Guidelines for Formalizing Fusion Object-Oriented Analysis Models

B. W. Bates, J-M. Bruel, R. B. France*, and M. M. Larrondo-Petrie

Department of Computer Science & Engineering
Florida Atlantic University
Boca Raton, FL-33431, USA.
Email: {batesb,bruel,robert,maria}@cse.fau.edu

Abstract. The growing interest in object-oriented analysis and design methods (OOMs) in the software development industry can be attributed to the support they give to some of the more significant software engineering principles, for example, separation of concerns and generality. On the other hand, most OOMs, like their structured analysis and design predecessors, produce models that are not amenable to rigorous semantic analyses. This problem can be attributed to the lack of firm semantic bases for the modeling notations and concepts. In this paper we show how a particular OOM, the *Fusion* analysis method, can be made more formal while preserving its essential qualities. Our approach involves integrating the *Z* formal specification style with the Fusion method. The result is an OOM that produces semantically analyzable Fusion models of behavior at the requirements level.

Keywords: Formal Specification Techniques, Object-Oriented Analysis, Transformations.

1 Introduction

As software developers grapple with the problem of producing high quality software, attention has turned to object-oriented methods (OOMs), mainly because of the support they give to some of the more significant software engineering principles, in particular, separation of concerns, design for change, anticipation of change, and incrementality (e.g., see [4, 18, 21]). As was the case with earlier graphical structured approaches, industry's growing interest in OOMs can also be attributed to the use of simple, visually-appealing concepts and notation. On the other hand, like earlier structured analysis and design (SA/D) approaches (e.g., see [5, 28, 29]), most popular OOMs lack formal semantic bases, making it very difficult to rigorously reason about and with the models they produce. In this sense, most OOMs are *informal* because their applications are likely to produce ambiguous specifications that are not amenable to rigorous semantic analyses.

The lack of semantic bases in most OOMs severely inhibits their use in the specification and analysis of complex information systems. The need to rigorously investigate the behavior of such systems suggests the use of formal specification techniques (FSTs). A FST consists of a formal language (i.e., a language with a precise syntax and semantics), and mechanisms for deriving consequences from specifications expressed in the formal language. FSTs utilize mathematical concepts and notation to precisely define theories and models of application behavior. Precise specifications facilitate effective communication among persons with a stake in the development of the software, while the ability to reason about specified properties allows developers to predict the behavior of implementations during the specification phases of software development.

Other object-oriented and SA/SD methods have been integrated in a formal environment (e.g., see [3, 10, 14, 16, 20]). In this paper we describe some guidelines resulting from our work on formalizing Fusion [4], an OOM developed in industry. We chose Fusion because it incorporates some of the best object-oriented modeling ideas from previous OOMs in a single coherent method. The approach we used can be used to formalize other OOMs as well. We show how Fusion can be integrated with Z [24], a popular and mature FST. We chose Z because of its maturity, expressiveness and the availability of tools (e.g., CADiZ [25], *f*UZZ [23], and `ProofPower` [13]). The integrated specification method that we created can be used to develop Fusion analysis models that are amenable to rigorous semantic analysis.

In Section 2 we discuss the benefits of integrating informal specification techniques (ISTs) with FSTs, and give an overview of Z and Fusion. Section 3 provides the rules supporting the translation of Fusion analysis models to Z specifications. Finally, Section 4 gives our conclusions and an overview of our ongoing work in this area.

* This work was partially funded by NSF grant CCR-9410396.

2 Integrating formal and informal specification techniques

The application of OOMs can be made more rigorous by providing formal foundations supporting the semantic analysis of the models. One approach is to interpret the models using underlying formal semantic models. The formal interpretation can be used to reason about and with the models. We describe this approach as *interpretive*. In using this approach one has to make an effort to validate the semantic model. Validation in this sense means ensuring that the semantic model is "consistent" with the intuitive interpretation of the models, that is, that the semantic model does not result in a modeling language that requires the modeler to learn new meanings for old constructs.

Another approach is to use the OOM with a suitable FST. The integration of an OOM and a FST is an instance of a class of specification techniques that we call *integrated formal/informal specification techniques* (FISTs) [6, 7, 8]. The use of FISTs can be beneficial in the following respects:

- *FISTs enable an evolutionary approach to the use of FSTs in industry.* FISTs allow an organization to preserve, and even enhance, its investment in ISTs while taking advantage of FST-related benefits.
- *FSTs and graphical ISTs can complement each other.* The relatively simple and graphical nature of IST specifications often makes them more presentable than the more detailed, often textual, formal specifications. On the other hand, the lack of firm semantic bases for ISTs inhibits their use in rigorous specification and analysis of behavior. FSTs are needed for such activities.

