ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Antony Bryant and Lesley Semmens (Eds)

Methods Integration

Proceedings of the Methods Integration Workshop, Leeds, 25-26 March 1996

Paper:

Synthesis - An Integrated, Object-Oriented Method and Tool for Requirements Specification in Z

Mark Rawson and Pat Allen

Published in collaboration with the
British Computer Society

Synthesis - An Integrated, Object-Oriented Method and Tool for Requirements Specification in Z

Mark Rawson Pat Allen

School of Computing and Mathematics, University of Huddersfield,
Huddersfield, HD1 3DH, UK

Abstract

This paper discusses a method and supporting CASE tool for the production of formal specifications in Z [25] on the basis of an object-oriented analysis model. The analyst describes the required objects using diagrams and structured text and specifies formally the responses to events at the level of the individual object instances. The tool then generates the Z specification. The user of the tool need not in general be concerned with much of the structure and specific syntax of Z itself. The structure and components of the analysis models and the related Z are closely matched to aid understanding and support traceability. The tool enables the user to navigate between related components within and between both the analysis models and the generated Z specification. The paper describes the components of the models and the style of Z which is generated.

1 Introduction.

1.1 The growth of methods integration.

Methods integration is an approach to software development which involves the integration of normally distinct methods - in particular, formal and the more traditional structured methods. Following the pioneering work in the field by a number of authors, as reviewed in [22], the value of the approach is becoming widely recognised, as reported in the recent National Physical Laboratory survey on formal methods takeup [3], which revealed that 39% of existing users of formal methods used them together with structured (including object-oriented) methods. It is probably the case that most industrial users are taking a pragmatic approach, using a structured method to partition the system into modules, which are then specified formally. An example of this type of approach is the work carried out by Rolls Royce and Associates, who use Yourdon [29] together with VDM [11]. Authors such as France and Docker [8], Semmens and Allen [20, 21] and Plat [16] pioneered a more fully integrated approach, where a formal semantics is defined for the graphical and textual notations of the structured method. The work reported here builds in particular on the work of Semmens and Allen, adopting a similar approach to the formalization process.

Semmens and Allen's work in structured methods integration is developed and extended here in two ways. Firstly, given the increasing use of object-oriented languages and methods, it is a natural development to examine how object-oriented analysis and modelling techniques might be integrated with formal methods. Indeed, the arguably more advanced structuring mechanisms of the object-oriented approach may provide scope for producing better structured formal specifications. The inherent modularity of object-oriented specifications provides a structuring mechanism which enables modular verification and refinement. Moreover, object-oriented methods should benefit from a greater degree of formality, particularly in relation to their stated aim of increasing reuse of code and other development products, by improving precision of specification.

Secondly, it was considered unlikely that completely integrated formal-and-structured methods would be usable for large scale development without effective tool support. It would be far too time-consuming and error prone to perform the translations manually, or to check for consistency between the diagrams and the mathematical text. Existing CASE tools and formal methods support tools clearly could not provide

adequate support for fully integrated methods. In [2], Allen and Semmens describe a category of CASE tool, the 'specification assistant', which would provide the kind of support required.

This paper describes an integrated method for producing specifications in Z with an object-oriented structure and a 'specification assistant' designed to support the method.

1.2 Z and Object-oriented analysis.

Synthesis is a method which integrates the production of Z specifications into an object-oriented approach to requirements analysis, making use of well established models and techniques. The analysis approach is essentially a version of OMT [18], tailored to facilitate the production of a fully formal specification. Other authors have worked on the formalization of Shlaer and Mellor's Object Oriented Analysis method, in order to produce Z from an OOA specification [12, 15], and there has been some work done on the integration of Z with (parts of) the HOOD method [9, 17]. However, these approaches are not as fully integrated or as formally complete as the work described here, and there is as yet little or no tool support, as far as we are aware.

We have chosen standard Z as the basis for the Synthesis method, mainly because it is an established language with well developed techniques for its use and a range of tool support (type checkers etc.). Also, the structuring mechanisms provided by the Z language, such as schema inclusion and the schema calculus make it particularly suitable for building specifications of complex systems incrementally. A recent book [26] describes several approaches which have been proposed for adding object-oriented concepts to Z. Most of these have involved extending Z with object-oriented features, such as classes [1, 5, 7, 14, 19]. There is no reason why these extended languages should not be used in the context of an integrated method, but currently they are not standardised and lack tool support. Also, as Smith [24] points out, existing methods of refinement and verification already developed for Z cannot be used. Moreover, the extensions (such as the use of 'method invocations') tend to be design-oriented and are not necessarily appropriate for requirements specification. Several authors have also described ways of using Z in a more object-oriented style [10, 17, 24, 27] and certain ideas developed in their work are incorporated here.

1.3 Overview of the Synthesis method.

The Synthesis method involves the creation of a Class Model to describe the main structural components of the specification, namely the object classes and the relationships between them. Such relationships include super/sub-class hierarchies and aggregation ('is a part of') relationships. The Class Model also includes the structural composition of objects (their attributes). The Class Model is a model of the system state, structured as sets of object instances and relationships. System processes need to be defined in terms of their effect on the system state. The 'object oriented' approach to analysis seeks to associate processes with those components of the system state identified by the analyst as 'objects'.

The association of processes and objects is achieved by integrating the process descriptions within object life histories, represented as a state chart for each particular object instance or combined (orthogonal) state charts for aggregate objects (that is, objects with component parts). The transitions between states are triggered by events (significant externally or internally occurring states of affairs) and result in actions (processes representing individual objects' responses to the events). Actions may themselves 'generate' (record the occurrence of) events which then trigger further responses. The system is thus conceived of as a set of interacting objects each responsible for carrying out parts of the overall system functionality, where necessary triggering the actions of other objects via the generation of events.

Other models are also incorporated, to support the production of an accurate and consistent model of the system requirements. *Scenarios* represent interactive system behaviour as possible sequences of events and event responses. The interaction between objects is also shown in an *Object Interaction Model*, similar to the Object Communication Model used in the Shlaer/Mellor method. An *Object-Event Matrix* is incorporated to ensure completeness and consistency with reference to the object life histories and the system events, as recorded in the *Event Catalogue*.

