Towards Rigorous Analysis of Fusion Models: The MIRG Experiences
R.B. France, J-M. Bruel and G. Raghaven
Towards Rigorous Analysis of Fusion Models: The MIRG Experiences

Robert B. France
Department of Computer Science & Engineering, Florida Atlantic University
Boca Raton, FL-33431-0991, USA

Jean-Michel Bruel
Laboratoire IRIT/SEIR, bat. 1R1
118, rte de Narbonne
31062 Toulouse Cedex, France

Gopal Raghavan
Department of Computer Science & Engineering, Florida Atlantic University
Boca Raton, FL-33431-0991, USA

Abstract
Most popular graphical object-oriented methods (OOMs) provide good mechanisms for handling complexity and constructs that facilitate the creation of abstract, highly-structured models of systems. These qualities make the use of OOMs in the development of complex systems attractive. On the other hand, the models produced by these methods are not amenable to rigorous semantic analysis.

Research carried out by the Methods Integration Research Group (MIRG) at Florida Atlantic University on integrating formal specification techniques (FSTs) with less-formal, but richly-structured, graphical OOMs is concerned primarily with developing precise and analyzable counterparts of the models produced by the OOMs. In this paper we give an overview of the current MIRG work on using FSTs to support rigorous analysis of OO models. Specifically, we outline our approach to formalizing Fusion’s design and analysis models, and indicate how the formal models can be used to support semantic analysis across requirements and design development stages.

Keywords: Formal specification techniques, Fusion, Integrated formal and informal specification techniques, Object-oriented methods, Z.

1 Introduction
In most mature graphical object-oriented methods (OOMs), the syntax of the modeling constructs is well-defined, but the underlying semantics are loosely described using a mixture of natural language and diagrams. The lack of modeling precision can often lead to disagreements within a software development team over appropriate use of the modeling constructs, and limit their ability to rigorously analyze the models.

The loose interpretations provided by textbooks on methods are often intentional: their intended purpose is to guide the use of the notation and not to prescribe how they will be used and interpreted. This allows the developer to tailor the interpretations of the notation (within the loose constraints provided by the guidelines) to his/her usage. If the models are to be used as a basis for further development, or as a communication medium in a team environment, it is essential that there be a means for determining the precise meaning of the models.

---

*This work was partially funded by NSF grant CCR-9410396.*
Towards Rigorous Analysis of Fusion Models: The MIRG Experiences

One approach to making OOMs more precise and amenable to rigorous analysis is to provide modelers with mechanisms that allow them to express modeled concepts in a precise and analyzable manner. Work on integrated formal and informal specification techniques have as their goal the development of such mechanisms. Here, integration means providing a bridge from the informal modeling constructs to the formal notation (e.g., see [6]). Over the last year, the Methods Integration Research Group (MIRG) at Florida Atlantic University has worked primarily on formalizing OO modeling techniques, in particular, the Fusion method [5] (e.g., see [1, 6]), and, more recently, the Unified Modeling Language (UML) [2] (e.g., see [9]). We used the formal notation $Z$ [10] in our work because of its maturity, expressiveness, the availability of good analysis tools, and its apparent compatibility with OO analysis modeling concepts.

Our previous work focused on the formalization and analysis of OO analysis models, in particular, the formalization of class diagrams. This work resulted in a prototype tool, called FuZE (Fusion/Z Environment) [3], that generates Z specifications from Fusion Object Models. In this paper we present our most recent work on refining and extending the core concepts underlying FuZE. Our refinements have resulted in new Object Model transformation rules based on an extensional view of classes (i.e., a class is viewed as a collection of instances). The extensional semantics makes it unnecessary to model object identifiers explicitly (as in our previous approach). This simplified the modeling of the more dynamic design models by removing the need to explicitly maintain the uniqueness of object identifiers, especially when objects are created and destroyed. We have also developed a more stable formalization technique for Fusion design models that is an improvement over the technique described in [4]. We also describe our planned extensions to the FuZE prototype tool.

In section 2 we outline the new rules for formalizing Fusion analysis models. In section 3 we illustrate our approach to formalizing Fusion design models, and outline how the formalizations can be used to establish the consistency of design models against the analysis models. In section 4 we outline our planned extensions to the FuZE prototype, and in section 5 we conclude with an outline of our plans to extend the work described in the paper.

2 Formalization of Fusion Analysis Models

In Fusion, the structural aspect of a problem is modeled by an Object Model, and the required behavior of a system is modeled by an Operation Model that consists of operation schemata and a lifecycle model. A Fusion operation schema describes the required behavior of a system operation in terms of the conditions under which the operation can be executed and the externally observable effect of the operation. A lifecycle model defines restrictions on the relative ordering of operation invocations.

