Rigorous Object-Oriented Modeling: Integrating Formal and Informal Notations

R. B. France[1*], J.-M. Bruel[2], M. M. Larrondo-Petrie[1], E. Grant[1]

[1] Department of Computer Science & Engineering
Florida Atlantic University
Boca Raton, FL-33431, USA.
[2] Laboratoire IRIT/SIERA, bat. 1R1
118, rte de Narbonne
31062 Toulouse Cedex, France

Abstract. The high-quality modeling experiences embedded in the more mature graphical OO methods (OOMs) makes their application to complex systems attractive, but the lack of firm semantic bases for the modeling notations can significantly hamper the development of such systems. One approach to making OOMs more precise and amenable to rigorous analysis is to integrate them with suitable formal modeling techniques. In this paper we describe a technique for integrating an OOM, the Fusion method, and a formal specification notation, Z.

1 Introduction

The more mature informal object-oriented (OO) modeling techniques (e.g., Fusion [7]) provide good support for developing concise, highly-structured, models of behavior from a variety of perspectives. These techniques are based on some of the best modeling experiences available, and consist of rich sets of structuring and abstraction mechanisms. A deterrent to the use of OO methods (OOMs) for the development of complex systems is their lack of support for rigorous analysis. This is a result of the loosely-defined semantics for the modeling notations.

An approach to making informal OO models more precise and amenable to rigorous analysis is to integrate them with suitable formal notations. Several studies in this area have been published (e.g., see [1, 10, 11]). In our work, "integration" means providing a bridge from the informal OO modeling concepts to the formal notation. Our work differs from others in that our focus is on producing formalizations that can directly support verification and validation activities, and for which mechanical support is possible. For this reason, we have used only formal notations for which there exist sound sets of analysis tools. The Z notation [4, 16] we use is supported by typecheckers (e.g., ZTC [14]), animators (e.g., ZANS [13]), and theorem proving environments (e.g., Z/EVES [8]). We assume that the reader is familiar with the Z notation. For a detailed description of the Z notation see [16].

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

Our past work focused on the formalization of the analysis models of the Fusion method [7] and it resulted in a set of rules for transforming Fusion Object Models to Z specifications (e.g., see [1]). We built a prototype tool, FuZE [5], that automatically generates Z specifications from Fusion Object Models. Z analysis tools (ZTC and ZANS) can be called from within the tool to analyze the generated Z specifications. The tool has been applied by graduate students at Florida Atlantic University on non-trivial projects and case studies (e.g., see [9]). In general, our experiences indicate that formalization and analysis of informal models can uncover problems with the informal models, and lead to a deeper understanding of the problem. Our applications of the integrated methods on case studies also uncovered limitations of our previous rules. In this paper we present a technique for transforming Fusion Object Models (OMs) to Z specifications that improves upon our previous technique. The new technique supports the automated transformation of a wider range of OMs.

We have also extended our transformation technique to the design models of Fusion. We present some of our current work on formalizing Fusion design models in this paper. In section 2 we give an overview of the Fusion analysis and design modeling techniques. In section 3 we outline our technique for formalizing Fusion's analysis models, and in section 4 we present the results of our recent work on formalizing Fusion's design models. We conclude in section 5 with a summary of our work and an outline of ongoing work.

2 The Fusion Modeling Techniques

Fusion is an object-oriented software development methodology that combines and extends existing techniques, e.g., Rumbaugh's Object Modeling Technique (OMT) [15], Booch's technique [2], Wirfs-Brock's Class Responsibility Collaborator [17] (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 analysis, design, and implementation.

2.1 Fusion Analysis Models

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

Object Model. An *Object Model* (OM) defines the static structure of the information manipulated by the application in terms of classes and the relationships among them. An example of an OM for a petrol dispensing system is given in Fig. 1 (taken from [7]). In an OM a class is represented by a box that consists of a partition containing the name of the class, and another containing a list of attributes for the class. Type information for attributes may or may not be shown in an OM. Relationships are depicted as diamond adorned lines between classes. In the diamond the name of the relationship and any attributes associated with it are given. In Fusion, relationships represent ***optional*** links between