1.4 Production cell case study.

The case study used to evaluate and illustrate the Synthesis approach is one which has been developed as a bench mark case study for the use of formal methods and notations. It was developed in Germany as part of the KorSo project and a number of specifications have been produced [13].

A production cell (illustrated at Figure 1) essentially involves the transfer of metal plates from a feeder belt to a press and thence to a deposit belt. This is modelled on a real-life process control application. However, the authors of the case study have introduced a crane which conveys the plate from the deposit belt back to the feedbelt, in order that the control process may be simulated as a continuing cycle. In a little more detail, the process is as follows.

Figure 1: The Production Cell.

Initially, the plate is fed onto a table which then rotates and elevates to the height required for removal. A robot with two extendible arms then removes the plate with one arm and rotates to place it on the press. This process is integrated with the removal of a pressed plate from the press by the other arm. The press has three positions - open for unloading, open for loading and closed (pressing). The robot can rotate clockwise and anti-clockwise. The arms can only extend and retract and have a magnetic pick up at one end. The crane can travel between the two belts and also has an extendible magnetic gripper similar in operating principle to the robot arms. Obviously, we shall focus in this paper on the behaviour of certain objects and on key components of the approach. A research report with full details of the whole specification process is in preparation..

2 Application of the Synthesis method.

A method for specifying requirements must incorporate two main components. Firstly, we need a language or notation with which to model and describe the requirements. Secondly, we must define a set of steps or heuristics for determining precisely what they are and how they are to be expressed. It is this latter component which is generally lacking or inadequate in formal “methods” and which provides the main justification for methods integration. One of the reasons (perhaps the main reason) why Z was chosen for this approach is its highly flexible structuring mechanisms based on the schema, schema inclusion and the schema calculus. These mechanisms enable components of the formal specification to be closely associated

with components of the analysis models at varying levels of abstraction, thus ensuring a consistent structure to the specification. How this is achieved will become clearer as the example specification is developed.

The Synthesis Method makes use of the following models.

- A Class Model
- A Type Catalogue
- An Event Catalogue
- An Object Life History (for each class)
- (Event/response) Scenarios
- An Action Specification (for each action)
- An Event-Object Matrix
- An Object Interaction Model

These components, and the Z which is associated with each, are described in the following sections. Most of the Z shown in this paper is in fact generated by the supporting CASE tool and is accessible by hypertext navigation within the tool from the model component with which it is associated. We shall say more about the nature of the tool support later in the paper.

3 The system state.

A Z specification normally requires a full description of the state of the system, before the operations of the system can be defined. Its production is then the first task which must normally be addressed. However, the object-oriented approach allows greater scope for examining component object behaviour separately and composing the system behaviour incrementally. The state of the system is modelled using a class model. The class model in object-oriented analysis is based on its predecessor, the Entity-Relationship Diagram, but incorporates or emphasizes certain additional features, namely typing, super/sub-classing and aggregation.

3.1 Classes and objects.

The major structural component in an object-oriented specification is the object class. The initial activity involves describing the system state in terms of its component parts (sets of objects) and the relationships between them. In a control system such as the one in question, real world objects suggest themselves as candidates. This may not always be the case, of course, and more guidance may be required.

Clearly then, the major objects here are the feeder and deposit belts, robot, crane, press, table, robot arms, crane gripper. Certain objects will have similar characteristics, thus presenting opportunities for subclassing and inheritance, as we shall see.

A diagrammatic representation of the class model for the production cell is shown as Figure 2. This will be easily comprehensible to an analyst versed in OMT or any similar approach. Note that sub-classes are positioned within the super-classes and that aggregation is shown as single lines without arrows.

3.2 Attributes and variables.

Objects in the real world have qualities, usually modelled as *attributes*. These descriptive qualities typically have constrained values, or *types*. A type represents a set of values that an attribute can have. The class variables and their types are specified in the tool using structured text editors. Type definitions are listed in the Type Catalogue (not shown here).

The available types mirror those of Z, that is given types, enumerated types, sets, mappings (functions) and so forth. For each class, we need to determine the variables (and their types) which will be used to describe the relevant qualities of (instances of) the class. For example, the current state of the crane may be *waiting at deposit belt*, *picking up from deposit belt* and so forth, and its motor is moving *forward* or

Figure 2: The Class Model.

backward or is *stopped*. The variables are therefore defined as being of enumerated types with the required given values.

<div style="border-bottom: 1px solid black; margin-bottom: 5px;"><i>Crane</i></div> <div> <i>craneid</i> : <i>CRANEID</i> <i>cranestate</i> : <i>CRANE_STATE</i> <i>crane_motor</i> : <i>CRANE_MOTOR</i> </div>

The enumerated type definitions translate directly into Z free types.

CRANE_MOTOR ::= *Forward* | *Backward* | *Stopped*

CRANE_STATE ::= *WaitingAtDepositBelt* | *PickingUpFromDepositBelt*
 | ...other states ...

In this case, the *CRANE_MOTOR* type is specified by the analyst, whilst the *CRANE_STATE* type is generated by the CASE tool, based on the states specified in the relevant object life history (as described later).

The system state comprises instances (objects) of the specified classes, each with a unique identifier. The system state is thus modelled as a mapping of identifiers to instances, one for each class.

<div style="border-bottom: 1px solid black; margin-bottom: 5px;"><i>Cranes</i></div> <div> <i>crane</i> : <i>CRANEID</i> $\mapsto$ <i>Crane</i> $\forall id : \text{dom } crane \bullet (crane(id)).craneid = id$ </div>

3.3 Relationships.

The most general type of relationship is one which exists between instances of classes. This is modelled in a standard way in the Class Model. A relationship has a name, cardinality and optionality.

<i>RobotLoadsPlateWithDeliveryArm</i>
<i>Robots</i>
<i>DeliveryArms</i>
<i>loadsplatewith</i> : <i>ROBOTID</i> $\leftrightarrow$ <i>ARMID</i>
<i>RobotLoadsPlateWithDeliveryArmIsOneToOne</i>
<i>RobotLoadsPlateWithDeliveryArmIsCompulsoryToCompulsory</i>

Schema inclusion is used here and in general to improve readability at each level of abstraction, as well as ensuring a close match between components of the analysis model and components of the formal specification. The expansions are omitted here, but follow an approach similar to that used in the Yourdon-Z method [20]. Additional predicates may also be specified to further constrain the relationship (e.g. to model cardinality more precisely).