In this section we illustrate the formalization of Fusion's Object and Operation Models. To illustrate our approach we formalize the Object Model and an operation schema for a system that keeps track of depots that store environmentally damaging chemicals (taken from [5]). We will refer to this system as the ECO system. The ECO system consists of depots, each having a loading bay (for offloading drums of chemicals) and a set of storage facilities (called store buildings). The system is required to generate reports on the status of each depot, and allocation lists that list drums allocated to, and stored in store buildings.

2.1 Formalization of the Fusion Object Models

The approach to formalizing the Object Model described here differs from our earlier work in that the formalization is not based on an intensional model of a class. The intensional model we used required the explicit modeling of object identifiers, which meant that constraints expressing the uniqueness of identifiers had to be explicitly stated. The problem is exacerbated at the design level, where management of object identifiers accounts for a disproportionately large number of conjuncts. This problem led us to consider an extensional semantics for classes. In this paper, classes are extensionally defined as sets of instances drawn from a Z basic type. Instances are related to their attributes and/or components via Z functions.

Below we outline the rules for formalizing Object Model constructs, and give examples of their application. The Object Model in Fig. 1 will be used to illustrate the rules.

---

1See http://www.cse.fau.edu/research/MIRG/
Towards Rigorous Analysis of Fusion Models: The MIRG Experiences

2.1.1 Formalization of simple classes

A class that is not an aggregate or a subclass is referred to as a simple class. A Z schema formalizing a class is called a class schema. The following transformation rule describes how class schemata for simple classes are obtained.

**Rule 1 (Formalizing Simple Classes)** The class schema for a simple class has the following form:

```
[CLASSNAME]

Class Attributes
Class attributes
[Constraints on attribute values provided by specifier]
```

```
ClassName
classobjects : PCLASSNAME
classobject_attribs : CLASSNAME ➔

Class Attributes
dom classobject_attribs = classobjects
[Constraints on instances provided by specifier]
```

In the above, classobjects is a set of instances of the class, and the classobject_attribs function relates instances to their attributes.

The set CLASSNAME can be viewed as the set of object identifiers for the class.

The Z schemata for the simple classes in the ECO system Object Model are given below:

```
[LOADINGBAY, BUILDING, REPORT, DRUM]

LoadingBay
loadingbays : P LOADINGBAY

Building
buildings : P BUILDING
```

Figure 1: The ECO System Object Model
Towards Rigorous Analysis of Fusion Models: The MIRG Experiences

\[
[\text{CHEMICAL\_TYPE, IDENTIFIER}]
\]

<table>
<thead>
<tr>
<th>Drum_Attributes</th>
<th>Drum_Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{chemical_type : CHEMICAL_TYPE}</td>
<td>\text{drums : P DRUM}</td>
</tr>
<tr>
<td>\text{identifies : IDENTIFIER}</td>
<td>\text{drum_attrs : DRUM}</td>
</tr>
<tr>
<td>\text{dom drum_attrs = drums}</td>
<td>\text{dom drum_attrs = drums}</td>
</tr>
</tbody>
</table>

2.1.2 Formalizing binary relationships

Relationships between classes are defined by Z schemata called relationship schemata. The following rule outlines the form of relationship schemata.

**Rule 2 (Formalizing Binary Relationships)** A binary relationship, \( \text{rel} \), between two classes defined by class schemata \( \text{ClassA} \) and \( \text{ClassB} \) is formalized by a Z schema, called a relationship schema, of the following form:

\[
\begin{align*}
\text{Rel} & \subseteq \text{ClassA} \times \text{ClassB} \\
\text{rel} & : \text{CLASSA} \leftrightarrow \text{CLASSEB} \\
\end{align*}
\]

The multiplicity constraints are expressed in terms of Z macros that are essentially short-hand forms of more elaborate Z predicate expressions. The above transformation rule is used to formalize the \text{neighbours} and \text{contains} relationship in the ECO system Object Model.

\[
\begin{align*}
\text{Neighbours} & \subseteq \text{BUILDING} \times \text{BUILDING} \\
\text{neighbours} & : \text{BUILDING} \leftrightarrow \text{BUILDING} \\
\text{neighbours} \in 1 & \rightarrow \ast [\text{buildings, buildings}] \\
\forall b : \text{buildings} & \bullet (b, b) \notin \text{neighbours} \\
\end{align*}
\]

In \text{Contains}, \text{contains} \in 1 \rightarrow \ast [\text{loadingbays, drums}] is a macro whose expansion results in a predicate expressing the 1-to-many constraint between elements in \text{loadingbays} and \text{drums}.