FISTs can be classified as either *supplemental* or *interpretive*, depending on the relative roles of the formal and informal models. In a supplemental FIST the formal and informal concepts supplement each other, that is, the formal and informal specifications developed using the FIST capture complementary aspects of a system's functionality. An example of a supplementary FIST involving the Fusion method [4] and Z [24] is one in which Z is used to express operation specifications, and Fusion models are used to express the static and life-cycle aspects of applications.

In an interpretive FIST, the formal model provides a more precise description of behavior captured by the informal model. In some cases an interpretive FIST can have the same results as the interpretive approach discussed earlier in this section. This happens when there is a formal set of rules that map IST modeling constructs to FST constructs. The existence of formal mappings between the notations is not a necessary part of interpretive FSTs. Formalization of the informal models can be done in a cognitive manner, that is, it can be based on a developer's understanding of the application area and of the intent captured in the informal models. A consequence of this cognitive approach to formalization [6] is that there may not always be a clear connection between parts of the formal model and the informal model, for example, the formal models may have details that are not representable in the informal notation.

The manner in which FSTs and ISTs are integrated in a FIST is usually based on one or more of the following considerations:

- *Preservation of the 'intuitive' interpretation of the informal specification*: The integration should be such that the formal specifications provide interpretations of the informal specifications that are consistent with intuitively-held interpretations of the informal specifications. A FIST that does not support intuitively-held interpretations of its informal specifications may require that the informal specifications be developed in ways that are not consistent with practices associated with the use of the ISTs. In such cases, a specifier learning and using the FIST is not likely to take full advantage of his prior experience with the ISTs.
- *Level of automated support for moving between formal and informal specifications*: Developing formal specifications can be a very arduous task. Well-defined relationships between FST and IST constructs and concepts can provide the basis for mechanical generation of some parts of the formal specification. In the other direction, it is often possible, given a well-defined relationship between formal and informal specification elements, to mechanically generate informal specifications from formal specifications. This transformation is often easier to automate because it usually entails simply choosing what information to hide.
- *Degree of integration*: In some cases it may not be worthwhile to formally interpret all aspects of an informal specification. It may be sufficient to formalize only those parts that can benefit from more rigorous specification and analysis.

In this paper we describe an interpretive FIST that integrates Fusion analysis models [4] with the Z specification style [24]. Transformation rules are defined for some aspects of the formalization, some of which can be mechanized. We made every attempt to preserve the intuitive interpretation of Fusion models in our formalization; empirical investigation is still needed to validate the integration in this sense. In the following subsections we give an overview of Fusion and Z.

2.1 Overview of Fusion

Fusion [4] is an object-oriented software development methodology that combines and extends existing techniques, e.g. Rumbaugh's Objet Model Technique (OMT) [18], Booch's technique [2], Wirfs-Brock's Class Responsibility Collaborator [26] (CRC) technique, and Jacobson's Objectory [12]. Fusion claims to take the best ideas from these methods and incorporate them into a single coherent method that covers software analysis, design, and implementation. In this section we present an overview of the Fusion analysis process and products.

In Fusion's analysis phase the required behavior of the system is described by

- an Object Model that defines the static structure of the information manipulated by the application in terms of classes and the relationships among them; and
- an Interface Model that defines the externally observable behavior of the application. The Interface Model, in turn, consists of two models:

  Operation Model: This model characterizes the effect of application services in terms of the observable state changes they make and the output events they send to the environment.

  Life-Cycle Model: This model characterizes the allowable sequences of service invocations for the application.

A data dictionary is maintained and updated throughout the development process.

The services identified in the interface model are not associated with class operations at the analysis level. In fact, Fusion does not attach operations to classes in this phase; this is done during design.

The notation used in the Object Model is summarized in Fig. 1. There are three different types of associations in Fusion:

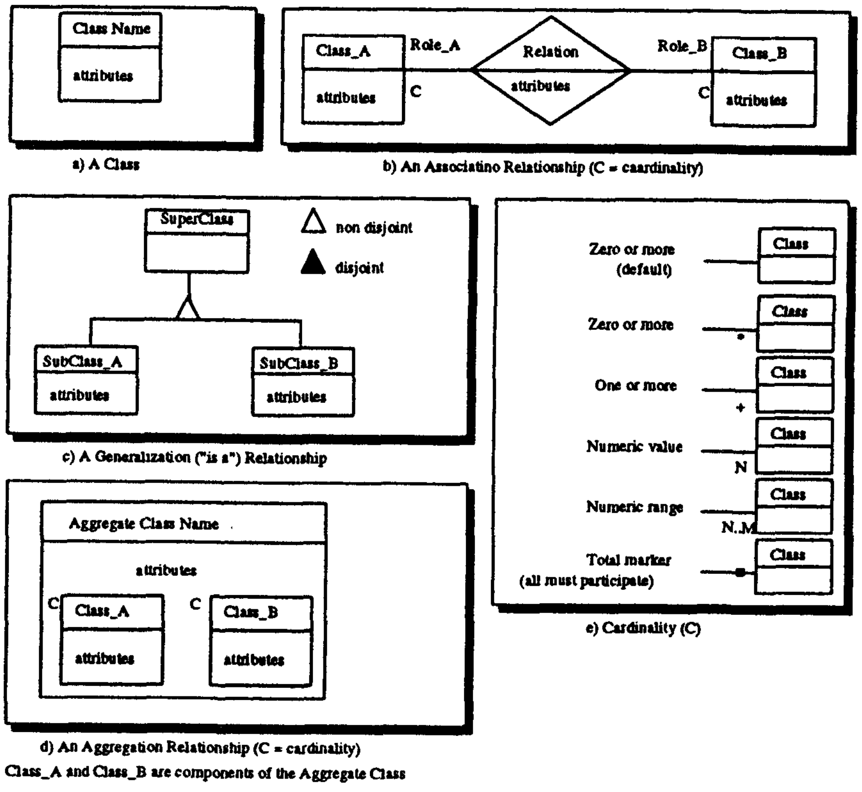


Fig. 1. Summary of Fusion Object Model Notation

Generalization: A triangle symbol $\triangle$ symbolizes the "is a" relationship between two classes (see Fig. 1.c). The subclasses may partition the superclass (i.e., can be disjoint), or they may overlap (i.e., can be non disjoint). The disjoint subclasses are joined to their superclass with a filled-in triangle $\blacktriangle$.

Aggregation: The relationship between an aggregate class and its component classes, is shown as classes embedded in the aggregate class (see Fig. 1.d). Cardinality is shown adjacent to each component class. The symbols used for cardinality are given in Fig. 1.e.

Relationship: This represents more general forms of associations between classes. The relationship is assigned a name and can have attributes associated with it (see Fig. 1.b).

A filled-in rectangle or "marker" on a relationship means that all members must participate in the relationship. Relationships can be annotated with assertions (called invariants) that stipulate that certain properties must be maintained, and roles, which indicate the role objects play in the relationships. Invariants can also be associated with individual classes.

The Fusion Operation Model consists of a set of operation schemas. Each schema consists of the following parts:

Description: an informal and concise description of the operation.
Reads: a list of the items that the operation reads.
Changes: a list of the items the operation changes.
Sends: a list of the events the operation sends to other objects in the environment.
Assumes: a condition that describes what is assumed true at the start of the operation.
Result: describes what is true after the operation has completed its execution.

The Life-Cycle Model is expressed in a language that allows one to express relationships such as "event x is followed by event y", "either event x or event y occurs", "event x is optional", "the steps of events x and y are interleaved".

More detailed descriptions of Fusion notation will be given when needed in the remainder of this paper.

2.2 The Z Notation

The Z specification language [11, 24] is a general purpose specification language which was developed by the Programming Research Group at Oxford. It has a strong mathematical basis in predicate logic and set theory. Static aspects of applications are represented by sets and dynamic aspects by operations on sets.

In this section we introduce only the Z notations necessary to understand the specifications given in this paper[2].

The primary structuring construct in Z is the schema. A schema has two parts: a declaration and a predicate part. The declaration part consists of variable declarations of the form $w : \mathit{Type}$, where w is a variable name and Type is a type name. Intuitively, the preceding declaration means that the value of w is a member of the set named by Type (types are sets in Z). The predicate part consists of a predicate that expresses the relationships among the declared variables.

There are two ways of writing schemas, vertically or horizontally:

```
┌─ Schema ──────────────────────
│ Declaration
├───────────────
│ Predicate
└───────────────────────────────
```

$$\mathit{Schema} \mathrel{\widehat{=}} [\, \mathit{Declaration} \mid \mathit{Predicate} \,]$$

Schemas are used to model both the static and dynamic aspects of a system. A schema that captures the static aspect of a system will be referred to as a state schema, and a schema that captures the dynamic aspect will be referred to as an operation schema. In a state schema the components of a system's state are declared in the declaration section and constraints on the state are given in the predicate part.