3.4 Associative objects.

There are frequently good reasons for modelling certain relationships as objects in their own right, particularly if one requires additional variables to be attached. Such a relationship is modelled as an *associative object*. Suppose for example that the movement of a plate by a robot arm was to be such an object, called, say, a *Transfer*. The associated class schema would then be as follows.

<i>Transfer</i>
<i>transferid</i> : <i>MOVEMENTID</i>
<i>robotid</i> : <i>ROBOTID</i>
<i>armid</i> : <i>ARMID</i>
...others ...

The set of Transfer objects is then modelled as for other objects.

3.5 Super/subclasses and inheritance.

Two relationships are of especial importance in object-oriented analysis, namely the subclass and the aggregation relationships, both of which, however, have varying semantic interpretations in terms of their structural composition (component variables) and in terms of behavioural conformance and/or substitutability.

In Synthesis, subclasses “inherit” the variables of superclasses. This is often expressed in Z as a schema inclusion.

<i>FeedBelt</i>
<i>Belt</i>
...other variables ...
...additional predicate ...

Whilst straightforward and convenient, this approach is not entirely satisfactory and leads to difficulties in distinguishing variable names, when, for example, different subclass instances are incorporated into an aggregate object state or when separate subclass instance variables must be referred to in a precondition or action specification. In fact, the use of common variable names is always potentially problematic and it is safer to avoid it. Therefore, particularly where appropriate tool support is available, explicit inheritance and renaming of variables is preferred.

3.6 Aggregation.

Aggregation is a structuring mechanism which enables the analyst to model one object in terms of component objects, informally, the “is a part of” relationship. As far as component variables are concerned, the mechanism for implementing aggregation in the Synthesis method makes use of schema inclusion in a similar way to the subclassing mechanism.

Consider the table object. The elevator part of the table is:-

<i>Elevator</i>
<i>elevatorstate</i> : <i>ELEVATOR_STATE</i>
<i>elevator_motor</i> : <i>ELEVATOR_MOTOR</i>

The turntable part is:-

<i>Turntable</i>
<i>turntablestate</i> : <i>TURNTABLE_STATE</i>
<i>turntable_motor</i> : <i>ROTOR_MOTOR</i>

The table is now:-

<i>Table</i>
<i>tableid</i> : <i>TABLEID</i>
<i>Elevator</i>
<i>Turntable</i>

The major difference between aggregates and subclasses at this stage is that we do not consider that there are distinct sets of elevators and turntables as part of the system state. We will deal with the concepts of subclass and aggregation in terms of object behaviour in later sections.

3.7 Bringing together the system state.

So far, we have constructed a model of most of the required system state. The Z schema which brings together the components described thus far is as follows.

<i>ProductionCellState</i>
<i>Presses</i>
<i>Belts</i>
<i>FeedBelts</i>
...Other objects ...
<i>RobotLoadsPlate WithDeliveryArm</i>
<i>RobotUnloadsPlate WithDepositArm</i>
...Other relationships ...

We may add predicates over the state as a whole - that is, global state invariants. This schema will be slightly modified later, as we consider the behavioural aspects of the system, in particular by the addition of a component to model the storage of generated events.

4 Modelling system behaviour.

As indicated earlier, our description of system behaviour is based on an event/response model. Events represent occurrences which signal the need for some action. For example, there may be changes in the real world external to the system being modelled or an input of data from a sensor or user. The system response will generally involve a change of state of the system. That is, one or more of the components (objects or relationships) which constitute the system state will be modified. Moreover, part of the system response may

be to generate events which in turn will stimulate further response. Thus the overall system response to an external stimulus may be complex and involve separately defined responses, ordered in time. We will need, therefore, a mechanism for describing the occurrence of system generated ('internal') events and for describing the system response to such events.

4.1 System behaviour in Z.

A Z model of system behaviour relies on descriptions of state changes (plus input and output) with implicit (that is, undistinguished) pre- and postconditions. A Z operation is defined in terms of its effect on the system state. For any particular configuration of the system state, if the (implicit) precondition of an operation is true, then the poststate of the operation is a possible next state for the system. If no other operation precondition is met, then the poststate is the *only* possible subsequent state. The possible state changes resulting from the sequential application of the defined operations thus represent the possible behaviour of the system. We can see, then, that the "enabling" of a system operation depends on three factors.

- The state of the system.
- The presence and value of inputs (possibly in relation to the state).
- The occurrence of events and the value of any associated data.

These factors will be explicitly identified and defined in our model of system behaviour and then combined within the structure of the model and associated Z specification to constrain that behaviour.

The system response to individual events in terms of state changes may be complex. We build up our description of system behaviour by defining partial operations on object instances, combining them into full object responses using the schema calculus and then using operation promotion to define system level response. This use of the schema calculus and operation promotion follows a common approach to the construction of Z specifications [28].

4.2 System behaviour in Synthesis.

The Synthesis method requires the construction of four separate but related models, each of which provides a view on the system behaviour, together with an Event Catalogue.

- Object Life Histories, incorporating partial system responses as they affect objects in different states.
- One or more Scenarios, representing sequences of events and responses in terms of object state changes and event generation.
- An Object Interaction Model, showing at a high level of abstraction how objects interact via event generation and response.
- An Event-Object Matrix, showing in tabular form the overall effect of event responses in terms of object creation, reading, updating and destruction, to allow cross-checking.

5 Events.

Events are classified as internal or external. External events represent occurrences or situations external to (in the environment of) the system being modelled. Internal events are occurrences affecting objects which are part of the *modelled* system state. They will signal the fact that a particular object in the system is in a particular state and that some further response is required from another object. For example, when the robot has finished loading the press, the press may then proceed to press the loaded plate.

Explicit events are not strictly necessary to describe the fact that some internal state of affairs has arisen. A precondition to an operation would suffice. However, explicitly identifying events as triggers of system responses enables us to think and reason more effectively about system behaviour over time, that is, as sequences of distinct events and responses. This is similar to the approach taken in process-oriented

formalisms, such as CSP, CCS, LOTOS and so forth. We shall see in later sections that the event trace, or scenario, forms an important part of the Synthesis method too. The scenario relies on our being able to describe system behaviour in terms of sequences of events and system responses to events.

