ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**D Duke, University of York, UK and A S Evans, University of Bradford, UK
(Eds)**

BCS-FACS Northern Formal Methods Workshop

Proceedings of the BCS-FACS Northern Formal Methods Workshop,
Ilkley, UK. 23-24 September 1996

CASE-Based Rigorous Object-Oriented Modeling

J.-M. Bruel, R.B. France, M.M. Larrondo-Petrie, B. Chintapally
and G.K. Raghavan

Published in collaboration with the
British Computer Society

CASE-based Rigorous Object-Oriented Modeling

J.-M. Bruel, R. B. France, M. M. Larrondo-Petrie,
B. Chintapally and G. K. Raghavan*

Department of Computer Science & Engineering

Florida Atlantic University

Boca Raton, FL-33431, USA.

Phone: (407)-367-3855

Fax: (407)-367-2800

Contact: `robert@cse.fau.edu`

Abstract

The use of a rigorous development environment is needed when developing a complex system, such as real-time and critical systems. Preventing industry from embracing formal method techniques is the lack of industrial strength tools to support formalization and analysis. We are developing an approach and a tool environment for the development of complex systems that integrates a graphical object-oriented technique (Fusion) and a formal method (Z) with analysis tools. The tool, called FuZE (Fusion/Z Environment), has been used successfully to specify and analyze non-trivial systems. This paper describes the specification transformation rules and the extension of the environment to include transformation of design documents such as Object Interaction Graphs.

1 Introduction

Despite their strengths, object-oriented methods (OOMs) suffer from a lack of firm semantic bases, which makes it difficult to precisely express and rigorously reason about requirements and designs. In particular, these limitations severely hamper the use of OOMs in the development of critical and distributed systems. Formal methods give the capability to conduct rigorous analysis but their mathematical notation is cumbersome to read and the lack of available industrial strength tools prevent them from being more prevalent in industry.

In previous papers (e.g., see [1, 6]) we described our work on formalizing the Fusion [4] object-oriented analysis modeling techniques to make the models produced more amenable to formal analysis. We chose the Fusion method because it incorporates some of the best object-oriented modeling concepts from previous OOMs, and the formal notation Z [10] because of its maturity, expressiveness, the availability of analysis tools, and its apparent compatibility with Fusion analysis modeling concepts.

In our approach, formal rules are used to transform Object Model structures into Z specifications, and guidelines are given for formalizing operation schemas. We have implemented the transformation rules for the Object Model in a tool that we are currently developing called FuZE (Fusion/Z Environment) [2] which also provides automated support for rigorous analysis of Fusion analysis models.

In this paper we present an improved version of the Fusion/Z transformation rules and discuss their implementation in FuZE. We also discuss our ongoing work on formalizing Fusion design models, and present a formalization of Object Interaction Graphs.

There have been several Z-based object-oriented languages proposed, such as FOOM [11]. However, in our approach we choose not to use any of these extensions, instead we use the Z notation as recommended by the people working around the *Open Distributed Processing* [8]. This allows us to benefit of the support

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

Figure 1: An integration model

of Z in terms of tools, and avoid the introduction at the analysis level of design choices, as would be the case if we used techniques such as Object-Z [5], where operations have to be attached to a particular class at the requirements stage.

In section 2 we present the improved transformation rules. Section 3 gives an overview of FuZE. Section 4 and 5 propose rules for formalizing object interaction graphs. Section 6 summarizes the results and future work.

2 Formalizing Fusion Object Models

In this section we present the improved versions of the rules. The improvements concern the formalization of generalization/specialization hierarchies and the manner in which the Z formalization of the Fusion Object Model is incrementally developed.

2.1 Representing classes in Z

In our approach, a class structure is represented by Z state schemas. The attributes of a class and object identifiers of a class (implicitly defined in the Object Model) are represented by state variables. Any invariants associated with the class (expressed in the form of annotations in the Object Model) are expressed formally in the predicate part of the schema representation. The type name associated with an attribute of a class in the Object Model is used to define the attribute in the Z schema for the class (the type must also be defined in Z, for example, as a basic type, or a schema). If an attribute is not associated with a type in the Object Model, the capitalized name of the attribute is used as a type name and declared as a Z basic type. The resulting schema is called a *class schema*.

