.

Entity objects are easily expressed in AMN, as discussed in [Lan96a, F⁺96], and as shown by the system library machines of the BToolkit, by using the machine as an object repository. A set, S is defined which represents all possible instances of the entity (called the object identity [DR93]), and a variable, $\mathcal{I}$ is used to identify current instances of the class. Attributes are then shown as mappings from $\mathcal{I}$ to the correct type, where the type of mapping reflects the participation and cardinality rules for the entity. In some circumstances, it may be possible to dispense with the object identity in favor of a key value, and we encourage this where possible. So, for figure 3, where *attribute1* is a key value (and therefore mandatory), *attribute2* is a mandatory 1 to 1, and *attribute3* and *attribute4* are optional 1 to many relationships, then we will have the following machine, which SEES another machine, Context, which we use to define the global sets. This allows a number of machines to access the same type definitions¹. The types T1,...,T4 are declared in Context, but they could be declared within the machine itself if they are not shared.

MACHINE Entity1

SEES Context

VARIABLES attribute1, attribute2, attribute3, attribute4

INVARIANT

```

  attribute1 ⊆ T1 ∧
  attribute2 ∈ attribute1 ↦ T2 ∧
  attribute3 ∈ attribute1 → T3 ∧
  attribute4 ∈ attribute1 → T4

```

OPERATIONS

¹Sharing of types can be done in AMN using either the SEES and the USES clauses. USES binds tighter than SEES, since it allows the variables of the used machine to be mentioned in the invariant of the using, with the affect that they can no longer be refined separately.

Figure 4: Control Objects are shown as arrowed circles.

```

:
END

```

2.2.3 Control Objects

Control objects contain the behaviour of the system which do not belong naturally to either the interface or entity objects. It is possible to define a system by simply forcing behaviour into the interface and entity objects, but this leads to a more tightly coupled design, with changes impacting on a large number of objects. Generally, control objects simply direct the interactions of interface and entity objects. Figure 4 shows how the control objects are shown diagrammatically.

Since the control objects are usually directing the interaction of the other object types, they are most easily represented in our machines as algorithmic refinements of the use case operations, ie *UseCase1* and *UseCase2* from section 2.1. We will show this in more detail in the case study of section 3.

2.3 Refining the Use Case

Once we have defined the objects which refine the use case, we can show this refinement formally. A static object diagram can be used to show how the objects interact (which we do in section 3). Formally, we define a refinement machine of our interface layer, and include the object machines we defined in the previous section.

```

REFINEMENT ExampleR
REFINES Example
INCLUDES Interface1, Interface2, Interface3, Entity1
INVARIANT R
OPERATIONS
  UseCase1 ≐
  UseCase2 ≐
END

```

3 A Simple Case Study

We will now use a simple case study to illustrate the ideas of the previous section. Our example is that of an invoicing system, where an invoice clerk enters orders to be invoiced in a batch fashion. All orders entered must be orders that already exist in the system (although we do not need to consider how those orders are entered), and when processed an order will be marked invoiced if there is enough stock to fill the order, otherwise it will be left as pending. We do not need to consider stock control, as that is done outside the system.

An extended version of this example is given in [Sho98].

3.1 Defining the Use Cases

The system must allow the invoice clerk to enter orders into the system, and to process those orders, and so we have two use cases, *QueueOrder* and *ProcessOrders* as shown by figure 5. We identify only one role, that of the invoice clerk, although it is possible that there are a number of actors filling that role. We discuss the use case analysis with

Figure 5: We identify two use cases and one role.

the client to ensure that this is indeed how the system should behave. We would also need to clarify any problems that the analysis raises. Such problems would be very high level, since we are using a very coarse grain analysis. For example, the client may talk about *valid* orders being queued, in which case we would have to clarify what makes an order valid.

We define this formally with the following machine

```

MACHINE Invoice
VARIABLES InvoiceQueue, Pending, Invoiced
INVARIANT
  InvoiceQueue  $\in \mathbb{P} \text{ ORDER} \wedge$ 
  Pending  $\in \text{ORDER} \leftrightarrow \text{STOCK} \times \mathbb{N} \wedge$ 
  Invoiced  $\in \mathbb{P} \text{ ORDER} \wedge$ 
   $\text{dom Pending} \cap \text{Invoices} = \{\}$ 
OPERATIONS
  QueueOrder  $\hat{=}$ 
    ANY order WHERE
      order  $\notin$  InvoiceQueue  $\wedge$ 
      order  $\in \text{dom Pending}$ 
    THEN
      InvoiceQueue := InvoiceQueue  $\cup$  {order}
    END;
  ProcessOrders  $\hat{=}$ 
    ANY filled WHERE
      filled  $\subseteq$  InvoiceQueue
    THEN
      InvoiceQueue :=  $\{\}$  || Pending := filled  $\triangleleft$  Pending ||
      Invoiced := filled
    END
END

```

All we say about an order that we wish to add to our set of queued orders is that it must currently be pending, and all we say about processing the queue is that some of those queued orders become invoiced, and the queue afterwards is empty. This very abstract level of specification allows many different valid implementations. The queue may be processed in a first come/first served fashion, or we may attempt to fill as many different orders as we can. Such questions should be discussed with the client. This is the value of the integration to the formalist - its helps define *what* it is that we should be specifying.