5.1 Identifying events.

We now need to consider how we determine what the relevant events are, both external and internal. We may draw on experience and guidance in using existing analysis methods in this context. Firstly, we must identify all the external circumstances which will require system response and produce an event list and initial Event Catalogue. We may also note at this stage any major internal circumstances which are of particular significance, such as the table reaching the correct height and angle for the plate to be removed by the robot arm or the plate being deposited on the deposit belt. In the case of internal events, we can see that the actions of one part of the system have produced a situation which requires some response from some other part of it. This initial event list will be supplemented in due course by events which are required to construct a complete and coherent model of system behaviour.

5.2 Event catalogue.

The Event Catalogue describes all events in the system. The information associated with an event is as follows.

- Name
- A textual description
- Source: generated by which object action(s) or external entity
- Data: any data associated with the event
- Object response(s): the object actions which have been defined as responding to the event

It is worth noting at this point that the sources and responses to events are derivable from the object life histories and the object interaction model, detailed later, and, indeed, this is automatically carried out by the supporting CASE tool.

In terms of the case study, the external events represent a variety of sensor inputs, summarized as follows (bearing in mind that there may be more than one production cell).

1. A press reaches its loading/unloading/pressing position.
2. Sensors signal the extension of arms/grippers.
3. Sensors signal the angle of rotation of robots/tables.
4. A table reaches its loading/unloading position.
5. A crane is detected over a deposit/feed belt.
6. A plate is detected at the end of a feed/deposit belt.

Associated with each type of event is a Z schema containing appropriate variables.

<i>RobotHasRotatedEvent</i> <i>robot</i> : <i>ROBOTID</i> <i>angle</i> : $\mathbb{N}$

Now, part of the response of a press to receiving the relevant signal from the sensor might be to generate an event to signal that it is ready for loading.

$\text{PressIsReadyForLoadingEvent}$ $\text{press} : \text{PRESSID}$ $\text{robot} : \text{ROBOTID}$
--

External events will be treated as inputs to the system, as we shall see later. The occurrence of events and the system's response to them are deemed to be instantaneous. There is, therefore, no problem of events being ignored or missed. However, a system response may involve the generation of more than one event. The order of occurrence or response is not explicitly constrained. So we need a variable to record the internal events which have occurred at any particular point in time. Our model of the system state can now be completed by incorporating a component containing the set of outstanding internal events. Since Z is strictly typed, a type constructor is used to 'cast' the different event types, in the following form.

$$\text{EVENT} ::= \text{event}_1 \langle\langle \text{Event}_1 \rangle\rangle \mid \dots \mid \text{event}_n \langle\langle \text{Event}_n \rangle\rangle$$

InternalEvents $\text{internal_events} : \mathbb{P} \text{EVENT}$

This schema is then included in the system state schema shown earlier.

6 Constructing the behavioural specification.

We deal firstly with those components of the analysis model from which the formal specification is derived and show how the specification in Z is produced. Then we will address the issues of clarity, completeness and consistency, introducing the complementary system models, namely Scenarios, the Event-object Matrix and the Object Interaction Model.

6.1 Object behaviour and object lifecycles.

The dynamic behaviour of objects is modelled using a version of state charts (enhanced state transition diagrams), here called Object Life Histories (OLHs). The basic object life history comprises a set of discrete states with transitions between them. Attached to transitions are the events which trigger them, any additional precondition and the responding action. The actions collectively describe the objects' responses to the events which cause the state transitions.

In terms of our case study, we may decide that certain objects' behaviour can be usefully defined in isolation. This 'bottom up' approach will help in identifying generally useful objects which will form the basis of our specification (and possibly others). Figure 3 shows the relatively straightforward life history of a belt.

In considering the behaviour of individual objects, we need basically to consider the distinct states such objects may be in, what circumstances lead to changes of state and what actions are associated with these changes.

6.2 States.

The belt in the production cell is a finite state object. A belt is either running or stopped. It might however have been the case that the belt had a variable speed. In such a case, states may represent abstractions over a number of states. We might, for example (in a different system), have a 'bank account' object and define an 'overdrawn' state representing all those states of the object where the balance is less than zero. For each state in the object's history, then, we need to specify what that state *means* by attaching a suitable predicate defined over the attributes of the object in question. We include in each object's state a variable representing the possible discrete states of the object.

Belt $\text{beltid} : \text{BELTID}$ $\text{belt_state} : \text{BELT_STATE}$

Figure 3: The Belt Object Life History.

The actual, physical objects to be controlled in the case study include a belt - actually a belt motor, which should be switched on and off. It is of course the control of this particular object that we are specifying. Actuator controls could be modelled as outputs, which would tend to imply one-off control signals (however often produced). Alternatively (and more abstractly), we may include state variables representing the state of the actuator output. This is then considered to be a continuous control signal (although it may not be implemented as such). We introduce, therefore, a variable representing the actuator

<i>Belt</i> <i>beltid</i> : <i>BELTID</i> <i>belt_state</i> : <i>BELT_STATE</i> <i>belt_motor</i> : <i>BELT_MOTOR</i>
--

with a suitable type

BELT_MOTOR ::= *On* | *Off*

The state specification is then:-

<i>BeltRunning</i>	_____
<i>Belt</i>	_____
<i>belt_state = Running</i>	
<i>belt_motor = On</i>	

6.3 Event triggers.

A state transition represents the basic object response to the occurrence of an event. Attached to each transition, therefore, is a reference to the event which triggers it. The ‘Plate Detected’ event in Figure 3 refers to the sensor input detailed earlier. The ‘Safe To Run Belt’ event will be generated by objects associated with belts to signal that the relevant belt may continue running.

The modelling in Z of the event occurrence depends on whether the event is external or internal. The (external) ‘Plate Detected’ event, for example, is straightforward..

<i>RespondingToPlateDetectedEvent</i>	_____
<i>plate_detected_event? : PlateDetectedEvent</i>	

Let us now look at the occurrence of the internal ‘Safe To Run Belt’ event.