[2] The reader is invited to see [24] or the URL: `http://www.comlab.ox.ac.uk/archive/z.html`.

The declaration $w : Schema$, where *Schema* is a schema name, declares a variable w with components (declared in *Schema*) that satisfy the predicate part of *Schema*.

Let S be a schema defined as:

```
┌─ S ──────────────────────
│ x : ℕ; y : ℙℕ
├──────────────
│ x ≤ #y
└──────────────────────────
```

for the declaration $w : S$

$w.x$ denotes the projection function: $w.x$ is w's x component,
θS is a tuple consisting of (schema variable, value) pairs, where the values are said to be bound to the schema variables (commonly used to equate the before and after state components of an operation),
pred S is the predicate part of a schema.

An operation schema defines the relationship between the state at the start and at the end of an operation's execution. The declaration part of an operation schema declares variables representing the before and after state, inputs, outputs, and any other variables needed to define the relationship. The predicate part of the schema defines the relationship between the before and after states.

The following conventions are used for variable names in operation schemas:

unprimed variable (e.g., w) - value of variable before operation execution;
primed variable (e.g., w') - value of variable after operation execution;
variable ending in '?' - an input to the operation; and
variable ending in '!' - an output from the operation.

$\Delta S == S \wedge S'$ denotes the change of state schema,
$\Xi S == [\Delta S \mid \theta S' = \theta S]$ denotes that the state schema does not change.

3 Fusion to Z translation guidelines

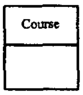
3.1 Object Model

In examining the Fusion Object Model it was noticed that the model has three major classes of constructs: objects (classes), relations, and annotations (invariants). In the following subsections we give rules guiding the translation of instances of these constructs to Z specifications.

Representing objects in Z In Fusion, an object in an Object Model can have zero or more attributes associated with it. The attributes are values, not objects.

Objects that have no attributes associated with them are represented as Z schemas that contain only an object identifier.

Rule 1 (attributeless objects) *Objects that contain no attributes are represented by Z schemas. An object identifier attribute is defined in order to identify the instances of the class.*

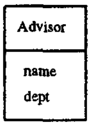


```
[CRID]
┌─ Course ─────────────────
│ id : CRID
└──────────────────────────
```

Fig. 2. Formal specification of a class without attributes

Objects with attributes are represented by Z state schemas, in which the attributes are declared as state variables, and any invariant annotations are expressed formally in the predicate part (e.g., see Fig. 3).

Rule 2 *Objects with attributes are made into state schemas. Attributes are declared as state variables, and invariants (if any) are formally expressed in the predicate part. If an attribute is not associated with a type, the capitalized name of the attribute is used as a type name and declared as a basic Z type. Each object instance has a unique identifier that is explicitly defined through an attribute.*



$[ADVID, NAME, DEPT]$

$$\begin{array}{|l}\multicolumn{1}{l}{Advisor}\\ \hline id : ADVID \\ name : NAME \\ dept : DEPT \\ \hline\end{array}$$

Fig. 3. Formal specification of a class with attribute

Operations that modify the state of a system may only affect a subset of the variables. In Z one has to specify that state variables that are not affected by operations are left unchanged, for example, if state variable x is unaffected by an operation's execution then the equation $x' = x$ should appear in the predicate part of the operation schema. If an operation affects only a small part of the state consisting of a large number of variables, then numerous equations of the above form need to be written. One can structure state schemas in a way that minimizes the need to write a large number of such equations.

The rule below provides some guidelines for structuring schemas to minimize the impact of this problem. It utilizes schema inclusion to achieve structuring. When a schema A is included in another schema B then B's declaration part includes all the declarations in A, and the predicate part of A is logically 'anded' to the predicate part of B. The rule involves identifying sets of attributes of a class that are unaffected by a group of operations that act on the instances of the class. These attributes are defined in separate schemas which are included in the schema representing the class (e.g., see Fig. 4).

Rule 3 (schema structuring) *Determine sets of attributes that are not changed often by operations and declare them in separate schemas. The schema representing the class is then formed by including these schemas in the declaration part along with any attributes that are not declared in any of the included schemas.*

Rule 4 (instances) *Instances of classes are represented as bindings of values to variables declared in Z schemas representing the classes. The identifier of an instance is the value bound to the identifier variable (id) declared in the Z schema for the instance class (see Fig. 5).*

Representing relationships in Z We created a mathematical toolkit that includes Z definitions of the different types of general binary relationships that can exist between objects in an Object Model. The types are defined by parameterized Z schemas. We utilize graphical symbols in the toolkit to make clear the connection between formalizations and the Fusion representations of relationships. In Fig. 6 we give an example of a toolkit formalization of a Fusion relationship and show how it is used to represent a particular relationship in a model. If the relationship is associated with a cardinality other than 0 and 1, then the cardinality constraint is expressed in the predicate part of the schema (e.g., see Fig. 7).