2.1.3 Formalizing Aggregate Classes

An aggregate class is a class that contains class components that may be related. In Fusion, an aggregate does not imply lifetime binding of components and the aggregate. An aggregate allows one to treat an Object Model substructure as a class. Below we give the form of Z schemata defining aggregate objects.
Towards Rigorous Analysis of Fusion Models: The MIRG Experiences

Rule 3 (Formalizing Aggregate Classes) An aggregate class, AggClass, with attributes defined by an attribute schema, AggAttributes, component classes defined by schemata Comp_1, ..., and relationships rel_1 between CLASS_1 and CLASS_2, ..., is formalized by a schema that has the following form:

\[
\begin{align*}
\text{AGGCLASS} & \\
\text{AggClass} & \quad \text{[component class schema]} \\
\text{Comp}_1, \ldots & \quad \text{[component class schemata]} \\
\text{Rel}_1, \ldots & \quad \text{[relationship schemata; Rel_1 defines rel_1]} \\
\text{aggo}{\text{bjects}} & : \text{P AGGCLASS} \quad \text{[aggregate instances]} \\
\text{agg}{\text{attributes}} & : \text{AGGCLASS} \rightarrow \text{AggAttributes} \quad \text{[aggregate attributes]} \\
\text{agg}{\text{comp}}_1 & : \text{AGGCLASS} \rightarrow \text{COMP}_1; \ldots \quad \text{[aggregate components]} \\
\text{agg}{\text{rel}}_1 & : \text{AGGCLASS} \rightarrow (\text{CLASS}_1 \rightarrow \text{CLASS}_2; \ldots) \quad \text{[aggregate relationships]} \\
\cdots & \quad \text{[Other constraints provided by specifier]}
\end{align*}
\]

Note that a relationship in an aggregate instance must be a subrelation of the larger relationship between the classes concerned. Also, the range restriction for a function mapping aggregates to a particular class of components (e.g., agg.comp_1) can be strengthened by making the range equal to (rather than a subset of) the set of instances of the function's target class.

Below we apply Rule 3 to the formalization of the Depot aggregate class.

\[
\begin{align*}
\text{DEPOT} & \\
\text{Neighbours} & \\
\text{LoadingBay} & \\
\text{depots} & : \text{P DEPOT} \\
\text{depot.buildings} & : \text{DEPOT} \rightarrow \text{PBUILDING} \\
\text{depot.buildingrel} & : \text{DEPOT} \rightarrow (\text{BUILDING} \rightarrow \text{BUILDING}) \\
\text{depot.loadingbay} & : \text{DEPOT} \rightarrow \text{LOADINGBAY}
\end{align*}
\]

\[
\begin{align*}
\text{dom depot.buildings} & = \text{depots} \\
\text{dom depot.buildingrel} & = \text{depots} \\
\text{dom depot.loadingbay} & = \text{depots} \\
\text{ran depot.loadingbay} & \subseteq \text{loadingbays} \\
\text{ran depot.buildings} & \subseteq \text{buildings} \\
\forall b & : \text{ran depot.buildings} \subseteq \text{buildings} \\
\forall d & : \text{depots} \bullet \text{depot.buildingrel}(d) \subseteq \text{neighbours} \wedge \\
& \text{dom(depot.buildingrel(d))} \subseteq \text{depot.buildings}(d) \wedge \\
& \text{ran(depot.buildingrel(d))} \subseteq \text{depot.buildings}(d)
\end{align*}
\]

2.1.4 Formalizing specialization hierarchies

In our approach, a specialization hierarchy represents a subdivision of the root superclass object space, where a root superclass is a class that is not a subclass of another class. Subclasses can partition their superclass's object space, or there could be superclass instances that are not subclass instances.