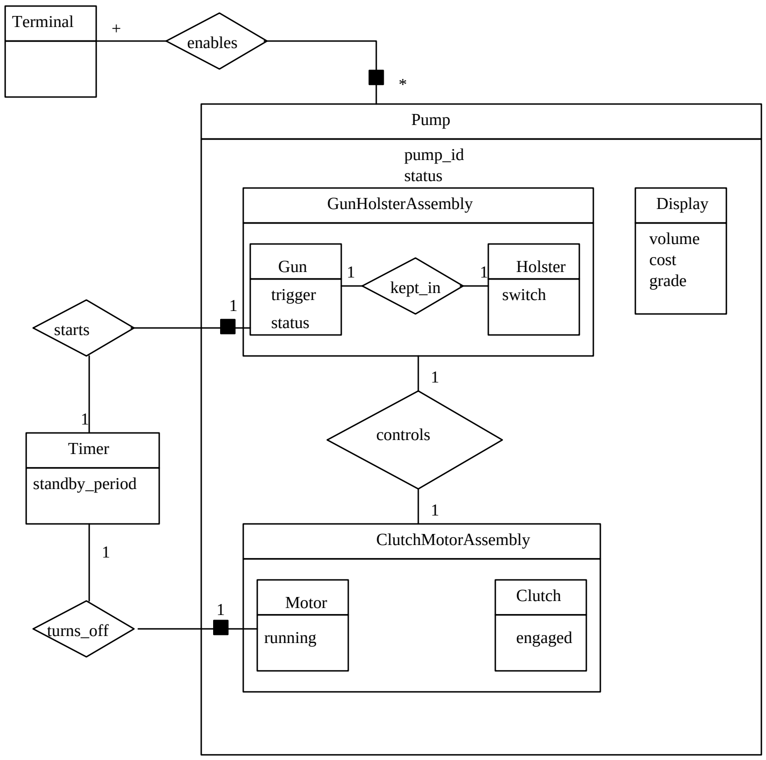

Fig. 1. Object Model for *Pump*

objects (class instances). Relationships are associated with cardinalities that restrict the number of class instances (objects) that can be linked to each other. A cardinality can be a single number, an integer range, '*' (0 or more), or '+' (1 or more). A black box attached to a relationship indicates that all existing objects of the adjacent class must be linked (i.e., it indicates mandatory links). For example, the relationship ***enables*** indicates that every existing ***Pump*** object must be linked to 1 or more ***Terminal*** objects, and an existing ***Terminal*** object may or may not be associated with (0 or more) ***Pump*** objects. In Fusion, classes that contain a class structure are called ***aggregates***. The classes in an aggregate are called ***components***. For example, ***Pump*** is an aggregate with components ***GunHolsterAssembly***, ***Display*** and ***ClutchMotorAssembly***. Fusion also provides constructs for building generalization/specialization structures. Additional constraints on class structures can be expressed as annotations in OMs.

Interface Model. This model defines the externally observable behavior of the application. It consists of two models: the Operation Model and the Lifecycle Model. An ***Operation Model*** characterizes the observable effects of system operations. A system operation is one that can be invoked by users of the system. Each system operation is described by a Fusion ***Operation Schema*** (OS). A ***Life-Cycle Model*** characterizes the allowable sequences of system operation invocations for the application.

The Data Dictionary. The ***Data Dictionary*** provides information about the classes and operations not included in the previous models (e.g., attribute type information).

2.2 Fusion Design Models

In the Fusion design phase a system's behavior is modeled in terms of interactions between the objects of the system. The primary model of functional behavior is the ***Object Interaction Graph*** (OIG). An OIG is developed for each system operation described in the Operation Model, and it describes the object interactions that accomplish the operation.

Graphically, an OIG is a collection of boxes, each representing an object of the system to be implemented, connected by labeled arrows representing message passing between the objects (e.g., see Fig. 2). The order in which interactions can take place is constrained by sequence labels associated with the interactions. Interactions shown in an OIG do not indicate that they must take place; they indicate only the possibility of such an interaction taking place.

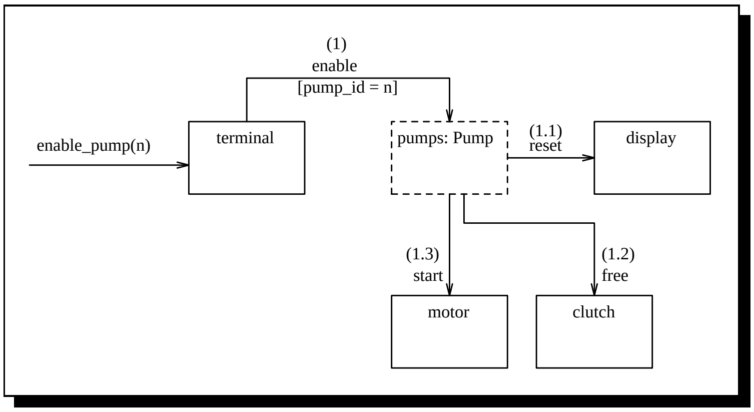

Fig. 2. The *enable_pump* Fusion OIG

An individual object is represented as a solid line box, and a collection of objects is represented as a broken line box. One object in an OIG is identified as the ***controller*** and all other objects are referred to as ***collaborators***. The controller object is distinguished by being the only object with an arrow entering it that does not originate from another object (or collection of objects) in the OIG. A complete OIG model includes a descriptive text stating the conditions under which interactions take place.