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 declared in the corresponding Z schema of the instance class.

Consider for example a *Customer* class with attributes *name*, *ss_number*, *ph_number*, *address* and *date_of_birth*, as shown in Fig. 2.

The corresponding Z schema for the customer class, automatically generated by FuZE, is:

$$[Customer_ID, NAME, SS_NUMBER, PH_NUMBER, ADDRESS, DATE_OF_BIRTH]$$

Figure 2: Customer class structure diagram

In this formal model, the capitalized name of the attributes (for example, *NAME*, *SS_NUMBER*) are declared as basic Z type and used within Z schema as type names. The object class *Customer* is identified by the object identifier *ident*, which is of type *Customer_ID*.

2.2 Representing relationships in Z

In Fusion, a relationship is a tuple of objects, for example, $\langle \text{Customer}; \text{Account} \rangle$. In a binary relationship only two object classes are involved. In general, depending on the type of relationship, any number of object classes may be involved. A relationship is used to model the idea of association or correspondence between the objects belonging to classes. The relationship between classes expresses only the possibility of a relationship between the objects of the classes involved. The number of separate objects that participate in the relationship determines the arity of the relationship. In a ternary relationship, three separate objects relate to each other. Relationships involving more than three objects are referred to as n-ary relationships. Ternary relationships are difficult to implement and are usually avoided. There are work-arounds to convert the ternary relationships to binary relationship, which are easy to handle. As a part of the formalization tool, FuZE, we have a mathematical toolkit. The mathematical toolkit provides Z definitions of the common types of binary relationship (for example, one-to-one, one-to-many, many-to-many) that can exist between objects in a Fusion Object model. Each relationship is represented by a Z type (for example, *onetone*, *onetomany*), which is defined by parameterized Z schemas. Graphical symbols are used to make the relationships clear. Fig. 3 is an extract from the mathematical toolkit, which shows a parameterized Z schema, Z types and graphical symbols to represent the relationship.

A binary relationship between a *Customer* class and *Account* class is shown by a Fusion Object Model in Fig. 4.

A customer can maintain one or more accounts in a bank and an account is allotted to only one customer. This binary relationship between *Customer* and *Account* class is captured well and represented explicitly in the *Init* schema:

$[X, Y]$	
$1 \xleftrightarrow{\bullet} *$	$1 : \mathbb{P}(X \leftrightarrow Y)$
$1 \xleftrightarrow{\bullet} 1$	$1 : \mathbb{P}(X \rightsquigarrow Y)$
$* \xleftrightarrow{\bullet} 1$	$1 : \mathbb{P}(X \rightarrow Y)$
$\forall R : X \leftrightarrow Y \bullet (R \in 1 \xleftrightarrow{\bullet} * \Rightarrow$ $R^{-1} \in (Y \rightarrow X))$	

Figure 3: An extract of the Mathematical Toolkit

Figure 4: Customer class structure diagram

<i>Init</i>	
<i>customer</i> : $\mathbb{P} \text{ Customer}$	
<i>account</i> : $\mathbb{P} \text{ Account}$	
<i>has</i> : $\text{Customer} \leftrightarrow \text{Account}$	
$\forall i, j : \text{account} \bullet (i.\text{ident} = j.\text{ident} \Leftrightarrow i = j)$	
$\forall i, j : \text{customer} \bullet (i.\text{ident} = j.\text{ident} \Leftrightarrow i = j)$	
$has \in 1 \xleftrightarrow{\bullet} * [\text{customer}, \text{account}]$	

Other binary relationships, like one-to-one, one-to-many, many-to-many, can be similarly represented.

2.3 Representing aggregation structures

Aggregation is a mechanism for structuring the object model. It is used to represent the “part-whole” or “a-part-of” relationship. Aggregation is similar to relationship and can be formed by taking tuples of class instances. Aggregation is transitive and antisymmetric (An example of transitivity: if A is a part of B and B is a part of C , then A is part of C . An example of antisymmetry: if A is part of B then B is not part of A). In most cases, the existence of a component object depends on the existence of the aggregate object, of which it is a part. Fig. 5 shows an aggregate structure in Fusion, with Consortium as an aggregate class and Bank as a component class.