Below we outline the rule for formalizing specialization hierarchies.
Rule 4 (Formalizing Specialization Hierarchies) A subclass, SubClass, in a hierarchy with a root superclass whose objects are represented by the set, SUPERCLASS, is specified as follows:

$$\text{SubClass}$$
$$\text{subclassobjects} : P \text{SUPERCLASS}$$
$$\text{subclassattrs} : \text{SUPERCLASS} \rightarrow \text{SubClass\_Attributes}$$
$$\text{dom subclassattrs} = \text{subclassobjects}$$
[Other constraints provided by specifier]

The schema SubClass\_Attributes defines the attributes that are particular to the subclass. Given a specialization hierarchy with a root superclass defined by a schema, Superclass, and subclasses defined by schemata Sub\_1, ..., Sub\_n, the following Z schema defines the properties of the hierarchy:

$$\text{SuperClassStruct}$$
$$\text{SuperClass}$$
$$\text{Sub\_}1; \ldots; \text{Sub\_n}$$
[Constraints on hierarchy]

Some of the constraints that are generated for specialization hierarchies are given below:

- If the subclasses Sub\_p, ..., Sub\_q partition their superclass, then the following constraint must be present:

$$\langle \text{sub\_p\_objects}, \ldots, \text{sub\_q\_objects} \rangle \text{ partition super\_objects}$$

- If the subclasses Sub\_p, ..., Sub\_q are disjoint then the following constraint must be present:

$$\text{disjoint} \langle \text{sub\_p\_objects}, \ldots, \text{sub\_q\_objects} \rangle$$

Below we give the formalization of the Building specialization hierarchy in the ECO system Object Model:

$$\text{AdminBuilding}$$
$$\text{adminbuildings} : P \text{BUILDING}$$

$$\text{StoreBuilding\_Attributes}$$
$$\text{max\_capacity} : N$$

$$\text{StoreBuilding}$$
$$\text{storebuildings} : P \text{BUILDING}$$
$$\text{storebuildingattrs} : \text{BUILDING} \rightarrow \text{StoreBuilding\_Attributes}$$
$$\text{dom storebuildingattrs} = \text{storebuildings}$$

$$\text{BuildingStruct}$$
$$\text{Building}$$
$$\text{AdminBuilding}$$
$$\text{StoreBuilding}$$
$$\langle \text{adminbuildings, storebuildings} \rangle$$
$$\text{partition buildings}$$

2.1.5 Putting it all together

Once the parts of the Object Model are specified, a Z schema for the model is obtained by including schemata defining the specialization hierarchies, aggregates, relationships, and classes in a configuration schema. A configuration schema can be viewed as a characterization of the valid states of a system. The configuration schema for the ECO system is given below:
Towards Rigorous Analysis of Fusion Models: The MIRG Experiences

ECOSystem
BuildingStruct
AllocationListStruct
ReportStruct
Contains
Monitors

2.2 Formalization of Fusion operation schema

An operation schema describes a system operation in terms of data read by the operation, items changed by the operation, events sent to other objects, conditions that must hold for the operation to execute, and the effect of the operation. An example of an operation schema is given below:

<table>
<thead>
<tr>
<th>Operation</th>
<th>depot_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Requests a status report on storage buildings and loading bay.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reads</th>
<th>stores and loading_bay of depot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changes</td>
<td>new status_report</td>
</tr>
<tr>
<td>Sends</td>
<td>manager: {status_report}</td>
</tr>
<tr>
<td>Assumes</td>
<td></td>
</tr>
<tr>
<td>Result</td>
<td>A status_report of the stores in the depot and the loading_bay has been sent to the manager.</td>
</tr>
</tbody>
</table>

From the above it can be seen that Fusion’s operation schemata are expressed mostly in natural language. Automatic generation of formal specifications from such models is not conceivable (given current technology), but the structured form permits generation of partial formal specifications. We developed the following guidelines for formalizing Fusion’s operation schemata:

1. The operation name is used as the name of the Z operation schema.

2. If the Changes section is non-empty, then the declaration $\delta SystemConfig$, where SystemConfig is the configuration schema for the Object Model, is made in the Z operation schema. This declaration indicates that a change in the configuration (state) is possible. More precisely, it introduces unprimed variables representing configuration elements before the operation’s execution, and primed variables representing configuration elements after execution, and a predicate expressing invariant properties on before- and after-configurations. If the Changes section is empty, then the declaration $\Xi SystemConfig$, that introduces a predicate equating the before- and after-configurations, is made.

3. In the Reads section, supplied elements are inputs to the operation, and the other elements are elements in the before-configuration. For each supplied element an input variable (with a name ending in ‘?’) is declared in the Z schema. For each of the other elements in this section a variable representing the element is declared, and a conjunct stating that the element is in the set of instances for the class in the before-state must be stated in the predicate part.

4. In the Changes section, each changed element is represented by a variable. Other variables representing components and/or attributes of these elements can also be added if they make the predicate part easier to construct. The new elements are instances created by the operation. For each new element a conjunct stating that the element is in the after-state, but not in the before-state is required. For all other elements, conjuncts stating that the elements are in the before-, and after-states are needed.