An example of an OIG is shown in Fig. 2. The controller in the OIG is the object ***terminal***. The message ***enable*** is sent to all objects in the collection ***pumps*** with $\boldsymbol{pump\_id} = n$. Each selected object in ***pumps*** can then send messages ***reset***, ***free***, and ***start*** (in that order, as indicated by the interaction sequence labels).

The other models of the Fusion design phase are:

- a ***Visibility Graph*** that models access restrictions and lifetime bindings of objects,
- an ***Inheritance Graph*** that defines the inheritance structure of the solution, and
- a ***Class Description*** (CD) for each class in the system to be implemented.

The CDs provide a textual organization of the information found in the other Fusion Design Models.

3 Formalizing Fusion Analysis Models

In this section we outline a technique we developed for formalizing the Object and Operation Models of the Fusion Method. The Lifecycle Model is expressed formally in Fusion so there is no need to formalize it. We use models from the petrol system case study given in [7] to illustrate the formalization technique. The OM for the part of the system we will use in our illustration is shown in Fig. 1.

In our formalization, an OM is a characterization of the valid states of a system, and the Operation Model specifies operations in terms of their observable effects on valid states. Here, a valid state of a system is a linked set of objects that exhibit the properties expressed in the OM (i.e., the structure is consistent with cardinality and other constraints, including those expressed as annotations in the OM). A valid state is called a ***configuration*** in this paper.

3.1 Formalizing the Object Model

The formalization of an OM results in a Z schema that characterizes the configurations of the system modeled by the OM. A Z schema consists of two parts: a declaration part in which variables are declared, and a predicate part in which a predicate constraining the declared variables is given. A Z schema characterization of an OM consists of variables representing objects and links among objects in the declaration part, and constraints on the objects and links in the predicate part.

The formalization of an OM can be completely automated if the OM annotations are expressed formally in predicate logic. In practice, annotations expressed in predicate logic notation can be significantly longer than concise natural language annotations. A related problem is the sometimes lengthy Z specifications automatically produced by the OM-to-Z transformation process. We found that properties that are concisely expressed in natural language (e.g., "object identifiers are unique" and "subclasses are disjoint") often required lengthy formal statements. To alleviate this problem we developed a set of high-level Z definitional constructs, called ***Z macros***, for some commonly occurring properties. Z macros can be automatically translated ('compiled') to Z specification elements. Whenever possible, the modeler can use the macros to express annotations. Properties not covered by the macros have to be expressed in predicate logic notation before complete Z specifications can be produced from OMs.

In transforming OMs to Z specifications, Z macros are used wherever possible in the result output to the modeler. These macros are transformed to 'raw' Z notation before they are analyzed by the Z analysis tools. Examples of macros are given throughout this section.

Below we outline the steps for formalizing a Fusion OM.

Step 1: Formalize Basic Classes. A basic class is a class that is not an aggregate or a subclass in a specialization hierarchy. For each basic class a Z schema called a ***class schema*** is produced. A class schema consists of a variable representing the identifier of a class object, and variables representing object attributes. Attribute and object identifier types are specified as Z ***basic types*** or as pre-defined Z types (e.g., $\mathbb{N}$, the natural number type). In Z a type is a set and a basic type is a set of primitive elements. The basic type declarations and the class schema for a basic class have the following forms:

$$[CLASSNAME, ATTR1\_TYPE, \ldots]$$

$$\begin{array}{|l}
\multicolumn{1}{l}{ClassName} \\
\hline
ident : CLASSNAME \\
attr1 : ATTR1_TYPE; \ \ldots \\
\hline
[Constraints\ on\ attribute\ values] \\
\hline
\end{array}$$

In the above $\boldsymbol{CLASSNAME}$, $\boldsymbol{ATTR1\_TYPE}$; ... are Z basic types. $\boldsymbol{CLASSNAME}$ is the set from which identifiers of class objects are drawn and $\boldsymbol{ATTR1\_TYPE}$, ... are attribute types. The predicate in the predicate part of the schema constrains attribute values. If the constraints are expressed in natural language in the OM then a human must provide their formal expressions. Note that the formalization forces a type to be associated with each attribute; if one is not given in the OM the FuZE tool uses the capitalized attribute name as the type of the attribute (the type is introduced as a basic type).