In Z, aggregate structures are represented by declaring the components as sets in a schema representing the aggregate object class. Any cardinality constraints are formally expressed in the predicate part. Our current formalization of aggregation makes clear the component nature of aggregates: a component instance

Figure 5: Object Model showing aggregation structure

in an aggregate cannot exist as a separate instance outside of the aggregate. Thus if a model shows two classes with the same name, one of which is in an aggregate and the other outside of the aggregate, then the classes are interpreted as two disjoint sets of instances.

A Z specification for the aggregation structure is shown below:

[*TYPE*, *SS_NUMBER*, *PH_NUMBER*, *NAME*, *DATE_OF_BIRTH*, *CREDIT_LIMIT*]
 [*BALANCE*, *ADDRESS*, *Account_ID*, *Bank_ID*, *Consortium_ID*, *Customer_ID*]

<i>Bank</i>	
<i>ident</i> : <i>Bank_ID</i>	
<i>Consortium</i>	
<i>ident</i> : <i>Consortium_ID</i>	
<i>bank</i> : $\mathbb{P} \text{Bank}$	
$\forall i, j : \text{bank} \bullet i.\text{ident} = j.\text{ident} \Leftrightarrow i = j$	
<i>Init</i>	
<i>customer</i> : $\mathbb{P} \text{Customer}$	
<i>consortium</i> : $\mathbb{P} \text{Consortium}$	
<i>account</i> : $\mathbb{P} \text{Account}$	
<i>holds</i> : <i>Bank</i> $\leftrightarrow$ <i>Customer</i>	
<i>has</i> : <i>Customer</i> $\leftrightarrow$ <i>Account</i>	
$\forall i, j : \text{account} \bullet (i.\text{ident} = j.\text{ident} \Leftrightarrow i = j)$	
$\forall i, j : \text{customer} \bullet (i.\text{ident} = j.\text{ident} \Leftrightarrow i = j)$	
$\forall i, j : \text{consortium} \bullet (i.\text{ident} = j.\text{ident} \Leftrightarrow i = j)$	
$\text{holds} \in 1 \longleftrightarrow 1[\{x : \text{Bank} \mid \exists y : \text{consortium} \bullet x \in y.\text{bank}\}, \text{customer}]$	
$\text{has} \in 1 \longleftrightarrow * [\text{customer}, \text{account}]$	

2.4 Representing generalization hierarchies

Generalization is a powerful tool for abstraction. It captures the similarities and differences among classes. It is used to represent the “is-a” relationship between classes. It provides a relationship between a class and

one or more refined versions of it. The refined class is called subclass and the class being refined is called superclass. A subclass inherits all the data and operations from the superclass. Fig. 6 shows a generalization relationship in which the *ATM* and *CashierStation* is generalized as an *EntryStation*.

Figure 6: An object model showing generalization hierarchy

The top class of a generalization/specialization hierarchy is formalized in the same way that other classes are defined in Z. Specializations of a class are formally defined by state schemas in which a variable of the general (parent) class type (representing the inherited part of the specialization) is declared in the declaration part, along with variables for the other attributes of the specialization. The identifier of a specialization is the inherited identifier of the general class. Specializations may or may not partition a general class, and may or may not be pairwise disjoint. Our current formalization supports the precise specification of such constraints.

For multiple specializations it may be the case that some of the superclasses have identically named attributes. It may then be necessary to rename the variables before including them in the subclass schema. Renaming is not necessary when the superclasses inherit the identically named attributes from a common ancestor (in this case, only single copies of the common attributes are included; this can be achieved in Z by hiding the other copies before inclusion). Renaming is necessary when the identically named attributes originate from different ancestors. A Z specification for the generalization structure is shown below:

[*TYPE*, ...]