5. The data declared in the Sends section are declared as output variables in the Z schema.

6. The description in the Assum es section is reexpressed as a conjunct in the predicate part of the Z schema. The conjunct is the precondition for the operation.

7. The description in the Result section is reexpressed as a conjunct describing the effect of the operation in the predicate part of the Z schema.
Towards Rigorous Analysis of Fusion Models: The MIRG Experiences

Because of the natural language used, the formalization of the Assumes and Result sections requires significant human input. However, some of the above guidelines identify conjuncts that must be present in the Z schemata.

We used the above guidelines to develop the following Z schema for the depot\_status operation:

```
Depot Status
Ξ BuildingStruct
Ξ Contains
Δ AllocationListStruct
Δ ReportStruct
Δ Monitors
depot : DEPOT [target depot]
status_report : REPORT [new status report]
alloc_list : ALLOCATIONLIST [new allocation list]
SR_drums : PDRUM [drums on loading bay]
depot_holdings : BUILDING ⇒ DRUM [contents of stores]

∃ d : depots • d = depot
status_report \notin reports [new object not in before-state]
alloc_list \notin allocationlists
... [Definitions of SR_drums and depot_holdings]
status_drums' = status_drums \cup \{status_report ⇒ SR_drums\}
status_alloclist' = status_alloclist \cup \{status_report ⇒ alloc_list\}
status_allocations' = status_allocations \cup \{alloc_list\}
allocation_holds' = allocation_holds \cup \{alloc_list ⇒ depot_holdings\}
monitors' = monitors \cup \{depot ⇒ status_report\}
... [All other variables are unchanged]
```

3 Formalization of Fusion Design

The following models are produced during design in Fusion:

- **Object interaction Graphs** depict the object interactions that take place when system operations are invoked.

- **Visibility Graphs** model access restrictions and lifetime bindings of objects.

- **The Inheritance Graph** defines the inheritance structure of the solution.

- **A Class Description** is produced for each class in the design.

In [4] we presented our early attempt at formalizing Fusion's Object Interaction Graph. That approach used the Z promotion mechanism to promote the effect of messages on individual objects to the system level. We found that this approach offered few opportunities for automated support, and, furthermore, the Z analysis tools (e.g., the animation tool) could not handle analysis of schemata that explicitly used schema bindings.

This led us to the approach outlined in this section. In our current approach, the formalization is based on information in all four of the Fusion design models. The result is a more complete and precise design model that can be verified against analysis models. Key to our current approach is the **Design Object Model** (DOM), a design model that brings together descriptions of the static aspects of a design found in Fusion models. The DOM allows us to specify functionality in the same manner analysis-level functionality is defined, that is, as transformations on (design-level) configurations. The effect of OIGs is formalized in terms of transitions on the configurations (states) defined by the DOM. Verification of design-level functionality involves establishing a relationship between the analysis Object Model and the DOM, and checking that the effects of OIGs are consistent with the effects specified for the system operations at the analysis level.
3.1 Formalization of Design Object Models

Descriptions of static features of a solution found in Fusion design models are brought together in a DOM. Specifically, Class Descriptions and the analysis Object Model provide most of the information needed to build a DOM. The development of a DOM is guided in our approach by few rules, for example, an object, $O_1$, that is bound to another object $O_2$, is modeled as an object attribute of the $O_2$ class. If an object, $O_1$, is declared as shared but not bound to $O_2$ in a Class Description, then a reference (depicted as a directed arrow) from the $O_1$ class to the $O_2$ class is present in the DOM. Cardinality constraints for references are obtained from the analysis Object Model. These rules are all that are needed to obtain a DOM for the ECO system.

Part of the DOM (without annotations) for the ECO system is given in Fig. 2. We do not show the subclasses of Allocation List and Report because they are the same as in the analysis class model (see Fig. 1). The missing DOM annotations express additional constraints on object structure.

![Diagram of ECO System Design Object Model]

Figure 2: The ECO System Design Object Model

The formalization of DOMs is no different than the formalization of analysis models. Below we give some of the schemata resulting from the formalization of the ECO system DOM.