<i>RespondingToSafeToRunBeltEvent</i>	_____
<i>InternalEvents</i>	
<i>safe_to_run_belt_event : SafeToRunBeltEvent</i>	
<i>safetorunbelt(safe_to_run_belt_event) ∈ internal_events</i>	

It is again worth emphasizing at this point that this part of the specification is generated by the tool. Note also that these schemas make the event data available to the specifier via the included event instance. This is also tool supported by listing the variables in scope where this is required to be known, for example, when specifying preconditions and actions.

6.4 Preconditions.

The occurrence of an event is of course effectively a precondition for the transition to occur. We may specify further preconditions. A precondition should have a unique name and one or more predicates. Thus the same precondition may occur elsewhere in the specification. The predicate of a precondition may be defined with reference to any part of the system state. This is acceptable, encapsulation notwithstanding, because no part of the state will be modified by the precondition (it does not refer to any ‘after’ state). We should, however, explicitly identify which objects and relationships will be in scope for the precondition. This will enable us to include the relevant variables in the Z translation and identify object/relationship reads in the design of the software. On this basis, a support tool will generate the required schemas and identify and display all the variables in scope, thus allowing the analyst to concentrate on the specification of the predicates.

The mechanism for the specification of preconditions is essentially similar to that for actions, described in Section 7.

7 Actions.

In Synthesis, the analyst is required to specify a number of actions, each attached to one or more transitions on an object life history and thus associated with the event(s) which trigger it. Most of the framework of the formal specification is standardised and therefore tool generated. The analyst may therefore concentrate on the responses to events at the level of the individual object instance. This is an important point. Whilst the specification overall may be complex, particularly its formal translation, the behaviour of the system is conceived of as consisting of the combined behaviour of a number of separate components, each of which

can be considered in isolation, with the aggregate system functionality being (automatically) composed in accordance with well defined rules. Each component should in itself be both easily comprehensible and implementable. This is precisely the purpose and benefit of a structured (including object-oriented) approach.

The analyst is required to identify and specify:-

- The name of the action
- The system level effect (create, read, update or destroy) on the object involved
- Inputs, outputs and any local variables
- Information requirements ('reads') from objects and relationships
- The system level effects on relationships to which the current object is party
- Internally occurring events
- Formal predicates defining the effect of the action

```
PickUpFromTable Updates A Robot
Responds to event TableAtPickUpPosition
Reads no objects
Affects no relationships involving owner object
Reads relationship RobotLoadsPlateWithDeliveryArm
Has no inputs
Has no outputs
Uses no local variables
Has event data
ready\_to\_extend\_arm\_event.arm : ARMID
ready\_to\_extend\_arm\_event.extension : \nat
Has read only variables
robotid : ROBOTID
robotstate : ROBOT\_STATE
deliveryarmid : DELIVERYARMID
Updates owner class variables
rotor, rotor' : ROTOR\_MOTOR
Generates events ReadyToExtendArm
Has predicates
ready\_to\_extend\_arm\_event.arm = deliveryarmid
ready\_to\_extend\_arm\_event.extension = ROBOT\_TO\_TABLE
rotor' = Off
End of action specification.
```

Figure 4: The Action Specification.

Figure 4 shows the structure of an action specification. It is created in a structured text (syntax directed) editor in a standard format. The variables in scope are automatically listed by the tool. As is generally the case, the editing frame also allows hypertext navigations to component definitions (such as type definitions or event specifications).

We shall now consider the components of the Z specification associated with each part of the action specification.

7.1 The system level effect.

The first line of the action specification determines the system level effect, that is, which of four standard schemas will be used in the promotion of the operation specification to the system level. Recall that operation promotion is a common Z technique for transforming the effect of an operation defined over an entity or object *instance* to one defined over the set of such objects making up part of the system state [28].

The update schema, for example, sets the value of the ‘before’ state of the robot instance to the robot identified by the object identifier and updates the after state of the robots to include the new (updated) value. The two robot states will be related by the action predicate.

<i>UpdateARobot</i>	_____
$\Delta Robot$	
$\Delta Robots$	
$\theta Robot = robot(robotid)$	
$robot' = robot \oplus \{robotid \mapsto \theta Robot'\}$	

Effects of actions on relationships are dealt with in a similar manner.

7.2 Event generation.

Each action attached to a transition may result in the generation of one or more internal events, which need to be added to the outstanding internal events, along with those generated by other objects or other component parts of the current object. To formalize this model in Z, we start with the events generated by a particular (partial) action, building up to the object and system response to the event step by step. This process will be detailed later in considering the composition of the system event response. For now, we require a variable to record the events generated by any particular object.

<i>RobotInternalEvents</i>	_____
$robot_internal_events : \mathbb{P} \text{ EVENT}$	

The value that this variable has will depend on which particular partial response is triggered. For the action we are considering, the associated schema is the following.

<i>RobotPickUpFromTableEvents</i>	_____
<i>RobotInternalEvents</i>	
$ready_to_extend_arm_event : ReadyToExtendArmEvent$	
$robot_internal_events =$	
$\{readytoextendarm(ready_to_extend_arm_event)\}$	

7.3 Action predicate.

The analyst is required to specify the action effect in Z, using an appropriate concrete syntax. This requires knowledge of the specific syntax as well as an understanding of the model components and their semantics. However, as can be seen in Figure 4, the predicate may be fairly straightforward, involving equations over variables which are automatically included in the relevant structured text editor.

7.4 Complete action description.

The schema for the Action Specification for the partial action can now be constructed from the component schemas.

<i>RobotPickUpFromTable</i> <i>RobotIdIsUnchanged</i> <i>UpdateARobot</i> <i>ReadsRobotLoadsPlateWithDeliveryArm</i> <i>RobotPickUpFromTableEvents</i>
<i>ready_to_extend_arm_event.arm</i> = <i>deliveryarmid</i> <i>ready_to_extend_arm_event.extension</i> = <i>ROBOT_TO_TABLE</i> <i>rotor'</i> = <i>Off</i>

If a different object event response is defined for different states, the composed event response is the disjunction of the partial responses. This of course accords with a common method of defining partial operations in Z and constructing total operations by disjunction. This will be detailed later, after we have considered the implications of aggregation and subclassing.