<i>EntryStation</i> <i>ident</i> : <i>EntryStation_ID</i>
--

<i>CashierStation</i> <i>entrystation</i> : <i>EntryStation</i>
--

<i>ATM</i> <i>entrystation</i> : <i>EntryStation</i> <i>cash_on_hand</i> : <i>CASH_ON_HAND</i> <i>dispensed</i> : <i>DISPENSED</i>

<i>Init</i>	
<i>entrystation</i> : $\mathbb{P}$ <i>EntryStation</i>	
<i>customer</i> : $\mathbb{P}$ <i>Customer</i>	
<i>consortium</i> : $\mathbb{P}$ <i>Consortium</i>	
<i>cashierstation</i> : $\mathbb{P}$ <i>CashierStation</i>	
<i>atm</i> : $\mathbb{P}$ <i>ATM</i>	
<i>account</i> : $\mathbb{P}$ <i>Account</i>	
<i>owns</i> ₁ : <i>ATM</i> $\leftrightarrow$ <i>Consortium</i>	
<i>holds</i> : <i>Bank</i> $\leftrightarrow$ <i>Customer</i>	
<i>owns</i> ₂ : <i>CashierStation</i> $\leftrightarrow$ <i>Bank</i>	
<i>has</i> : <i>Customer</i> $\leftrightarrow$ <i>Account</i>	
$\forall sup : entrystation \bullet \neg ((\exists sub : atm \bullet sub.entrystation = sup) \Leftrightarrow (\exists sub : cashierstation \bullet sub.entrystation = sup))$	
$\forall sub1 : atm; sub2 : cashierstation \bullet sub1.entrystation \neq sub2.entrystation$	
$\forall sub : atm \bullet sub.entrystation \in entrystation$	
$\forall i, j : consortium \bullet (i.ident = j.ident \Leftrightarrow i = j)$	
$\forall i, j : atm \bullet (i.ident = j.ident \Leftrightarrow i = j)$	
$\forall sub : cashierstation \bullet sub.entrystation \in entrystation$	
$\forall i, j : cashierstation \bullet (i.ident = j.ident \Leftrightarrow i = j)$	
$\forall i, j : account \bullet (i.ident = j.ident \Leftrightarrow i = j)$	
$\forall i, j : customer \bullet (i.ident = j.ident \Leftrightarrow i = j)$	
$\forall i, j : entrystation \bullet (i.ident = j.ident \Leftrightarrow i = j)$	
$owns_1 \in 1 \longleftrightarrow 1[atm, consortium]$	
$owns_2 \in 1 \longleftrightarrow 1[cashierstation, \{x : Bank \mid \exists y : consortium \bullet x \in y.bank\}]$	
$holds \in 1 \longleftrightarrow 1[\{x : Bank \mid \exists y : consortium \bullet x \in y.bank\}, customer]$	
$has \in 1 \longleftrightarrow * [customer, account]$	

2.5 Putting it all together

Once the parts of the Object Model are defined they are collected in one schema that is a representation of the model. Invariants involving parts that were separately defined are expressed in the predicate part of this schema. Some restructuring of the schemas and/or redefinition of the parts may be needed to reduce the complexity of the specification produced by the rules.

A complete object model of an ATM example is shown in Fig. 7.

We do not provide the corresponding Z specification because of lack of space but it can be found in [3].

3 CASE tool support: an Overview of FuZE

We have implemented the rules previously described in a tool, called FuZE, built on top of the Paradigm Plus object-oriented modeling CASE tool [9]. Paradigm Plus supports a powerful script language which can be used to customize the behavior of the tool, methods checking, code generation, etc. The script language has complete access to the back end repository. We used the script constructs to query the object repository for necessary information.

The script language is BASIC-like and has the logical and arithmetic power of any higher programming languages. It has the semantics and control of a structured language and has access to the operating system shell. Its extensive function library, and efficient file input/output facilitates its use. We used the script language to build FuZE on top of Paradigm Plus. The script defines the mappings from Fusion Object Diagram to the Z language. The initial Object Diagram will be converted to a Z specification file (*.out*) by FuZE. ZTC is run on the Z specification to generate a printable type checked Formal Z specification (*.zed*) file. We also built a mathematical tool kit with all the necessary mathematical definitions and cardinality mappings used by ZTC. $\text{\LaTeX}$ is launched to generate a postscript version of the formal Z specification.

Figure 7: Fusion Object Model of an ATM

A .tex file is also generated by the script to enable L^AT_EX to generate the postscript version. An Object Interaction Graph showing the FuZE sequence of events and their inputs is shown in Fig. 8.