- **Store Attributes**:
  - max_capacity : $N$
  - neighbours : F BUILDING
  - drums : F DRUM

- **Depot Attributes**:
  - stores : F BUILDING
  - monitor : MONITOR
  - loadingbay : LOADINGBAY

- **LoadingBay Attributes**:
  - monitor : MONITOR
  - drums : F DRUM

- **Store Building**
  - stores : F BUILDING
  - store_attributes : BUILDING $\rightarrow$ Store_Attributes

- **DOM store_attributes = stores**
  - $\forall s : store\ s \in (store\_attributes\ (s)).neighbours$
  - $\forall s_1, s_2 : store\ s_1 \neq s_2 \Rightarrow (store\_attributes\ (s_1)).drums \cap (store\_attributes\ (s_2)).drums = \emptyset$
Towards Rigorous Analysis of Fusion Models: The MIRG Experiences

\begin{align*}
\text{depots} & : F \text{ DEPOT} \\
\text{depot\_attrs} & : \text{DEPOT} \rightarrow \text{Depot\_Attributes} \\
\text{dom depot\_attrs} & = \text{depots} \\
\forall d_1, d_2 : \text{depots} | d_1 \neq d_2 \Rightarrow \\
& (\text{depot\_attrs}(d_1)).\text{stores} \cap (\text{depot\_attrs}(d_2)).\text{stores} = \land \\
& (\text{depot\_attrs}(d_1)).\text{monitor} \neq (\text{depot\_attrs}(d_2)).\text{monitor} \land \\
& (\text{depot\_attrs}(d_1)).\text{loadingbay} \neq (\text{depot\_attrs}(d_2)).\text{loadingbay}
\end{align*}

The design-level configuration schema, characterizing the valid design object structures for the ECO system, states the following constraints:

1. Every store building is part of a depot in a configuration.
2. Every loading bay is part of a depot in a configuration.
3. The drums in a configuration are exactly those drums in store buildings and drums on loading bays.
4. A drum on a loading bay cannot be in a store building.
5. The store buildings associated with a loading bay are precisely those storing building in the loading bay’s depot.
6. The neighbours of a store building in a depot is a subset of the depot’s store buildings.
7. Monitors are unique across depots and loading bays.
8. In a configuration, the depot referred to by a report is a depot in the configuration.
9. The drums associated with a store building in an allocation list are a subset of the drums in the store building.
10. The drums associated with a store building in a status allocation list are precisely those drums stored in the store building.
11. The store buildings in a status report are store buildings of the depot associated with the report.

Part of the configuration schema for the ECO system DOM is given below:

\begin{align*}
\text{ECO\_DesignState} \\
\text{Depot} \\
\text{LoadingBay} \\
\text{BuildingStruct} \\
\text{AllocationStruct} \\
\text{ReportStruct}
\end{align*}

\begin{align*}
\text{stores} & = \bigcup \{ d : \text{depots} \Rightarrow (\text{depot\_attrs}(d)).\text{stores} \} \\
\text{loadingbays} & = \big\{ d : \text{depots} \Rightarrow (\text{depot\_attrs}(d)).\text{loadingbay} \big\}
\end{align*}

3.2 Formalization of an OIG

The effect of messages on objects are defined in method specifications found in Class Descriptions. We use these informal descriptions to formalize the effects of OIGs. The use of natural language to specify methods severely limits the automated support possible for the formalization of the OIG. We are developing an intermediate notation that can help bridge the gap between the natural language descriptions and Z specifications.
Towards Rigorous Analysis of Fusion Models: The MIRG Experiences

We have developed a technique that human specifiers can use to systematically formalize an OIG. In this section we illustrate the technique by formalizing the OIG for the \texttt{depot\_status} operation shown in Fig. 3 (adapted from [5]).

The narrative accompanying the OIG is given below:

\textbf{operation Depot::depot\_status():}

Create a new allocation list. \hfill (1)  
For all \texttt{s} in the \texttt{stores} collection put details of the store contents into the allocation list. \hfill (2)  
Retrieve details of the contents of the loading bay. \hfill (3)  
Create a new status report initialized with the loading bay contents and the allocation list. \hfill (4)  
Notify the depot monitor of the status report. \hfill (5)

In this paper a \textit{parent message} is one that spawns other messages (called \textit{child messages}). For example, \texttt{depot\_status} is a parent message with respect to the messages labeled (2), (3), (4), and (5). A \textit{leaf message} is one that does not spawn other messages. Formalization of an OIG starts by defining the effects of leaf messages, then defining the effects of parent messages in terms of the effects of their child messages.