8 Aggregation and orthogonal state machines.

Where an object's state has been composed as an aggregate of separate states in the Class Model, its behaviour is composed of separate life histories for each component state. That is, we use orthogonal state machines to model its composite behaviour, as in OMT. There are very distinct advantages in doing this, the main one of which is illustrated in considering the behaviour of the table (Figure 5). We have two orthogonal machines with four states each, but would potentially require sixteen in a single machine. By focussing on each aspect of its behaviour separately, we may significantly reduce the number of states and simplify our model.

The process of constructing each machine is identical to that of constructing single machines. However, the predicates used in defining the states may only refer to those variables contained in the relevant substate. In the Z translation, this is ensured by including only the substate schema.

<i>ElevatorRaising</i> <i>Elevator</i>
...state predicate ...

We consider how the overall object response is composed in Section 10.

9 Subclasses and derived behaviour.

We may observe that the behaviour of the arm and the gripper are essentially similar. Both involve extending and retracting and picking up a plate by activating an electro-magnet. This leads us to identify a common superclass. In Synthesis, subclasses are required to provide substitutable (conformant) behaviour (see Hall [10] for a discussion of behavioural subtyping), according to formally specified rules which may be informally summarized as follows.

1. The life history of the superclass is duplicated in the subclass, except as provided by the remaining rules.
2. Orthogonal life histories may be defined over additional class attributes.
3. Creating events (transitions from the Dead state) are removed and may be replaced by different events.
4. Super and substates may be added to existing states.

Figure 5: The Table Object Life History.

5. Pre-conditions may be weakened, including moving transition starts to superstates of the original start state.
6. Post-conditions (action predicates) may be strengthened, including moving transitions ends to substates of the original end state.
7. Responses to entirely new events may be added to any component life history.
8. If a response to an existing event is to be added, the event may only be added to an orthogonal machine on which it already appears (except from a state which already has a transition with that event attached).

The latter constraint is required to ensure that an event response precondition is not strengthened by conjoining it with a further precondition or that the postcondition is not weakened by adding a disjoint postcondition.

10 Composing the event response.

Having defined a number of individual actions attached to the state transitions of objects, we now have to consider how these actions are combined to produce the overall system response to events. Note again that the Z constructs involved are produced automatically by the tool.

10.1 Composing object responses.

An object's response to an event may need to be composed in two ways. Firstly, the same event may appear on a number of transitions, thus implying that the object's response will differ in different circumstances.

Secondly, if the object is composed of sub-components (that is, it is an ‘aggregate’), each component may have a distinct response, which will contribute to the overall response. Clearly, the first case involves a disjunction of the partial actions and the second a conjunction.

Consider the behaviour of the table, composed of a turntable and an elevator (Figure 5). We may observe that an event signalling the rotation of the turntable may be responded to by the generation of an event to start the belt running, if both component parts are in the correct position, that is, if the table is already raised to the correct height and the event data indicates that the table is in position against the feed belt.

Recalling how the internal (generated) events for objects are recorded (Section 7.2), we may build up the table object’s response to the **TableHasRotated** event as follows. Note in particular that the events generated by an action are included in the action specification (Section 7.4). The components of the disjunctions in the following Z expressions are the precondition, pre- and post-states of the object and the associated action.

$$\begin{aligned}
 & \textit{TurntableResponseToTableHasRotatedEvent} \triangleq \\
 & \quad (\textit{RotatedToArmAngle} \wedge \\
 & \quad \textit{TurntableRotatingClockwise} \wedge \\
 & \quad \textit{TurntableAtArmAngle}' \wedge \\
 & \quad \textit{TurntableStop}) \\
 \vee \\
 & \quad (\textit{RotatedToFeedBeltAngle} \wedge \\
 & \quad \textit{TurntableRotatingAntiClockwise} \wedge \\
 & \quad \textit{TurntableAtFeedBeltAngle}' \wedge \\
 & \quad \textit{TurntableStop})
 \end{aligned}$$

$$\begin{aligned}
 & \textit{ElevatorResponseToTableHasRotatedEvent} \triangleq \\
 & \quad (\textit{RotatedToArmAngle} \wedge \\
 & \quad \textit{ElevatorAtArmHeight} \wedge \\
 & \quad \textit{ElevatorAtArmHeight}' \wedge \\
 & \quad \textit{ElevatorNotifyAtPickUpPosition}) \\
 \vee \\
 & \quad (\textit{RotatedToFeedBeltAngle} \wedge \\
 & \quad \textit{ElevatorAtFeedBeltHeight} \wedge \\
 & \quad \textit{ElevatorAtFeedBeltHeight}' \wedge \\
 & \quad \textit{ElevatorNotifyBeltSafeToRun})
 \end{aligned}$$

Now we can specify the overall object response to the event **TableHasRotated**.

$$\begin{aligned}
 & \textit{TableResponseToTableHasRotatedEvent} \triangleq \\
 & \quad \textit{TableRespondingToTableHasRotatedEvent} \wedge \\
 & \quad \textit{TurntableResponseToTableHasRotatedEvent} \wedge \\
 & \quad \textit{ElevatorResponseToTableHasRotatedEvent}
 \end{aligned}$$

The generated events are the union of the events generated by the component parts.

$ \begin{aligned} & \textit{TableResponseToTableHasRotatedEvents} \\ & \quad \textit{TableInternalEvents} \\ & \quad \textit{TurntableInternalEvents} \\ & \quad \textit{ElevatorInternalEvents} \end{aligned} $
$ \begin{aligned} & \textit{table_internal_events} = \\ & \quad \textit{turntable_internal_events} \cup \textit{elevator_internal_events} \end{aligned} $

10.2 Composing the system response.

We need first of all to construct the set of events generated by the separate objects.

$$\begin{array}{l}
 \text{SystemResponseToTableHasRotatedEvents} \\
 \Delta \text{InternalEvents} \\
 \text{TableResponseToTableHasRotatedEvents} \\
 \dots \text{other objects' generated events} \dots \\
 \hline
 \text{internal_events}' = \\
 \quad (\text{internal_events} \setminus \\
 \quad \quad \{ \text{tablehasrotated}(\text{table_has_rotated_event}) \}) \cup \\
 \quad \quad \text{table_internal_events} \cup \\
 \quad \quad \dots \text{other objects' events} \dots
 \end{array}$$

The overall system response to the event can now be defined. We use existential quantification rather than explicit variable hiding to hide the object variables in promoting the operation specifications.

$$\begin{array}{l}
 \text{SystemResponseToTableHasRotatedEvent} \hat{=} \\
 \quad \exists \Delta \text{Table}, \\
 \quad \dots \text{other objects} \dots \bullet \\
 \quad \quad \text{SystemResponseToTableHasRotatedEvents} \\
 \quad \quad \text{TableResponseToTableHasRotatedEvent} \wedge \\
 \quad \quad \dots \text{other objects' responses} \dots \\
 \quad \quad \exists \text{Robots} \wedge \\
 \quad \quad \exists \text{Cranes} \wedge \\
 \quad \quad \dots \text{other unchanged objects} \dots \\
 \quad \quad \exists \text{RobotLoadsPlateWithDeliveryArm} \\
 \quad \quad \dots \text{other unchanged relationships} \dots
 \end{array}$$

This completes the formal translation of the specification into Z.