Since the stock and order input are outside the scope of our system, we shall assume that they are static. We actually intend to implement this shared data using files, but that is of no concern at this level of abstraction.

Figure 6: The interface objects for the example.

3.2 Defining the Objects

3.2.1 Interface Objects

We will require two interfaces, one to allow the invoice clerk to enter those orders to be invoiced, and one to output any errors to the clerk, as show by figure 6. In accordance with section 2.2.1 we introduce two machines, *OrderDevice* and *ErrorReporter*.

```
MACHINE OrderDevice
SEES InvoiceTypes
OPERATIONS
  val  $\leftarrow$  GetOrder  $\hat{=}$ 
    BEGIN val : $\in$  ORDER END
END
```

```
MACHINE ErrorReporter
SEES InvoiceTypes
OPERATIONS
  OutputReport(rep)  $\hat{=}$ 
    PRE rep  $\in$  REPORT THEN
      skip
    END
END
```

The machine *InvoiceTypes* defines a number of shared sets, and we declare it in the next section.

3.2.2 Entity Objects

We can define three entity objects in our system, *StockItem*, *OrderItem*, and *QueuedOrders*. Although we are not responsible for the maintenance of stock and orders, we need to model them since we will access them when attempting to invoice orders. Figure 7 shows the entities and their attributes.

In addition to three machines to represent the entity objects, we will also introduce a machine which declares a number of shared sets (types).

```
MACHINE InvoiceTypes
SETS
  ORDER; STOCK; INVOICESTATE = { invoiced, pending }
END

MACHINE OrderItem
SEES InvoiceTypes
```

Figure 7: The entity objects of the example.

```

VARIABLES
  KnownOrder, StockItem, Quantity, Status
INVARIANT
  KnownOrder  $\subseteq$  ORDER  $\wedge$ 
  StockItem  $\in$  KnownOrder  $\rightarrow$  STOCK  $\wedge$ 
  Quantity  $\in$  KnownOrder  $\rightarrow$   $\mathbb{N}$   $\wedge$ 
  Status  $\in$  KnownOrder  $\rightarrow$  INVOICESTATE
OPERATIONS
  MakeInvoiced (order)  $\hat{=}$ 
END

MACHINE StockItem
SEES InvoiceTypes
VARIABLES
  KnownStock, SQuantity
INVARIANT
  KnownStock  $\subseteq$  STOCK  $\wedge$ 
  SQuantity  $\in$  KnownStock  $\rightarrow$   $\mathbb{N}$ 
OPERATIONS
  DecreaseStock(amount, stock)  $\hat{=}$ 
END

MACHINE QueuedOrders
SEES InvoiceTypes
VARIABLES
  QOrder
INVARIANT
  QOrder  $\in$  iseq ORDER
OPERATIONS
  QOrder(ord)  $\hat{=}$ 
END

```

Figure 8: The control objects of the example.

Figure 9: The use case in terms of the objects that refine it.

3.2.3 Control Objects

We identify two control objects, as shown in figure 8. These two objects simply control the communications of the objects involved in the use cases, and so we shall describe this as refinements of the use case operations, as described in section 2.2.3. We do this in the next section.

3.3 The Use Case View

We can now view the use case in terms of the objects that refine it, which we do in figure 9. Formally, we define a refinement of our interface layer machine, and include the object machines. The invariant equates the abstract and refined variables.

```

REFINEMENT InvoiceR
REFINES Invoice
INCLUDES OrderDevice, ErrorReporter, InvoiceTypes,
  OrderItem, StockItem, QueuedOrder
INVARIANT
  InvoiceQueue = ran QOrder ∧
  dom Pending = Status ▷ {pending} ∧
  Invoiced = Status ▷ {invoiced}
OPERATIONS
  QueueOrder ≐
    VAR ord IN
      ord ← GetOrder;
      IF (ord ∈ ran QOrder) ∨ (ord ∉ KnownOrder) THEN
        OutputReport(ord)
      ELSE
        QueueOrder(ord)
      END
    END;

```

```

ProcessOrder  $\hat{=}$ 
END

```

The IF...THEN...ELSE statement forms the control object *Order Receiver*.

4 Implementation Issues

A B AMN specification will usually be used to generate an implementation, and ultimately used to generate C code. The process of implementation is different to the process of specification, and B places a number of restrictions on us that we will discuss in the following sections.

4.1 Implementing the Objects

Ultimately, any data that is stored by a B implementation must be stored using one or more of the system library machines. For complex data types, the Base technology of the BToolkit can be used to automatically generate these machines.