To illustrate the formalization of basic classes we formalize the ***Holster***, ***Gun***, ***Clutch***, and ***Motor*** classes of the petrol system (see Fig. 1). The first step is to determine the attribute types. Type information not shown in the OM should be in the data dictionary. The data dictionary supporting the petrol system model (not given here for the sake of brevity), indicates that the attribute ***status*** of ***Gun*** can be one of the following values: ***Enabled***, ***Out_of_Service***, ***Disabled***. It also indicates that the attributes ***running*** (in ***Motor***), ***engaged*** (in ***Clutch***), and ***switch*** (in ***Holster***), can have one of two values: ***on***, ***off***. The basic types and class schemata for the classes are given below:

$[GUNID, HOLSTERID]$

$[MOTORID, CLUTCHID]$

$[TRIGGER]$

$STATUS ::= Enabled \mid Out\_of\_Service \mid Disabled$

$SETTING ::= on \mid off$

Holster
$ident : HOLSTERID$
$switch : SETTING$

Gun
$ident : GUNID$
$trigger : TRIGGER$
$status : STATUS$

Motor
$ident : MOTORID$
$running : SETTING$

Clutch
$ident : CLUTCHID$
$engaged : SETTING$

Step 2: Formalize Aggregate Classes and Subclasses. In this step, aggregates and subclasses are formalized. The class structure in an aggregate must be formalized before the aggregate can be completely formalized. The class structure is essentially an OM, and thus it can be (recursively) formalized using the OM formalization process we are outlining here. The class schema for an aggregate class has the following form:

$[AGGNAME, \ldots]$

AggName
$InternalStruct$
$ident : AGGNAME$
$attr1 : \mathbb{P}\, ATTR1\_TYPE;\ \ldots$

$[Constraints\ on\ attributes]$

InternalStruct defines the internal structure of the aggregate, that is, it defines the aggregate's components and relationships between the components. In the predicate part of ***AggName*** constraints on the attribute values are given.

To illustrate the formalization of aggregates we give the class schemata for ***GunHolsterAssembly*** and ***Pump***. The ***GunHolsterAssembly*** aggregate is formalized below (the schema characterizing internal structure is on the left):

GunHolsterInternal
$gun : Gun$
$holster : Holster$
$kept\_in : Gun \leftrightarrow Holster$

$kept\_in \in 1 \leftarrow\!\blacksquare\!\!\longrightarrow\!\blacksquare\!\rightarrow 1[\{gun\}, \{holster\}]$

$[GHAID]$

GunHolsterAssembly
$GunHolsterInternal$
$ident : GHAID$

The macro $1 \leftarrow\!\blacksquare\!\!\longrightarrow\!\blacksquare\!\rightarrow 1$ specifies a mandatory 1-to-1 relationship (Fusion requires that component instances be linked via enclosed relationships).

The class schema for the ***Pump*** aggregate is given below:

$$
\begin{array}{|l}
\hline
\textit{PumpInternal} \\
\textit{display} : \textit{Display} \\
\textit{gha} : \textit{GunHolsterAssembly} \\
\textit{cma} : \textit{ClutchMotorAssembly} \\
\textit{controls} : \textit{GunHolsterAssembly} \\
\quad \leftrightarrow \textit{ClutchMotorAssembly} \\
\hline
\textit{controls} \in 1 \longleftrightarrow 1[\{\textit{gha}\}, \{\textit{cma}\}] \\
\hline
\end{array}
$$

$$[\textit{PUMPID}, \textit{PUMP\_ID}]$$

$$
\begin{array}{|l}
\hline
\textit{Pump} \\
\textit{PumpInternal} \\
\textit{ident} : \textit{PUMPID} \\
\textit{pump\_id} : \textit{PUMP\_ID} \\
\textit{status} : \textit{STATUS} \\
\hline
\end{array}
$$

Formalization of subclasses is done by including superclass class schemata in subclass class schemata.

$$
\begin{array}{|l}
\hline
\textit{SubClassName} \\
\textit{SuperClass} \\
\textit{attr}1 : \textit{ATTR}1\_\textit{TYPE};\ \ldots \\
\hline
[\textit{Constraints on attributes}] \\
\hline
\end{array}
$$

3. Specify Constraints on Specialization Hierarchies. In a specialization hierarchy subclasses can be disjoint (i.e., an instance cannot belong to more than one subclass), or not disjoint (i.e., multiple inheritance is possible). Also, the instances of a superclass may all be instances of depicted subclasses (in which case the superclass is said to be ***abstract***), or there may be superclass instances that are not instances of any depicted subclass. Z macros are used to express these properties (not given here for lack of space).