Rule 5 *Relationships are represented by variables declared in a state schema. The type associated with a relationship variable is obtained from the mathematical toolkit defining standard relationship types (many, 0, 1). If the relationship has a cardinality other than many, 0, and 1 then the cardinality constraint is expressed in the predicate part of the schema.*

Without schema structuring:

```
┌─ STUDENT ─────────────────────────
│ id : STID
│ name : NAME
│ address : ADDRESS
│ ssnum : SS
│ gpa : GPA
│ courseenrolled : ℙ COURSE
│ ...
└───────────────────────────────────
```

```
┌─ AddCourse ───────────────────────
│ ΔSTUDENT
│ c? : COURSE
├──────────
│ c? ∉ courseenrolled
│ courseenrolled' = courseenrolled ∪ {c?}
│ id' = id
│ name' = name
│ address' = address
│ ssnum' = ssnum
│ ...
└───────────────────────────────────
```

After schema structuring:

```
┌─ StudentPart ─────────────────────
│ id : STID
│ name : NAME
│ address : ADDRESS
│ ssnum : SS
└───────────────────────────────────
```

```
┌─ STUDENT ─────────────────────────
│ StudentPart
│ gpa : GPA
│ courseenrolled : ℙ COURSE
│ ...
└───────────────────────────────────
```

```
┌─ AddCourse ───────────────────────
│ ΔSTUDENT
│ c? : COURSE
├──────────
│ c? ∉ courseenrolled
│ courseenrolled' = courseenrolled ∪ {c?}
│ ...
│ θStudentPart' = θStudentPart
└───────────────────────────────────
```

Fig. 4. Illustration of schema structuring

Advisor	
name	dept
Robert	CSE
Maria	CSE

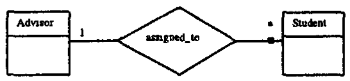
```
┌──────────────────────────────────
│ advisors : ℙ Advisor
│ a1, a2 : Advisor
├──────────
│ a1.id = 01
│ a1.name = Robert
│ a1.dept = CSE
│ a2.id = 02
│ a2.name = Maria
│ a2.dept = CSE
│ advisors = {a1, a2}
└──────────────────────────────────
```

Fig. 5. Particular instances representation

Fig. 8 shows how relationships with attributes are represented in Z.

Rule 6 (attributes) *Relationship attributes are handled by declaring the attributes as basic types (if only one) or schemas. A function that maps the relationship to its attributes is then defined in the relationship schema.*

N-ary relationships are represented in Z as variables of n-tuple types, where the classes (sequences of instances) involved in the tuple are the components of the tuple type. Cardinality constraints are expressed in the predicate part of the relationship schema (e.g., see Fig. 9).



$[X, Y]$

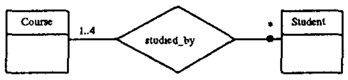
$1 \longleftrightarrow * : \mathbb{P}(X \leftrightarrow Y)$

$\forall R : X \leftrightarrow Y \bullet$
$\quad R \in 1 \longleftrightarrow * \Rightarrow R^{-1} \in (Y \rightarrow X)$

$advisors : \mathbb{P}\,Advisor$
$students : \mathbb{P}\,Student$
$assigned\_to : 1 \longleftrightarrow * \,[Advisor, Student]$

$\mathrm{dom}\; assigned\_to \subseteq advisors$
$\mathrm{ran}\; assigned\_to \subseteq students$
$\forall i, j : students \bullet i.id = j.id \Leftrightarrow i = j$
$\forall i, j : advisors \bullet i.id = j.id \Leftrightarrow i = j$

Fig. 6. Formal specification of a binary relationship



Relationship

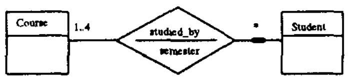
$students : \mathbb{P}\,Student$
$courses : \mathbb{P}\,Course$
$studied\_by : Student \leftrightarrow Course$

$\forall s : \mathrm{dom}\; studied\_by \bullet$
$\quad 1 \leq \#studied\_by(\!|\, \{s\} \,|\!) \leq 4$
$\mathrm{dom}\; studied\_by \subseteq students$
$\mathrm{ran}\; studied\_by \subseteq courses$
$\forall i, j : students \bullet i.id = j.id \Leftrightarrow i = j$
$\forall i, j : courses \bullet i.id = j.id \Leftrightarrow i = j$

Fig. 7. Integration of the cardinality constraints

Rule 7 (n-ary relationship) *Ternary or higher relationships are handled by modeling the relationship as a variable with an n-tuple type, where each class in the relation is a component of the type. Cardinality constraints are expressed in the predicate part of relationship schema.*

Representing aggregation structures An aggregate object is represented in Z by a schema in which the variables representing sets of component parts and relationships among the component parts are declared (e.g., see Fig. 10).