11 Additional models.

We now briefly outline the additional models incorporated into the method to assist in ensuring clarity, completeness and consistency. No further Z is generated with reference to the following models.

11.1 Scenarios.

A scenario is a description of the interactive behaviour of the system over time. It details in sequence the occurrence of events both external and internal and how objects respond, given their current states. A correct, or possible, scenario is one which describes behaviour consistent with the object lifecycle models. We may therefore check mechanically that the required behaviour is possible (assuming we have a correct scenario in mind) or that an undesirable scenario is not possible.

Creating a scenario is a very useful tool in the early stages of specification, where the overall system behaviour is difficult to understand, and may be sketched simply as an event sequence initially, with details of object states added as their behaviour is determined.

A scenario takes the following form. It is essentially a concrete representation of a sequence of simple or compound object transitions, each with a triggering event, optional pre-condition, state change(s) and subsequent generated events. We may note on each transition the possible objects responding to internal events, for the sake of clarity.

Feedbelt is Running

Table is AtFeedBeltAngle and AtFeedbeltHeight

Sensor signals PlateDetected to FeedBelt
 FeedBelt signals FeedBeltStopped to Table and is Stopped
 Table is RotatingClockWise and Raising
 Sensor signals TableHasRotated to Table
 If RotatedToRobotAngle, Table is AtRobotAngle and Raising
 Sensor signals UpperPositionReached to Table
 Table signals AtPickUpPosition to Robot and is AtRobotAngle and AtRobotHeight

11.2 The Event-Object Matrix.

As the model may be complex and the effects of event responses broken up and distributed between different objects, it is helpful to be able to tabulate the system level effects on objects to check for completeness and consistency. Recall that an event can result in the creation, reading, updating or destruction of an object. We use an Event-Object Matrix, similar to the Entity-Event matrix in Information Engineering [6], to record these effects. Full CASE tool support will generate the matrix automatically on the basis of the other models, since object effects are already explicitly identified. Inconsistency and incompleteness can also be automatically highlighted.

11.3 The Object Interaction Model.

The Object Interaction Model is a visual representation of the source and destination of events. It shows the objects in the system and the event interaction between them and is essentially the same as the Object Communication Model of Shlaer and Mellor [23]. This model is used to trace the thread of control through the system as it responds to some particular event. It permits us to do this visually by checking the destination(s) of generated events and thereby accessing the behavioural specification of the destination object(s). Event sources and responses are also recorded in the Event Catalogue, which permits hypertext navigations to and from the relevant action specifications.

An object interaction model may represent insight into the system that is not incorporated into the other models at some point in time. Therefore, although the model may be generated automatically on the basis of other models, it is considered preferable to develop it independently.

12 The CASE tool.

A prototype specification assistant has been developed to support the integrated method, some of the functionality of which has been referred to earlier in the paper. Essentially, it provides editors for diagrams and structured text (including the formal text), and automatically translates elements of the structured specification into the formal notation. It ensures consistency between the diagrams and other documents of the structured specification and the formal specification.

In general, an element of the Z specification, usually a schema, is associated with each separately identifiable component of each of the Synthesis model diagrams. The tool allows the associated Z to be viewed via an operation defined on the diagram element (as one item on the menu of available operations). The Z schema associated with any collection of components, such as the object (state) model as a whole, will generally comprise a set of inclusions of schemas associated with the subcomponents, as detailed earlier.

Thus the structure of the formal specification maps neatly onto the structure of the analysis models. The analyst can navigate along hypertext links throughout both the Synthesis models and the generated Z specification. For example, schema references (such as inclusions) are selectively expandable by displaying the actual schema referred to. The analyst should be able to concentrate on the details pertinent to any particular diagram or text object, at an appropriate level of abstraction for the task at hand.

The Z specification is currently produced in a format suitable for automatic type checking by the *fuzz* tool and for printing using $\text{\LaTeX}$. Alternative concrete representations could be generated for use with other tools, such as proof assistants.

13 Conclusions.

The formal specification produced using this method is large and highly standardized. It may, therefore, lack the elegance and ‘tailored’ quality of other formal specifications. However, the method is intended to be used in the specification of inherently complex systems and the specification is, therefore, highly structured and clear at each level of abstraction. The analysis *method* is essential to the production of correct system specifications. The Synthesis method and the consequent style and structure of the specification produced closely match a commonly used approach to the production of Z specifications, making extensive use of its structuring mechanisms - in particular, schema inclusion, the schema calculus and operation promotion.

Equally importantly, the method results in structurally distinct and clear object specifications, comprising an abstract specification of the state of individual objects and operations defined on that state. This is the basis for an object-oriented design and subsequent development. All of the important components are separately identifiable and therefore independently implementable. Integrating these separate components should produce a well structured system, provided that the initial analysis is sound. Important design decisions remain unconstrained, for example which specified operations should be implemented as separate methods, the extent of process concurrency, and so forth.

The project has demonstrated the feasibility of implementing substantial tool support for formal software development, at least in the early stages of the software development process. This enables much of the complexity of the formal specification to be hidden from the analyst, who is thus enabled to focus specifically on the required functionality in terms of individual object behaviour. There are two forms of representation of the specification, each usable independently, but with clear structural relationships between the two. A significant advantage of this approach is that this structure is also built into the tool and allows the specifier to navigate through the specification within the software environment, rather than being required to manipulate a purely textual representation.

