

UMLtranZ: An UML-Based Rigorous Requirements Modeling Technique

Robert B. France *; Emanuel Grant
Department of Computer Science
Colorado State University
Fort Collins, CO 80523, USA.

Jean-Michel Bruel
Laboratoire TASC
Department of Computer Science
University of Pau, France

January 13, 2000

Abstract

The high-quality modeling experiences embedded in the *Unified Modeling Language* (UML) make its application to complex systems desirable. On the other hand, the lack of precise semantics for UML notation hinders rigorous analysis of the models it produces. One approach to tackling this problem is to transform the imprecise models to analyzable models that express the intended behavior and structure precisely. In this paper we present a requirements modeling technique, called *UMLtranZ*, that utilizes UML Class Diagrams and Use Cases, which are expressed in a variant form of the Fusion Operation Model, and supports their transformation to Z specifications. The transformation of the models and the analysis of the resulting Z specifications provide opportunities for identifying ambiguities, gaps, and inconsistencies in the requirements that are being modeled.

1 Introduction

The *Unified Modeling Language* (UML) [28] is a set of notations for modeling systems from a variety of views using object-oriented (OO) concepts. A deterrent to the use of UML-based techniques for modeling the requirements of complex systems is the lack of support for rigorous analysis that goes beyond syntax checking. While the developers of the UML contend that the UML has a precise semantics, they chose to express the semantics in natural language. A problem is that the natural language descriptions can be misinterpreted, leading to confusion over intended meaning. Furthermore, subtle consequences that can lead to a deeper understanding of the concepts which, in turn, can result in more effective use of the notations, can get lost in an informal treatment of semantics.

The lack of a precisely defined semantics for models produced by OO techniques can also create the following problems during software development:

- Developers in a team can spend considerable time resolving disagreements over usage and interpretation of modeling constructs. Often, one cannot resolve the disputes by referring to the standard or text book descriptions of the techniques because the descriptions are informal or are presented in terms of examples that are not applicable to the problem being modeled.

*This work was partially funded by NSF grants CCR-9410396, CCR-9803491.

- Once the models are developed, it is difficult to give convincing arguments that they adequately capture the intended properties.
- It is difficult to ascertain that models of different views of the system are consistent with each other.
- It is difficult to argue convincingly that models produced in the design phase are consistent with the requirements models.

Despite these problems, informal OO analysis (OOA) techniques that are based on expert modeling experiences have significant strengths that cannot be lightly dismissed. In particular, some provide good abstraction and structuring mechanisms that permit the modeling of problem-oriented concepts from a variety of perspectives. Furthermore, the use of natural language statements and graphical representations in informal models is often preferred because they are easy to create, and, when properly created, are easier to understand than cryptic, mathematically-based expressions. Formal models that utilize mathematical notations can be difficult to create and understand primarily because of the effort required to relate abstract mathematical expressions to “real-world” concepts.

The exploratory nature of early requirements modeling activities often requires the flexibility (in terms of ease of change and tolerance towards incompleteness) that informal modeling notations can provide. Reliance on informally stated semantics is not necessarily a hindrance at this stage because the models are used primarily by modelers to gain an initial understanding of the problem. As the requirements modeling activity progresses, the informal models become less effective as a means for capturing and communicating required behavior. The lack of precision makes it difficult to carry out rigorous analyses that can lead to a better understanding of the problem, and that can uncover errors of omission, inconsistencies and ambiguities in the models. Lack of precision can also lead to the creation of ambiguous models, which, in turn, can lead to problematic communication.

In this paper we propose an approach to requirements modeling and analysis that allows developers to move from an informal modeling realm to a formal modeling realm. The approach allows a developer to (1) transform informal OOA class diagram (CD) and use case, expressed as a variant of the Fusion Operation Model [7], models to formal models, and to (2) rigorously analyze properties of these models. The technique is called *UMLtranZ* (UML model Transformation to Z) and it supports the transformation of UML Class Diagrams and to Z specifications [4, 31]. In UMLtranZ, Class Diagrams characterize the valid observable states of applications, and Use Cases define the net effect of system operations (application services). A valid application state is an object structure that has the properties specified in a Class Diagram.