Properties of specialization hierarchies are specified in a schema, called the ***configuration schema*** of the hierarchy. Such a schema has the following form:

$$
\begin{array}{|l}
\hline
\textit{SpecConfig} \\
\textit{supers} : \mathbb{P}\ \textit{SuperClass} \\
\textit{subs\_1} : \mathbb{P}\ \textit{SubClass\_1};\ \ldots;\ \textit{subs\_n} : \mathbb{P}\ \textit{SubClass\_n} \\
\hline
[\textit{One or more of the following Z macros included here}:] \\
(\textit{subs\_1}, \ldots, \textit{subs\_n})\ \textit{partition supers} \\
\textit{disjoint}(\textit{subs\_i}, \ldots, \textit{subs\_j}) \\
\textit{abstract}(\textit{supers}, \textit{subs\_1}, \ldots, \textit{subs\_n}) \\
\hline
\end{array}
$$

4. Specify relationships and other constraints among classes. This last step results in the OM's ***configuration schema***, that is, a characterization of the OM's configurations. Configuration schemata have the following form:

$$
\begin{array}{|l}
\hline
\textit{OMName} \\
\textit{Spec1Config};\ \ldots [\textit{specialization config schemata}] \\
\textit{classobjs\_1} : \mathbb{P}\ \textit{Class\_1};\ \ldots [\textit{sets of class objects}] \\
\textit{rel\_ij} : \textit{Class\_i} \leftrightarrow \textit{Class\_j};\ \ldots [\textit{relationships}] \\
\hline
[\textit{Cardinality and other constraints}] \\
\hline
\end{array}
$$

The configuration schema for an OM expresses constraints on relationships, and restrictions on the sharing of components across aggregates. For example, the presence of a relationship in an aggregate is a reference to a larger relationship, that is, the relationship in the aggregate is a subrelation of a larger relation. This property is stated in the configuration schema for all relationships enclosed in an aggregate. We have defined the following Z macros for commonly occurring properties:

- ***Relationship cardinalities:*** We have defined macros that represent Z formalizations of cardinality constraints. These macros are of the form:

  $$p \longleftrightarrow q[domain, range]$$

  (if the relationship is mandatory black boxes are placed on the appropriate end(s)), where p and q are the cardinalities, $domain$ is the domain of the relationship and $range$ is the range of the relationship.
- ***Object identifier uniqueness:*** The property that object identifiers in a configuration are unique is captured by the following macro:

  $$unique\_ids[\{s1, s2, \ldots\}] \mathrel{\widehat{=}} \forall\, i, j : s1 \bullet i.ident = j.ident \Leftrightarrow i = j \land \\ \forall\, i, j : s2 \bullet i.ident = j.ident \Leftrightarrow i = j \land \ldots$$

  In the above macro, $s1, s2, \ldots,$ are sets of objects.
- ***Constraints pertaining to sharing of aggregate components:*** Modelers may want to restrict the sharing of components across aggregate instances. For example, the following Z macro is used to specify that components cannot be shared across aggregates (in the following, s is the set of instances of the aggregate Agg in a configuration):
  - $physical\_aggregate(s)$: This macro expands to a predicate that prohibits sharing of components of Agg. For example, if $aggs$ is a set of objects of the aggregate Agg that has a single component class $comp$ characterized by the schema $COMP$ (i.e., $comp : COMP$), then $physical\_aggregate(aggs)$ expands to the following predicate:

    $$\forall\, a1, a2 : aggs \mid a1.comp = a2.comp \bullet a1 = a2$$

    The restriction as defined in the above is not recursive, for example, if $comp$ is also an aggregate, then the above macro does not restrict it to be a physical aggregate.

For the pump system, the following restrictions are stated as OM annotations (not shown in model given in Fig. 1):

- The $GunHolsterAssembly$ aggregate is a physical aggregate, that is, its components are not shared. The $ClutchMotorAssembly$ and the $Pump$ aggregates are also physical aggregates.
- All instances of component classes in a configuration must be a component of an aggregate in the configuration. For example, all instances of Gun in a configuration must be a component of one $GunHoslterAssembly$ aggregate instance in the configuration.