14 Future work.

The extent to which this approach can be extended through the development process forms the basis of ongoing research. In particular, CASE tool support for traceability is an important area. We should ideally be able to trace components through the development process from requirements capture to implementation and maintenance.

The generation of proof obligations, incorporation of refinement rules and subsequent integration with proof assistants (e.g. ProofPower-Z) remains to be explored.

The data repository of the prototype tool only incorporates sufficient detail of the abstract Z syntax to navigate the formal specification. A full(er) model of Z would facilitate further support, such as more complete, integrated, syntax directed editing facilities.

The eventual aim of the project is to provide an integrated, object-oriented method for the entire formal software development process, including software design/refinement, with CASE tool support specifically designed to support the integrated approach.

15 Acknowledgements

The authors gratefully acknowledge the work of Ben Ryan and Diane Kitchin in the implementation of the tool and the suggestion by Lesley Semmens that a CRUD matrix may be used for consistency checking.

References

- [1] A. J. Alencar and J. A. Goguen. OOZE: An Object-Oriented Z Environment. In P. America (ed), Proceedings of ECOOP’91 European Conference on Object-Oriented Programming Geneva, Switzerland, July 1991, Lecture Notes in Computer Science, vol 512, pp. 180-199, Springer-Verlag 1991.
- [2] Pat Allen and Lesley Semmens. Integration of Structured Methods and Formal Notations. ISD’94, Bled, Slovenia, September 1994.

- [3] S. Austin and G. I. Parkin. *Formal Methods: A Survey*, National Physical Laboratory, Queens Road, Teddington, Middlesex, TW11 0LW, UK, March 1993.
- [4] Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Yourdon Press Computing Series. Prentice-Hall, 1991.
- [5] E. Cusack and G. H. B. Rafsanjani. ZEST. In S. Stepney, R. Barden and D. Cooper (eds). *Object Orientation in Z*. Workshops in Computing, Springer-Verlag, 1992.
- [6] A. Davids. *Practical Information Engineering*, Pitman, 1992
- [7] R. Duke and P. King and G. A. Rose and G. Smith. The Object-Z Specification Language. Technical Report SVRC 91-1, Version 1. Software Verification Research Center, University of Queensland, Queensland 4072, Australia, May, 1991.
- [8] R. B. France and T. W. G. Docker. Flexibility and Rigour in Structured Analysis. In *Information Processing 89*, Elsevier Science Publishers B.V., 1989.
- [9] R. Di Giovanni and P. L. Iachini. Hood and Z for the development of complex software systems. In D. Bjørner and C. A. R. Hoare and H. Langmaack (eds), *VDM and Z – Formal Methods in Software Development*, Kiel, Germany, 17–21 April 1990, Lecture Notes in Computer Science vol 428, Springer-Verlag, 1990.
- [10] J. A. Hall. Specifying and Interpreting Class Hierarchies in Z. In J. P. Bowen and J. A. Hall (eds.), *Z User Workshop*, Cambridge 1994, Springer-Verlag 1994.
- [11] V. Hamilton. Experiences of combining Yourdon and VDM. Methods Integration Workshop, Leeds Metropolitan University, September 1991.
- [12] Jonathan Hammond. Producing Z specifications from object-oriented analysis. In J. P. Bowen and J. A. Hall, editors, *Z User Workshop, Cambridge, 1994*, pages 316–336. Springer Verlag, 1994.
- [13] C. Lewerentz and T. Lindner. *The Application of Formal Software Development : A Comparative Case Study*. Forschungszentrum Informatik, Department for Software Engineering, Haid-und-Neu-Strasse 10-14, D-7631 Karlsruhe, Germany. URL: ftp://ftp.fzi.de/pub/korso/production_cell/papers/Short.ps
- [14] K. Lano and H. Haughton. Reuse and adaptation of Z specifications. In J. P. Bowen and J. E. Nicholls (eds), *Z User Workshop 1992*, Workshops in Computing, pp 62-90, Springer-Verlag 1992.
- [15] S. M. Merad. FOOSA (Formal object oriented systems analysis) Phase 1 Issue 1.0. Technical report, DRA, Malvern, May 1992.
- [16] N. Plat. *Experiments with Formal Methods in Software Engineering*. PhD. Thesis, Technical University of Delft, The Netherlands, 1993.
- [17] M. G. Rawson and P. M. Allen. Objects in Z: A formalization of the HOOD (parent/child) object decomposition. Technical Report SCM 93/2, University of Huddersfield, Queensgate, Huddersfield, 1993.
- [18] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented Modeling and Design*. Prentice-Hall, 1991.
- [19] S. Schuman and D. Pitt. Object-oriented subsystem specification. In L. Meertens (ed), *Program Specification and Transformation*, pp. 313-341, North-Holland, 1987.
- [20] L. T. Semmens and P. M. Allen. Using Yourdon and Z: An approach to formal specification. In J. Nicholls, editor, *Proceedings of the Fifth Z User Workshop*. Springer Verlag, December 1990.
- [21] L. T. Semmens and P. M. Allen. Formalizing Yourdon. Methods Integration Workshop, Leeds Metropolitan University, September 1991.

- [22] L. T. Semmens, R. B. France, and T. W. G. Docker. Integrated structured analysis and formal specification techniques. *The Computer Journal*, 35(6):600–610, 1992.
- [23] Sally Shlaer and Stephen J. Mellor. *Object Lifecycles: Modelling the World in States*. Yourdon Press Computing Series. Prentice-Hall, 1992.
- [24] G. Smith. An object-oriented development framework for Z. In J. P. Bowen and J. A. Hall (eds) *Z User Workshop, Cambridge 1994*, Springer-Verlag 1994.
- [25] J. M. Spivey. *The Z Notation: A Reference Manual*. (Second edition) Prentice-Hall, 1992.
- [26] S. Stepney, R. Barden and D. Cooper (eds.). *Object Orientation in Z*. Workshops in Computing, Springer-Verlag, 1992.
- [27] P. Wysall and J. McDermid. An approach to object oriented specification using Z. In J. Nicholls (ed.), *Z User workshop, Oxford 1990*. Springer-Verlag, 1990.
- [28] J. B. Wordsworth. *Software Development with Z* Addison-Wesley, 1992.
- [29] Edward Yourdon. *Modern Structured Analysis*. Prentice-Hall, 1989.