A small script that defines attribute types is shown below:

```

FOREACH Attribute
  IF (type$ = "") THEN
    upcaseAttribute$ = UPCASE$(Attribute.name)
    gosub CheckDuplicate
    IF (duplicateFlag <> 1) THEN
      basic$(basicCounter) = UPCASE$(Attribute.name)
      basicCounter = basicCounter+1
    END IF
  END IF
NEXT

```

The above simple construct declares an attribute type, if there are no duplicates and if no type is declared. It uses the attribute name in upper case to declare the type.

Information which are not available through script queries on object repository are acquired by querying the prefilled information arrays of FuZE. For example, the information regarding the parent class should be known while constructing a schema for the subclasses, such information is not available before handusing the sequential query access of the object repository. Prefilled arrays equipped FuZE with power to handle cardinality, aggregation upto four levels deeper and ternary associations, etc. More details about automatic translation results can be found in our FuZE technical report [2].

Figure 8: FuZE inputs and outputs

4 Towards a formalization of Fusion design: formalizing the OIG

To go further with our environment, we are extending FuZE to cover the design phase and to provide support for creating formal executable models of behavior. This will allow developers to not only analyze their designs, but also determine how compliant their designs are with respect to the formalized analysis models.

In this paper we present the extensions we are going to make to the tool to support the formalization of Fusion design models, specifically the Object Interaction Graph (OIG). Fig. 9 illustrates the links between the different Fusion models that we consider.

An OIG is a collection of boxes, which represent *design objects*, that are linked by arrows representing *message passing* (see Fig. 10).

4.1 Design objects

The main components of the OIG are design objects, which we will simply refer to as objects throughout this paper. Each object is an instance of a class that may have been defined in the analysis model (i.e., new classes can appear in the design).

In the following we illustrate the transformation rules by considering the following file system, composed of *Directory* and *File* objects (we only give here the generated Z specification):

[*Type*, *DirectoryID*, *FileID*]

Figure 9: From the Analysis phase to the Design phase

Figure 10: The Object Interaction Graph Notation

<i>Directory</i>		<i>File</i>	
<i>ident</i> : <i>DirectoryID</i>	[object identifier]	<i>ident</i> : <i>FileID</i>	[object identifier]
<i>attributes</i> : <i>Type</i>	[attributes of the class]	<i>attributes</i> : <i>Type</i>	[attributes of the class]
...	[Invariants of the class]	...	[Invariants of the class]
<i>System</i>			
<i>directory</i> : $\mathbb{P}$ <i>Directory</i>			[instances of directories]
<i>file</i> : $\mathbb{P}$ <i>File</i>			[instances of files]
...			
...			[Invariants of the system]

The two first transformation rules concern the basic elements of the OIG:

- The design object

TRANSFORMATION 1 Objects of a Class

A design object is represented in Z as a binding of the state schema corresponding to its class:

- The collection of objects

TRANSFORMATION 2 Collection of objects of a Class

A collection of objects of the same class is represented in Z as a set of bindings of the state schema corresponding to its class:

4.2 Message passing

The only connection between objects in the OIG is the labeled arrow representing *message passing* from a *sender* (or client) to a *receiver* (or server). The message represents the invocation of a method in the receiver interface. A description of the method is given (informally in Fusion) in the data dictionary.

TRANSFORMATION 3 Message passing

Description:

operation Directory: secure(fn: Filename)
*//description in natural language
 of the method*

A message passing (labeled arrow) is represented in Z as an operation schema. The label of the arrow is used as a name for the function, and the parameters are translated as inputs of the operation schema^a. The described operation modify the receiver object.

^aNote that this transformation needs human intervention, as we have not yet found a way to translate a description in natural language into Z!

If we consider the specification described in section 4.1, we get the following Z operation schema:

$\frac{\text{secure}}{\Delta \text{System}}$	
$fn? : \text{File}$	
$d : \text{Directory}$	
$d \in \text{directory} \wedge fn? \in \text{file}$	[preconditions]
$\dots$	[Formal description of the operation]

We treat bidirectional dataflows and multiple arrows in the subsection 4.4, concerning the sequencing.