Relationship

$students : \mathbb{P}\,Student$
$courses : \mathbb{P}\,Course$
$studied\_by : Student \leftrightarrow Course$
$semester : (Students \leftrightarrow Course) \rightarrow SEMESTER$

$\forall s : \mathrm{dom}\; studied\_by \bullet$
$\quad 1 \leq \#studied\_by(\!|\, \{s\} \,|\!) \leq 4$
$\mathrm{dom}\; studied\_by \subseteq students$
$\mathrm{ran}\; studied\_by \subseteq courses$
$\forall i, j : students \bullet i.id = j.id \Leftrightarrow i = j$
$\forall i, j : courses \bullet i.id = j.id \Leftrightarrow i = j$
$\mathrm{dom}\; semester = studied\_by$

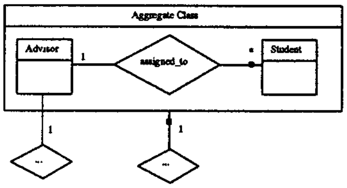
Fig. 8. Relationship with attributes

The schema represents a ternary relationship called *takes* involving the classes Student, Room, and Test.

```
Relationship
takes : ℙ(ℙ Student × ℙ Room × ℙ Test)
---
...
```

Fig. 9. Formalization of a ternary relationship

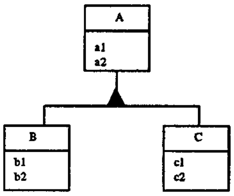
Rule 8 *Aggregation is handled by declaring components as sets containing elements of the component schemas types in the aggregate class schema. Relationships among the components are expressed as described by the rules on formalizing Fusion relationships.*



```
Aggregate_Class
id : AGGID
advisors : ℙ Advisor
students : ℙ Student
assigned_to : 1 ←——→→ * [Advisor, Student]
...
---
...
```

Fig. 10. Formalization of an aggregation

Representing generalization hierarchies A generalization hierarchy is represented in Z by including superclass schemas in subclass schemas. See Fig. 11 for an example.



```
A
id : ID
a1 : A1
a2 : A2
---
...
```

$A\_attributes == A \setminus (id)$

```
B                         C
A_attributes              A_attributes
id : ID                   id : ID
b1 : B1                   c1 : C1
b2 : B2                   c2 : C2
...                       ...
---                       ---
...                       ...
```

Fig. 11. Generalization hierarchy formalization

Putting it all together Once the parts of the Object Model are defined they are collected in a schema that is a representation of the model. Invariants involving parts that were separately defined are expressed in the predicate part of this schema.

Our rules provide a bottom-up approach to formalizing an Object Model. Some restructuring of the schemas and/or redefinition of the parts may be needed to reduce the complexity of the schemas produced by the rules.

3.2 Formalizing the Interface Model

We formalize only the Operation Model. The Fusion Life-Cycle Model is rigorous and analyzable; reexpression of this model in Z is not necessary.

Representing Operation Models Fusion Operation Models are translated to Z operation schemas in the following manner:

Rule 9 *The following are rules guiding the translation of a Fusion Operation schema to a Z operation schema:*

- *The operation name is used as the schema name.*
- *The informal definition is used to document the Z operation schema.*
- *Variables mentioned in the Reads section are declared as input variables in the Z schema.*
- *If variables are mentioned in the Changes section then the state is declared in the operation schema preceded by the Δ symbol (indicating that the operation causes a change in the state). Equations for variables not changed by the operations (i.e., not mentioned in the Changes section) are generated. For example, if state variable x is not mentioned in the Changes section then the equation $x' = x$ is generated; if no variable in the state schema, S, in which x is declared is mentioned in the Changes section then the equation $\theta S' = \theta S$ is generated.*
- *Variables mentioned in the Sends section are declared as output variables in the Z operation schema.*
- *The condition in the Assumes section, when formalized, is the precondition of the operation.*
- *the formalized content of the Result is given in the predicate part of the operation schema.*

In Fig. 12 the above rule is applied to a Fusion Operation Model.