Input and output is usually achieved using the system library machine *basic_io*. This machine can only be included in a development once, however, and so it will be necessary to merge all of the interface machines into one machine, with a number of operations. This should not impact too heavily on our wish to keep the system decoupled, since the interface and data of the system are still separate.

An implementation in B imposes a number of restrictions upon the specifier. Firstly, the principle of full hiding comes into effect, that is a machine can no longer view the variables of another machine. Instead, machines must provide operations to allow access to those variables. This has the effect of decoupling the design. Secondly, AMN does not allow the nesting of loops, as the proof obligations become too complex. This means that a redesign must take place to move the loops into different machines. This will alter the structure of the AMN development, and should be reflected in the object design. We do not cover this here.

4.2 B Machines as Components

A B development progresses towards an implementation via a number of refinements. However, there are two ways in which we can carry out this refinement. Firstly, a specification can be refined a number of AMN refinement constructs, producing a chain, where each link is refined by the next, concluding with the implementation. This is shown in figure 10, where we use a circle to represent a specification, and the box to represent the implementation. Each refinement step will refine the data, algorithm, structure of the specification, or possibly a number of these [DT96]. The implementation will be composed almost entirely from the system library machines. This is the approach we have discussed here in this report.

The second method of refinement sees B machines as coarse grain objects, or components [J⁺96]. We do not use refinements, but instead go from specification directly to implementation, as shown in figure 11. This leads to a tree like structure developing. Each of the component machines may represent very complex constructs - a component may represent something as large as an HTML editor, or as small as a queue data structure. As such, each component will itself be composed from a number of other components, although ultimately, all machines will be built using the system libraries.

The component method of specification allows us a greater amount of reuse, but the leap from specification to implementation may sometimes be very large. Using a number of smaller refinement steps allows us to understand the system in more detail, but the steps are likely to be domain dependent, and therefore less reusable. Components can also be difficult to build as they need to be very flexible, since we do not know their intended use when they are produced.

Figure 10: A development can progress to implementation via a number of refinements.

Figure 11: A development can be implemented using a number of coarse grain machines.

5 Conclusions

We have introduced the B method, and suggested how an object-oriented approach to specification can decouple the analysis and design process, and lead to reusable and maintainable developments. We start with a use case approach to determine the functionality the system must offer, and we then refine it using three types of objects - interface objects, entity objects, and control objects. We show how these objects can be formalised using the B method, and how they can be used to refine the use case analysis. We also consider the various ways of refining an AMN specification, using the refinement construct of the toolkit, or using a component driven approach.

It is important to note that we are not attempting to use all aspects of objects in our development. Instead, we are attempting to ease the task of specifying complex systems by making them open to change, and by decoupling the machines from each other so that they are less problem specific, and therefore more reusable. However, it is important that this process does not impact negatively on the specification, by making it less “natural”, or by producing complex proof obligations. These are issues for future research, as is the usefulness of other object-oriented concepts. For example, the idea of design patterns [G⁺96] is likely to be of use. Design patterns are descriptions of solutions to standard problems. In AMN, for example, some standard problems are how to share data between machines, and how to structure nested loops.

References

- [Abr96] J.R. Abrial. *The B Book: Assigning Programs to Meaning*. Cambridge University Press, 1996.
- [Boo94] Grady Booch. *Object-Oriented Analysis and Design*. Benjamin/Cummings Publishing, 1994.
- [DR93] R. Duke and G. Rose. Modelling object identity. *Australian Computer Science Communications*, 1993.
- [DT96] J. Draper and H. Treharne. The refinement of embedded software with the b-method. In *Northern Formal Methods Workshop*, 1996.
- [DW96] Desmond D’Souza and Alan Wills. Extending fusion - practice rigor and refinement. In *OO Development at work: Fusion in the Real World*, 1996.
- [F⁺96] Philippe Facon et al. Mapping object diagrams into b specifications. In *Methods Integration Workshop*, 1996.
- [G⁺96] Erich Gamma et al. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1996.
- [HS95] B Henderson-Sellers. Relationship metamodeling in object technology. In *2nd Workshop on Rigorous Object-Oriented Methods (ROOM2)*, May 1995.
- [J⁺96] Ivar Jacobson et al. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley, 1996.
- [Lan96a] Kevin Lano. *The B Language and Method*. Springer Verlag, 1996.
- [Lan96b] Kevin Lano. *Formal Object-Oriented Development*. Springer Verlag, 1996.
- [Lin94] Richard Linger. Cleanroom process model. *IEEE Software*, pages 50–58, March 1994.
- [R⁺91] J Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
- [Sho96] Richard Shore. An object-oriented approach to b. In *1st B Conference*, 1996.
- [Sho98] Richard Shore. A use case approach to specifying the invoicing problem using the b method. In *Invoice Workshop '98*, 1998.

- [Smi95] Graeme Smith. A fully abstract semantics of classes for object-z. *Formal Aspects of Computing*, 7:289–313, 1995.
- [Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA*, 1986.