The formalization of the leaf message (1) (\texttt{create\_list}) is given below:

\begin{verbatim}
create_list
  \exists ECO\_DesignState
  al : ALLOCATIONLIST
  al \notin allocations
  statusallocations' = statusallocations \cup \{al\}
\end{verbatim}

When formalizing messages to collections, the effect on a message on a single object in the store is specified first. The resulting schema is then used to specify the collective effect of the messages. For
Towards Rigorous Analysis of Fusion Models: The MIRG Experiences

example, formalization of the effects of `store_content` messages results in the following:

Effect on a single store object:

```
store_content

| ECO_DesignState
| st : BUILDING
| s! : BUILDING × DRUM
| st ∈ stores
| \( \text{first}(s!) = \text{st} \land \text{second}(s!) = (\text{store_attrs}(s))_.\text{drums} \)
```

Collective effect on depot stores:

```
store_content_coll

| ECO_DesignState
| d : DEPOT
| s_outs! : (BUILDING × DRUM)
| store_effects : store_content
| d ∈ depots
| (depot_attrs(d)).stores = \{ s : store_effects • s.st \}
| s_outs! = \{ s : store_effects • s.s! \}
```

The other leaf messages are similarly defined. In general, the effect of a leaf message is described by a schema of the form:

```
Op_leaf

| DesignState (or ΔDesignState)
| object : OBJECTCLASS
| in? : IN
| out! : OUT
| pre1 ⇒ effect1
| pre2 ⇒ effect2
| ...
| \( \neg (pre1 \lor pre2 \lor ...) \Rightarrow objects = objects' \)
```

In the above schema \( pre1, pre2 \ldots \) are the conditions (called `preconditions`) that must be true for the message to have the defined effects on the object represented by `object`. In our formalization, if none of the preconditions hold, then the state (i.e., the set of object instances) is unchanged.

Below we give the declarations for operation schemata of the remaining leaf messages (these declarations will be used to illustrate the formalization of parent messages):

```
put_in_loopeffect

| ECO_DesignState
| contents? : F (BUILDING × F DRUM)
| a : ALLOCATIONLIST
| a_atrib : AllocationList_Attributes
| ...
```

```
Idcontents

| ECO_DesignState
| d : DEPOT
| Iddrums! : F DRUM
| ...
```
Towards Rigorous Analysis of Fusion Models: The MIRG Experiences

In general, the formalization of the effect of a parent message, associated with child messages $Child_1$, $\ldots$, $Child_n$, results in a schema of the following form:

$$Parent_{\text{op}} \triangleq (\text{Int}_0 \upharpoonright Child_1 \upharpoonright \text{Int}_1 \upharpoonright \ldots \upharpoonright Child_n \upharpoonright \text{Int}) \lor$$
$$\ldots$$

Where $\text{Int}_p$ ($p$ is an integer) specifies the effect of an internal action taking place before the interaction described by the following schema, $Child_p$, takes place. The interactions that take place can vary from invocation to invocation of a particular parent operation (e.g., the particular behavior exhibited may depend on the inputs to the operation). Each disjunct represents an alternative behavior. Before composition, the inputs and outputs declared in the schemata describing the effects of child messages are renamed to reflect passing of data from one object to another, and to remove the symbols '!:' and ':'.

The $Z$ schema for the $\text{depot\_status}$ operation is defined as follows:

$$\text{depot\_status} \triangleq \text{create\_list} \upharpoonright$$
$$\text{store\_content\_coll}[s\_outs/s\_outs] \land \text{put\_in\_loose\_effect}[s\_outs/\text{contents?}, a/l[a]]\upharpoonright$$
$$\ldots$$
$$\text{create\_report}[\text{ldrums}/\text{drums?}, a/l[\text{list?}]]\upharpoonright$$
$$\text{notify\_status}[\text{report}/\text{report?}]\upharpoonright$$

### 3.3 Towards rigorous consistency checking

At the design level, the formalization of the Object Interaction Graph produces a model that can be formally verified against the formalized Operation Model. Our formalization approach allows us to base consistency checking on classical $Z$ refinement proofs [10].

We use the following approach to verify OIG formalizations against formalized Operation Models:

- Define a schema linking the design state (as characterized by the configuration schema of the DOM) and the analysis state (as characterized by the configuration schema of the analysis Object Model).

- Prove the applicability and correctness theorems for the operation. The applicability theorem essentially states that if the analysis level operation can start in a state, then the design level operation must be able to start in the corresponding state. The correctness theorem states that the effects defined at the two levels must be consistent with each other.