Operation: *view_WS*
Description: A request to view a worksheet.

Reads: supplied *student* : *stud_id*
Changes: none
Sends: *student* : *worksheet advisor* : *worksheet sys_adm* : *worksheet univ_adm* : *worksheet*
Assumes: *stud_id* is valid and a student can only view their own worksheet.
Result: Displays the student's worksheet.

view_WS

$id? : STID$
$ws! : Worksheet$
$\Xi AdvState$

$\exists s : students \bullet$
$\quad s.id = id? \Rightarrow ws! = getsheet(s)$

Fig. 12. Illustration of an operation translation

4 Conclusion

The use of OOMs in the development of complex information systems can be inhibited by their lack of formal semantic bases. In this this paper we discuss how a particular object-oriented analysis method, Fusion, can be integrated with a FST, Z, to produce analyzable object-oriented analysis models. A case study on an application of the Fusion/Z FIST described in this paper can be found in [1].

Our FIST approach can be abstracted as shown in Fig. 13, where the boxes represent models and the ovals represent activities.

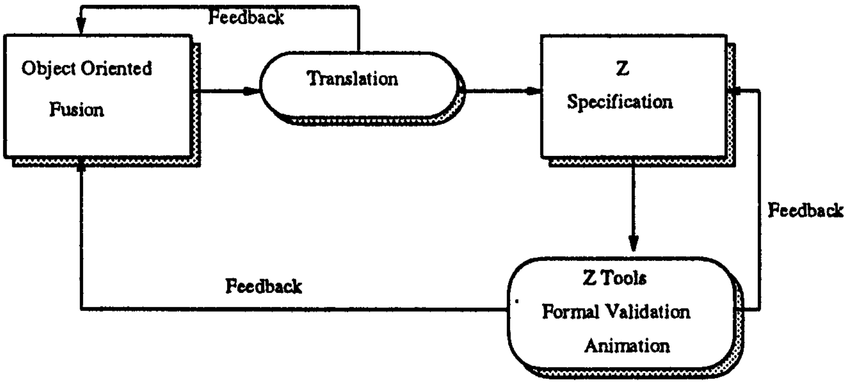


Fig. 13. The integration process

The translation activity is guided by the rules we stated in the previous section. Currently the translation process is manual, but we are working on formalizing the rules in order to identify those that can be mechanized to support partial automation of the translation.

In our experimentations with the Fusion/Z FIST[3], we found that formalizing the Fusion models helped identify problems with the object model, for example, inconsistent use of attribute names, undefined variables, and missing operation arguments. More generally, formalization revealed ambiguities, gaps in our knowledge of the systems being modeled, and inconsistent requirements. The feedback from the translation activity can be used to improve the Fusion models (as indicated by the feedback arrow leaving the translation activity in the figure).

The primary objective of integrating Fusion and Z is to create analyzable Fusion models. The integration allows one to use Z type checkers (e.g., ZTC [27] and *f*UZZ [23]) and Z animation tools, such as ZANS, to analyze models. In our experimentations, we used ZTC, *f*UZZ and ZANS and found that doing such analyses can reveal additional problems with the Fusion models (as well as the formalization of the problem).

From the above, it should be clear that the process of formalizing an informal OOM model is an iterative process. Formalization can help identify inadequacies in the informal model, which results in changes being made to the models. Formalization of the changed models is then carried out, and the process is repeated until a formalization that is consistent with the intent captured in the informal models is obtained.

It can be argued that an object-oriented version of Z such as Object-Z [22], MooZ [15] or SP-Z[19], would be a more appropriate formalism to integrate with an OOM. Unlike Z, the object-oriented versions of Z do not have sufficient tool support as yet. One of the reasons we chose Z is the availability of analysis tools. A goal of our research in this area is the development of a CASE tool supporting the creation and analysis of analyzable Fusion analysis models. It is important that the formal notation we use is adequately supported by analysis tools.

We anticipate that tools for analyzing object-oriented versions of Z will soon be available. For this reason we are currently defining rules to guide the integration of Fusion models with object-oriented versions of Z. One of the formalisms that we are looking at is Hall's object-oriented style [9]. Our derived Z specification has some common points with Hall's style, for example, our treatment of identity is based on ideas from Hall's approach. We should point out that it is not clear to us at this time whether there is a significant advantage in using an object-oriented style of Z with Fusion. The integrated approach described in this paper has been adequate for the problems we applied them to. One the objectives of our work on integrating Fusion with an object-oriented style of Z is to determine whether the object-oriented Z concepts have a significant impact on formalizing and analyzing Fusion models.