The configuration schema for the pump system is given below:

PumpSystem

$terms : \mathbb{P}\ Terminal$ [*terminal instances in configuration*]
$pumps : \mathbb{P}\ Pump$ [*pump instances in configuration*]
$timers : \mathbb{P}\ Timer$ [*timer instances in configuration*]
$enables : Terminal \leftrightarrow Pump$
$keptin : Gun \leftrightarrow Holster$
$controls : GunHoslterAssembly \leftrightarrow ClutchMotorAssembly$
$starts : Timer \leftrightarrow Gun$
$turnsoff : Timer \leftrightarrow Motor$

$enables \in + \longleftrightarrow * [terms, pumps]$

$starts \in 1 \longleftrightarrow 1[timers, \{p : pumps \bullet (p.gha).gun\}]$

$turnsoff \in 1 \longleftrightarrow 1[timers, \{p : pumps \bullet (p.cma).motor\}]$
$keptin = \bigcup\{p : pumps \bullet (p.gha).kept\_in\}$
[*linked components form a subrelation keptin*]
$\bigcup\{p : pumps \bullet p.controls\} = controls$
[*linked components form a subrelation controls*]
$physical\_aggregate(pumps)$
$physical\_aggregate(\{p : pumps \bullet p.gha\})$
$physical\_aggregate(\{p : pumps \bullet p.cma\})$
$unique\_ids(\{terms, timers, pumps\})$
... [*unique_ids macros for the components of pumps*]

3.2 Formalizing Operation Models

In this section we formalize one of the system operations for the petrol system. The formalization of the Fusion Operation Model is only partially mechanizable because the operation descriptions are written mostly in natural language. Currently our tool, given a Fusion operation schema, generates only the declaration part of the corresponding Z operation schema. The formal relationship between the before and after states of the operation must be provided by a human developer. This relationship is stated in the predicate part of the Z operation schema.

The Fusion description of the operation ***enable_pump*** and its corresponding formalization in Z is given below:

Operation: *enable_pump*
Description: Enables the pump, that is, makes it ready to dispense petrol.

Reads: supplied ***n***
Changes: $p : Pump$ with $p.pump\_id = n$, ***motor***, ***clutch***, ***display***
Sends:
Assumes:
Result: if ***p*** is enabled or out of service then no effect
otherwise:
Status of ***p*** is enabled.

Its ***display*** has been initialized.
its ***motor*** is running, and its ***clutch*** is not engaged.

$$
\begin{array}{|l}
\hline
\textit{enable\_pump} \\
\Delta PumpSystem \\
n? : PUMP\_ID \ [\textit{input declaration obtained from Reads section}] \\
p, p' : Pump \ [\textit{obtained from Changes section}] \\
\hline
p \in pumps \land p.pump\_id = n? = p'.pump\_id \\
(p.status = Enabled \lor p.status = Out\_of\_service) \Rightarrow pumps' = pumps \\
p.status = Disabled \Rightarrow p'.ident = p.ident \land p'.gha = p.gha \land \\
\quad p'.status = Enbaled \land (p'.display).volume = 0 \land \\
\quad (p'.display).grade = (p.display).grade \land (p'.display).cost = 0 \land \\
\quad ((p'.cma).motor).running = on \land ((p'.cma).clutch).engaged = off \land \\
\quad pumps' = (pumps \setminus \{p\}) \cup \{p'\} \\
\hline
\end{array}
$$

ΔPumpSystem defines two pump system configurations, one representing a configuration before execution of the operation (represented by unprimed variables) and the other representing a configuration after execution (represented by primed variables). In the next section we formalize an OO design for this operation.

4 Formalizing Fusion Design Models

In [6] we presented a formalization of OIGs that used the promotion mechanism of Z. We found this particular approach to be cumbersome, and it introduced notation (the θ notation) that was not handled well by the static and dynamic analysis tools available in our environment. In this section we present another approach that does not require the use of the θ notation.

Design Formalization Example

At the design level, additional classes may be required to implement a solution to the problem, thus there may be some objects in an OIG that are not objects of any class described by the analysis-level Object Model. Though Fusion does not have a Design-level Object Model (DOM), we have found the use of such a model useful in our integrated approach. A DOM can be constructed from information in the Fusion Class Descriptions (CDs). Classes that have objects as attributes in CDs are modeled in a DOM as aggregates in which the components are the object attributes. Object attributes can be bound (i.e., their lifetimes are bound to the lifetime of the aggregate), constant (i.e., they cannot be replaced by another instance), and exclusive (i.e., only the aggregate class can access them). These properties are indicated in the DOM. Part of the DOM for the pump system is given in Fig. 3 (this was obtained from the Class Descriptions given in [7], pages 169-170).