TRANSFORMATION 4 Link between sender and receiver

The message passing between a “sender” and a “receiver” is formalized by two operations linked together^a.

$$Op_1 == \exists \Delta B \bullet \text{message} \wedge \text{Promote_message}$$

^aWe use here the *promotion* technique, which is used in Z when a global operation is defined in terms of a local operation upon an indexed component [12, p. 193].

The aim of this paper is not to detail the Z part of the formalization, but we chose this representation of message passing, in a first attempt, to separate the Z operations in the same way they are split in the OIG, while keeping the formal link between Z schemas.

4.3 Message passing to collections

For messages passed to collections, the description of the operation follows rule 3 described in the previous subsection for message passing to one object. The difference is in the way the operation affects the system.

TRANSFORMATION 5 Message passing to collections

4.4 Message sequencing

The *message sequencing* involves labeling the messages with Dewey Decimal numbers (for example 1, 1.1, 1.1.1, 1.2, 2) that represent the order in which the messages are sent. In the Fusion OIG, the sequencing is not mandatory, it is a simple annotation. This leads sometimes to ambiguities, as arrows with no sequences take place at any time, i.e., the messages can be generated at any time. For example in Fig. 11, does *rep* occur after *req1* or after *req3*? There are situations where this information may be needed (especially at a high-level design, where one could delay making a decision as to when certain operations can take place) so we do not make the sequencing mandatory. The sequencing labels allow us to define formally in which order the messages are sent. If there is no label, no explicit sequencing is generated in the formal part. For a complete example of message sequencing formalization, see section 5.

Figure 11: An ambiguous message sequencing

TRANSFORMATION 6 Sequencing of operation

The OIG sequencing is used to generate the sequencing of Z operation, defined using the corresponding schema operator ($\circ$). The “leaves” operations are first defined formally, then composed and linked with the “root” operations.

$$Op == Op_1 \circ Op_2$$

4.5 Dynamic object creation

TRANSFORMATION 7 Dynamic object creation

A new object is represented in Z as an output binding of the state schema corresponding to its class:

$\begin{array}{l} \text{create} \\ \hline d? : Data \\ i! : Item \\ \hline i!.data = d? \end{array}$
--

5 Strength of the formalization

We illustrate in this section, how the formal model of the OIG is useful, by reconsidering the role of the OIG in the Fusion development lifecycle. The definition of the OIG aims to give a first “implementation”, at a

high level of abstraction, of the requirements captured at the analysis phase.

Note that the two models are complementary, and that, unlike our transformation rules defined for the analysis level, the Z specification does not capture all the information of the Fusion OIG, although it describe formally the operations involved (for example the controller of an operation is not described in the formal model).

We give in the following a complete example of the formalization of an OIG taken from [4]. In [1], we have illustrated the formalization of the analysis model of a gas delivery station. The *enable_pump* operation, defined at the analysis model is detailed in the design phase by an OIG.

We will demonstrate that the Z *enable_pump₂* operation, derived from the OIG (Fig. 13), is a refinement of the *enable_pump* operation defined at the analysis model (and derived from the Fusion operation model of the operation illustrated in Fig. 12).

We give here an extract of a more complete specification [1]. The analysis models (Object Diagram and Operation Model) of a *pump* and an operation *enable_pump* are shown in Fig. 12.

Operation: *enable_pump*

Description: Enables the pump ready to dispense petrol.

Reads: supplied *n*

Changes: *Pump* with *pump.pump_id = n*, *motor*, *clutch*, *display*

Sends:

Assumes:

Result: if *pump* is enabled or out of service then no effect
otherwise:

Status of *pump* is enabled.
Its *display* has been initialized.
Its *motor* is running, and its *clutch* is free.

Figure 12: The *enable_pump* Fusion Analysis models

Without giving all the definitions corresponding to the basic types, here is the corresponding Z specification:

 Figure 13: The *enable_pump* Fusion OIG

In the following, we give the Z model of the OIG given in Fig. 13.