[3] see [1] for a fuller account of our experiences

References

1. Brian W. Bates, Jean-Michel Bruel, Robert B. France, and Maria M. Larrondo-Petrie. Experiences with Formalizing Fusion Object-Oriented Analysis Models. FAU Technical Report TR-CSE-95-44, Department of Computer Science & Engineering, Florida Atlantic University, Boca Raton, FL–33431, USA, November 1995.
2. G. Booch. *Object-oriented analysis and design with applications.* Benjamin/Cummings, 1994.
3. Robert H. Bourdeau and Betty H.C. Cheng. A formal semantics for object model diagrams. *IEEE Transactions on Software Engineering*, 21(10):799–821, October 1995.
4. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method.* Prentice Hall, 1994.
5. T. DeMarco. *Structured Analysis and System Specification.* Prentice-Hall, 1978.
6. R. France and M. M. Larrondo-Petrie. A two-dimensional view of integrated formal and informal specification techniques. In *ZUM'95, Lecture Notes in Computer Science 967.* Springer-Verlag, 1995.
7. R. France and M. M. Larrondo-Petrie. Understanding the role of formal specification techniques in requirements engineering. In *in Proceedings of The 8th SEI Conference on Software Engineering Education, Lecture Notes in Computer Science 895.* Springer-Verlag, 1995, pages 207-222.
8. R. B. France and M. M. Larrondo-Petrie. From structured analysis to formal specifications: State of the theory. In *Proceedings of the 1994 ACM Computer Science Conference.* ACM, 1994.
9. J. A. Hall. Using Z as a specification calculus for object-oriented systems. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM and Z - Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 290–318. VDM-Europe, Springer-Verlag, 1990.
10. T. C. Hartrum and P. D. Bailor. Teaching formal extensions of informal-based object-oriented analysis methodologies. In *Proc. Computer Science Education*, pages 389–409, 1994.
11. I. J. Hayes, editor. *Specification Case Studies.* Prentice Hall International Series in Computer Science, 2nd edition, 1993.
12. I. Jacobson. *Object oriented software engineering.* Addison-Wesley, 1992.
13. R. B. Jones. ICL ProofPower. *BCS FACS FACTS*, Series III, 1(1):10–13, Winter 1992.
14. K. C. Mander and F. Polack. Rigorous specification using structured systems analysis and Z. *Information and Software Technology*, 37(5):285–291, May 1995.
15. S. L. Meira and A. L. C. Cavalcanti. Modular object-oriented Z specifications. In Nicholls [17], pages 173–192.
16. John Murphy and Jane Grimson. The Jupiter System: A Prototype for Multidatabase Interoperability. In *Proceedings of the 12th British National Conference on Databases, BNCOD-12*, Lecture Notes in Computer Science. Springer-Verlag, 1994.
17. J. E. Nicholls, editor. Workshops in Computing. Springer-Verlag, 1991.
18. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.
19. S. A. Schuman, D. H. Pitt, and P. J. Byers. Object-oriented process specification. In C. Rattray, editor, *Specification and Verification of Concurrent Systems*, Workshops in Computing, pages 21–70. Springer-Verlag, 1990.
20. Lesley T. Semmens and Pat M. Allen. Using Yourdon and Z: An approach to formal specification. In Nicholls [17], pages 228–253.
21. S. Shlaer and S. J. Mellor. *Object lifecycles: Modeling the world in states.* Prentice Hall, 1992.
22. Graeme Smith. *An Object-Oriented Approach to Formal Specification.* PhD thesis, Department of Computer Science, University of Queensland, St. Lucia 4072, Australia, October 1992.
23. J. M. Spivey. *The fUZZ Manual.* Computing Science Consultancy, 34 Westlands Grove, Stockton Lane, York YO3 0EF, UK, 2nd edition, July 1992.
24. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.
25. I. Toyn. *CADiZ Quick Reference Guide.* York Software Engineering Ltd, University of York, York YO1 5DD, UK, 1990.
26. R. Wirfs-Brock and B. Wilkerson. *Designing object oriented software.* Prentice-Hall, 1990.
27. Xiaoping Jia. *ZTC: A Type Checker for Z - User's Guide.* Institute for Software Engineering, Department of Computer Science and Information Systems, DePaul University, Chicago, IL 60604, USA, 1994.
28. E. Yourdon. *Modern Systems Analysis.* Prentice-Hall, 1989.
29. E. Yourdon and L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Prentice-Hall, 1979.