A formalized DOM is a characterization of valid design-level states of a system. In our design formalization technique, design models of operations are formalized in terms of relationships between before and after (valid) design states.

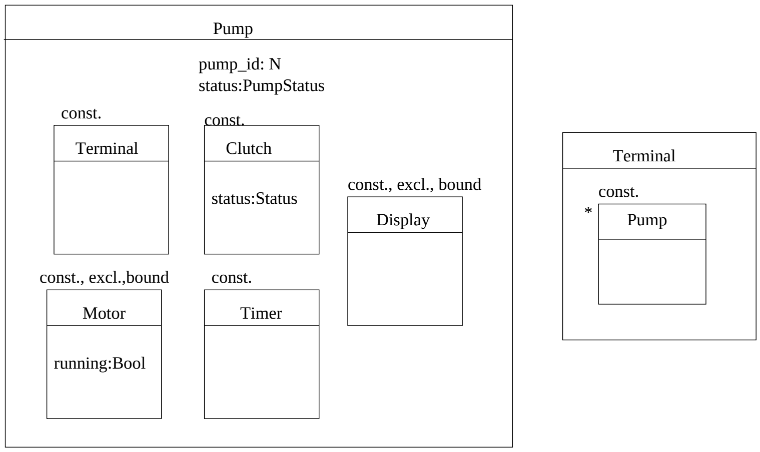

Fig. 3. Partial Design Object Model for *Pump*

In this section we outline the formalization of the OIG for the ***enable_pump*** system operation (see Fig. 2). The description associated with the OIG states:

> The invocation of ***enable_pump*** triggers the invocation of ***enable*** on a pump with identifier ***n*** (if there is no such pump then the operation halts, leaving the state unchanged). If the pump is in service and not already enabled then its display is reset, clutch is freed, and its motor is started (else the operation halts).

Our formalization of OIGs starts from the leaf objects (e.g., ***display***), and works its way backward towards the controller object (e.g., ***terminal***). Information about the effects of messages on objects is available from CDs (not given here). Below we give the formalizations of the ***enable_pump*** leaf messages.

reset

d, [*before state*]
d' : *Display* [*after state*]

$d'.ident = d.ident$
$d'.grade = d.grade$
$d'.volume = 0$
$d'.cost = 0$

free

c, c' : *Clutch*

$c'.ident = c.ident$
$c'.engaged = off$

start

m, m' : *Motor*

$m'.ident = m.ident$
$m'.running = on$

In the next step of the formalization, the effect of the ***enable*** operation is defined. The description associated with the OIG identifies two execution cases: the case when the pump is disabled and in service, and the other when the pump is enabled or out of service. The schemata formalizing the two cases are given below:

$$enablesub1 \mathrel{\widehat=} reset \wedge free \wedge start$$

$$\begin{array}{|l}
\multicolumn{1}{l}{enable_OK} \\ \hline
enablesub1 \\
p, p' : Pump \\
n? : \mathbb{N} \\ \hline
p.ident = p'.ident \\
p'.pump_id = p.pump_id = n? \\
d = p.display \wedge d' = p'.display \\
c = p.clutch \wedge c' = p'.clutch \\
m = p.motor \wedge m' = p'.motor \\
p.status = Disabled \wedge p'.status = Enabled \\
p'.timer = p.timer \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{No_enable} \\ \hline
p, p' : Pump \\
n? : \mathbb{N} \\ \hline
p.pump_id = n? \\
(p.status = Enabled \vee \\
\quad p.status = Out_of_Service) \\
p' = p \\ \hline
\end{array}$$

The schema ***enablesub*1** defines the combined effects of the suboperations. Note that in ***enable_OK*** no effect is defined for the ***terminal*** component of the pump. This is because the ***terminal*** component is an object in the OIG whose effect is not yet defined. On the other hand, the ***timer*** object does not take part in this operation and so its state is specified as unchanged here ($p'.timer = p.timer$).