As mentioned in TRANSFORMATION 6, we first describe the “leaf” operation *Op*₁₁ (labeled 1.1 in Fig. 13). We describe the “receiver” part of the operation (see TRANSFORMATION 4):

Note that we don't detail the specification of the class *Display*. We'll do the same for other classes, when the formal definition is not relevant.

We then describe the “sender” part of the operation:

<i>Promote_reset</i>	
$\Delta Pump$	
$\Delta Display$	[object variable]
$\theta Display = display$	[formal link between the variable and <i>display</i>]
$display' = \theta Display'$	

We can then define the first step of *enable*, which is the operation *Op_11*:

$$Op_{11} == \exists \Delta Display \bullet reset \wedge Promote_reset$$

We do the same for the *Op_12* and *Op_13*:

<i>free</i>	<i>Promote_free</i>
$\Delta Clutch$	$\Delta Pump$
$engaged' = True$	$\Delta Clutch$
	$\theta Clutch = clutchmotorassembly.clutch$
	$clutchmotorassembly'.clutch = \theta Clutch'$

$$Op_{12} == \exists \Delta Clutch \bullet free \wedge Promote_free$$

<i>start</i>	<i>Promote_start</i>
$\Delta Motor$	$\Delta Pump$
$running' = True$	$\Delta Motor$
	$\theta Motor = clutchmotorassembly.motor$
	$clutchmotorassembly'.motor = \theta Motor'$

$$Op_{13} == \exists \Delta Motor \bullet start \wedge Promote_start$$

Now we can describe the sequence of the 3 operations, defining a new operation *enable*¹:

$$enable == Op_{11} \circ Op_{12} \circ Op_{13}$$

The operation *enable* is now seen as a “local” operation. So we define the promotion for it:

<i>Promote_enable</i>
$\Delta System$
$\Delta Pump$
$input? : Terminal$
$input? \in terminal$
$\theta Pump = enables(input?)$
$enables' = enables \oplus \{input? \mapsto \theta Pump'\}$

And then we define the corresponding operation *Op_1*:

$$Op_1 == \exists \Delta Pump \bullet enable \wedge Promote_enable$$

¹ If there are no labels, this part of the transformation is skipped.

The *op_1* operation itself is seen as a “local” operation, related to *enable_pump*. So we define first its promotion schema:

$ \begin{array}{l} \textit{Promote_Op1} \text{ ---} \\ \Delta \textit{System} \\ \Xi \textit{Terminal} \\ n? : \textit{Pump} \\ \hline n? \in \textit{pump} \\ \theta \textit{Terminal} = (\textit{enables}^{-1} n?) \\ \textit{enables}' = \textit{enables} \oplus \{\theta \textit{Terminal}' \mapsto n?\} \end{array} $
--

Finally, we can formally define the operation *enable_pump₂*(*n*) as:

$$\textit{enable_pump}_2 == \exists \Xi \textit{Terminal} \bullet \textit{Op_1} \wedge \textit{Promote_Op1}$$

In order to prove that *enable_pump₂* is a correct refinement of what we had at the analysis level (*enable_pump*), we have to establish that both the following statements are theorems:

$$\begin{array}{l} \forall \textit{System}; \quad \textit{id}? : \textit{PUMP_ID}; \textit{input}? : \textit{Terminal}; n? : \textit{Pump} \bullet \\ \text{pre } \textit{enable_pump} \Rightarrow \text{pre } \textit{enable_pump}_2 \end{array} \quad (1)$$

$$\begin{array}{l} \forall \Delta \textit{System}; \quad \textit{id}? : \textit{PUMP_ID}; \textit{input}? : \textit{Terminal}; n? : \textit{Pump} \bullet \\ \text{pre } \textit{enable_pump} \wedge \textit{enable_pump}_2 \Rightarrow (\exists \textit{System}' \bullet \textit{enable_pump}) \end{array} \quad (2)$$

Eq. 1 expresses that we *weaken* the preconditions of the initial operation, by taking into account more possibilities. Eq. 2 expresses that we *strengthen* the operation.

The proof, though not given in this paper, is not difficult. The help of a tool such as ZANS [7] to calculate the preconditions and to expand the schema definitions is valuable.