The proof obligations can be stated as follows:

Let $Aop$ and $Cop$ be respectively the abstract operation (from the Operation Model) and the concrete operation (from the OIG), and let $\text{Abstraction}$ be a schema relating the concrete state space (including inputs and outputs) to the abstract state space. $Cop$ is an operation refinement of $Aop$, if:

- The applicability theorem:

$$\text{pre } Aop \land \text{Abstraction } \Rightarrow \text{pre } Cop$$

- The correctness theorem: ($Cop$ can be more deterministic than $Aop$)

$$\text{pre } Aop \land \text{Abstraction } \land Cop \land \text{Abstraction'} \Rightarrow Aop$$
Towards Rigorous Analysis of Fusion Models: The MIRG Experiences

4 Enhancing The FuZE Prototype

The early version of the Fusion to Z rules were implemented in a prototype tool, called FuZE (Fusion/Z Environment), built on top of the Paradigm-Plus OO modeling CASE tool (Release 2.01) [8]. Within FuZE one can invoke Z tools to analyze the Z specifications produced by the transformation subsystem. In the current version of FuZE one can call the typechecker ZTC, and the animator ZANS from within the tool.

A context level model of FuZE is shown in Fig. 4. Using a Fusion Object Diagram as input, the FuZE formalization process generates a file that is readable by the Z analysis tools. The ZTC typechecking tool [7] is launched by running a script, and it produces a type-checked Z specification in a readable ASCII format. A Latex file can be produced from the ZTC produced file by calling another script. LaTeX can be launched within FuZE to produce a Latex'd Z specification. The Z animator, ZANS, can also be launched within FuZE to formally analyze the Z specifications.

The FuZE user manual is available on the Web2, and students in the graduate software engineering courses have used it on modeling projects.

Building FuZE on top of a commercial CASE tool has the advantage of providing an industrial-strength environment, but it also has the disadvantage that most researchers do not have free access to the tool. MIRG is currently working on extending and broadening the scope of the FuZE prototype, so that it can be used with a variety of CASE tools, OO models, and formal notations.

Our current focus is on extending the tool to other graphical OOMs, and making it independent of particular CASE tool representations and meta-models. To facilitate portability of the new environment, we will develop a common representation of OO models, and base the transformations to a formal notation on this representation. This will make the OO model-to-formal model transformation independent of a particular tool vendor’s internal meta-model and representation. Users of particular CASE tools will have to create scripts that translate from the internal representation of the tool to the common representation in order to use our formalization tool.

The Common Data Interchange Format (CDIF)3 is one means of representing the models so that it is accessible to all CDIF compliant CASE tools. Though CDIF provides a good interface format, importing data is difficult. An alternative means of representing the meta-information is to define SGML tags for the various OO modeling constructs. SGML tags are easy to read and use. Furthermore, there are many public domain programs that are available on the internet to parse/convert SGML documents. SGML tags also facilitate manual input of data. Once the meta-information has been captured and represented in a well defined format, it becomes easy to dynamically parse the data and process it. Algorithms similar to the ones used in FuZE can then be used to formalize, typecheck, and animate the models. A context level model of the proposed extension to FuZE is shown in Fig. 5.

2See http://www.cse.fau.edu/research/MIRG/FUZE/
3See http://www.cdif.org/
Towards Rigorous Analysis of Fusion Models: The MIRG Experiences

Figure 5: Context level model of the proposed extension to FuZE

5 Conclusions

In this paper we describe the results of our most recent work on formalizing Fusion analysis and design models. Our previous work on formalizing Fusion analysis models resulted in a prototype tool that generates Z specifications from Object Models. The prototype was used by students on class projects, and by members of the Methods Integration Research Group (MIRG)\(^4\) on case studies. This use showed clearly the benefit of automated support. In our experimentation on FuZE we used ZTC 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). Another benefit of the formalization is that it uncovers deficiencies in the modeling technique. When such deficiencies are addressed, a well-defined notion of well-formed models can be formulated and used to drive the analyses of the models in CASE tools.

Our attempts to extend the formalization to the design stages revealed some problems with our former formalization technique. This led to the formalization approach described in this paper.

We are currently developing a design notation for describing object operations and interactions that will facilitate the automatic generation of Z specifications from analysis operation schemata and from Object Interaction Graphs. This intermediate notation will help bridge the gap between the informal Fusion descriptions and the more precise Z notation. We are also working on developing support for rigorous refinement of Fusion design models, and extending our formalization technique to cover inheritance structures involving redefinition and polymorphism. The objective of our research is to create a development method that exploits the complementary nature of informal structured and formal development techniques.

References


\(^4\)see: http://www.cse.fau.edu/research/MIRG

2nd BCS-FACS Northern Formal Methods Workshop
Towards Rigorous Analysis of Fusion Models: The MIRG Experiences