The ***enable*** operation for the pump with identifier n can now be defined as:

$$enable \mathrel{\widehat=} enable\_OK \vee No\_enable$$

The formalization of the ***enable_pump*** operation is expressed as follows (there are two cases to consider: the pump is linked to the terminal, and the pump is not linked to the terminal):

$$\begin{array}{|l}
\multicolumn{1}{l}{enable_pump_OK} \\ \hline
enable \\
t, t' : Terminal \\
n? : \mathbb{N} \\ \hline
p \in t.pumps \\
t'.pumps = (t.pumps \setminus \{p\}) \cup \{p'\} \\
p.terminal = t \wedge p'.terminal = t' \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{enable_pump_notOK} \\ \hline
t, t' : Terminal \\
n? : \mathbb{N} \\ \hline
\forall p : t.pumps \bullet p.pump_id \neq n? \\
t = t' \\ \hline
\end{array}$$

$$enable\_pump \mathrel{\widehat=} enable\_pump\_OK \vee enable\_pump\_notOK$$

The next step of the formalization is to promote the ***enable_pump*** operation to the system level. This is done below:

$$\begin{array}{|l}
\multicolumn{1}{l}{SYSenable_pump} \\ \hline
\Delta DesignPumpSystem \\ \hline
\exists\, en : enable_pump \bullet \\
\qquad (en.t \in terminals \wedge terminals' = (terminal \setminus \{en.t\}) \cup \{en.t'\}) \\
\ldots \\
[\mathit{conjuncts\ stating\ that\ all\ other\ configuration\ elements\ are\ unchanged}] \\ \hline
\end{array}$$

Using the above, and a refinement relationship between ***DesignPumpSystem*** (not given here) and ***PumpSystem*** (given in previous section), one can establish that the design is consistent with the analysis model of the operation. Verification techniques for Z can be used for this purpose. See [16] for more details on design verification with Z.

5 Conclusion and future works

In this paper we have illustrated the integration of an informal, graphical OOM with a formal specification technique. We used the formal models to tie together models produced in a particular phase of development, and across development phases. At the analysis level, the formalized Object Model characterizes the state affected by the operations specified in the formalized Operation Model. The formalized Object Model provides terms that can be used to express the effect of system operations precisely. Using the formalized models one can rigorously do internal consistency checks of models, as well as consistency checks across the models.

At the design level, the formalization of an OIG produces a model that can be formally verified against the formalized Operation Model.

We are currently developing rigorous refinement techniques for Fusion design. The intent is to develop an analysis and design development environment in which the complementary nature of formal and informal OO methods is exploited. We are also extending the prototype tool FuZE to support the techniques outlined in this paper.

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. Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, Menlo Park, CA, Second edition, 1994.
3. Jonathan P. Bowen and J. Anthony Hall, editors. *Z User Workshop, Cambridge 1994,* Workshops in Computing. Springer-Verlag, New York, 1994.
4. Stephen M. Brien and John E. Nicholls. Z base standard. Technical Monograph PRG-107, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, November 1992. Accepted for standardization under ISO/IEC JTC1/SC22.
5. Jean-Michel Bruel, Robert B. France, Bharat Chintapally, and Gopal K. Raghavan. A Tool for Rigorous Analysis of Object Models. In *Proceedings of the* 20^{th} *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'96),* Santa Barbara, California, July 29–August 2 1996.
6. Jean-Michel Bruel, Robert B. France, Maria M. Larrondo-Petrie, Bharat Chintapally, and Gopal K. Raghavan. CASE-based Rigorous Object-Oriented Modeling.

In *Proceedings of the Northern Formal Methods Workshop,* Bradford, UK, 23–24 September 1996.
7. 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.
8. Dan Craigen, Sentot Kromodimoeljo, Irwin Meisels, Bill Pase, and Mark Saaltink. EVES: An Overview. In S. Prehn and W. J. Toetenel, editors, *VDM'91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, pages 389–405. Springer-Verlag, 1991. Volume 1: Conference Contributions.
9. Robert B. France and Jean-Michel Bruel. The Role of Integrated Specification Techniques in Complex System Modeling and Analysis. In *Proceedings of the Workshop on Real-Time Systems Education (RTSE'96),* Daytona Beach, Florida, 20 April 1996.
10. J. Anthony Hall. Specifying and Interpreting Class Hierarchies in Z. In Bowen and Hall [3], pages 120–138.
11. Jonathan A. R. Hammond. Producing Z Specifications from Object-Oriented Analysis. In Bowen and Hall [3], pages 316–336.
12. I. Jacobson. *Object oriented software engineering.* Addison-Wesley, 1992.
13. 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.
14. Xiaoping Jia. *ZTC: A Z Type Checker, User's Guide, version 2.01.* Division of Software Engineering, School of Computer Science, Telecommunication, and Information Systems, DePaul University, Chicago, IL, USA, May 1995. Available via anonymous ftp at `ise.cs.depaul.edu`.
15. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.
16. J. Michael Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, Englewood Cliffs, NJ, Second edition, 1992.
17. R. Wirfs-Brock and B. Wilkerson. *Designing object oriented software.* Prentice-Hall, 1990.

This article was processed using the LaTeX macro package with LLNCS style