6 Conclusion and future works

In our experiments with FuZE, 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 in the semantics of Fusion, gaps in our knowledge of the system being modeled, and inconsistent requirements. The feedback from the translation activity can be used to improve the Fusion models.

The primary objectives of integrating Fusion and Z is to create analyzable Fusion models and facilitate generating Z specifications. The integration allows one to use Z type checkers (e.g., ZTC and *fuzz*) and Z animation tools, such as ZANS, to analyze models. In our experiments, we used ZTC, *fuzz* and ZANS and found that doing such analyses can reveal additional problems with the Fusion models (as well as with the formalization of the problem).

Presently the scripts are written to handle Fusion analysis models. Since Paradigm Plus has meta-information about the types of methods supported, the scripts could be generalized to handle all the OOMs supported by Paradigm Plus (e.g., Fusion, Rumbaugh, Coad Yourdon, Shlaer Mellor). Depending on the type of OOM used, the script can be written to handle the variation in notation between methods.

The improved transformation rules and expansion of FuZE support to include the design phase will enable consistency checks and incremental development using an integrated formal/informal development environment, moving the state of the art towards providing the tool supports industry requires to take advantage of the rigorous analysis capabilities available when using formal methods for formalizing object-oriented models. In the future we plan to exercise FuZE with larger, more complex models.

References

- [1] Brian W. Bates, Jean-Michel Bruel, Robert B. France, and Maria M. Larrondo-Petrie. Formalizing Fusion Object-Oriented Analysis Models. In Elie Najm and Jean-Bernard Stephani, editors, *Proceedings of the First IFIP International Workshop on Formal Methods for Open Object-based Distributed Systems*, Paris, France. Chapman & Hall, London, UK, 4–6 March 1996.
- [2] Jean-Michel Bruel, Bharat Chintapally, Robert B. France, and Gopal K. Raghavan. FuZE–Draft of the User’s Guide. FAU Technical Report TR-CSE-96-9, Department of Computer Science & Engineering, Florida Atlantic University, Boca Raton, FL-33431, USA, February 1996.
- [3] Jean-Michel Bruel, Robert B. France, Maria M. Larrondo-Petrie, Bharat Chintapally, and Gopal K. Raghavan. CASE-based Rigorous Object-Oriented Modeling. FAU Technical Report TR-CSE-96, Department of Computer Science & Engineering, Florida Atlantic University, Boca Raton, FL-33431, USA, August 1996.
- [4] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, Englewood Cliffs, NJ, Object-Oriented Series edition, 1994.
- [5] Roger Duke, Paul King, Gordon A. Rose, and Graeme Smith. The Object-Z specification language. In Timothy D. Korson, Vijay K. Vaishnavi, and Bertrand Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 5*, pages 465–483. Prentice Hall, 1991.
- [6] Robert B. France, Jean-Michel Bruel, and Maria M. Larrondo-Petrie. An Integrated Object-Oriented and Formal Modeling Environment. *To appear in the Journal of Object-Oriented Programming (JOOP)*, 1996.
- [7] Xiaoping Jia. *An Approach to Animating Z Specifications*. Division of Software Engineering, School of Computer Science, Telecommunication, and Information Systems, DePaul University, Chicago, IL, USA, 1995.
- [8] D. Randolph Johnson and Haim Kilov. Can a flat notation be used to specify an OO system: using Z to describe RM-ODP constructs. In Elie Najm and Jean-Bernard Stephani, editors, *Proceedings of the First IFIP International Workshop on Formal Methods for Open Object-based Distributed Systems*, Paris, France, pages 407–418. Chapman & Hall, London, UK, 4–6 March 1996.
- [9] Protosoft, Inc. *Paradigm Plus – Release 2.0*. Protosoft Technical Support, Houston, Texas, USA, January 1994.
- [10] J. Michael Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, Englewood Cliffs, NJ, Second edition, 1992.
- [11] Paul A. Swatman and Paula M. C. Swatman. FOOM - A Method for Acquiring Appropriate Inter-Organisational Systems. In *Proceedings of the 8th International EDI-IOS Conference*, Bled, Slovenia, 5–8 June 1995.
- [12] Jim C. D. Woodcock and Jim Davies, editors. *Using Z – Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science, Englewood Cliffs, NJ, 1996.