Development of a requirements model using the UMLtranZ technique involves developing Class Diagrams and Operation Models using a variation of the guidelines given in the Fusion text [7], and then transforming them to Z specifications that can be analyzed using Z analysis tools such as ZANS [21] (a Z animator) and Z-EVES [9] (a Z theorem-checker/prover). Our experiences with applying UMLtranZ indicate that defects in the requirements are uncovered not only during analysis of the Z specifications, but also during the transformation of the OOA models to Z specifications.

In section 2 we discuss related works, and give an overview of Class Diagrams, Operation Models, and the Z notation used in UMLtranZ. In section 3 we describe the Class Diagram and Operation Model semantics underlying the UMLtranZ technique and introduce the example that is used in subsequent sections to illustrate the application of UMLtranZ. In section 4 we describe the

Class Diagram-to-Z transformation process, and in section 5 we describe the Operation Model-to-Z transformation. In section 6 we discuss how the Z specifications can be used to support rigorous analysis of OOA models, and in section 7 we give an overview of a prototype tool that we are building to support UMLtranZ. We conclude in section 8 with a summary of major results and an outline of our plans to further develop UMLtranZ.

2 Formalizing Object-Oriented Analysis Models

In UMLtranZ, requirements modeling is carried out initially in the informal realm using a UML-variant of the Fusion OOA process. The Class Diagrams and Operation Models are then “formalized” by transforming them to formal, analyzable models. Properties expressed in terms of the informal modeling notation are translated to properties expressed in the formal notation, and rigorous analysis of the formal models is carried out to determine the validity of the properties.

The UMLtranZ transformation technique is an example of a class of techniques that has been referred to as “integrated methods (techniques)” (e.g., see [3, 14, 18, 19]). In our work, “integration” means providing mechanisms that support the transition from informal to formal modeling realms. The technique that we propose in this paper provides a gateway between informal and formal modeling realms; as informal models and properties are passed through the gateway they are transformed to entities that can be manipulated in the formal realm. Moving from the formal to the informal modeling realm in UMLtranZ is currently restricted to formal models that have the form generated by the informal-to-formal transformation (not discussed in this paper).

In the remainder of this section we discuss other approaches to “formalizing” OO notations and give an overview of the notations used in UMLtranZ.

2.1 Related Works

We have identified three broad classes of approaches to formalizing OO modeling concepts: *supplemental notations*, *OO-extended formal notations*, and *integrated techniques/method*.

In the supplemental approach parts of the informal models (e.g., those expressed in natural language) are replaced by more precise constructs. A good example of the supplemental approach can be found in Syntropy [8]. In Syntropy, OMT-like models are annotated with mathematical expressions. The UML developers recently developed a notation called the *Object Constraint Language* [32] that can be used to precisely annotate UML models. Use of the UML with the OCL is another example of the supplemental approach. Using precise annotations one can precisely state constraints on model structures, but annotations do not often provide the full semantics of the constructs they are associated with. The complexity of, this single view of a system is increased because of the use of multiple notations, the informal and formal, for its representation.

In the OO-extended formal notations approach an existing formal notation is extended with OO features. Several OO extensions of formal notations have been proposed in the literature (e.g., Z++ [24] and Object-Z [11]). Often the intent of adding OO modeling concepts to formal notations is to enhance the structuring capabilities of the base formal language. In this respect these approaches have indeed resulted in richer formal notations. Furthermore, incorporating OO concepts into formal notations requires that the OO concepts be formalized. The result is a body of work on formal notions of object behavior and class structures. A drawback with this approach is that the semantics of the resulting language can become quite complex. The complexity can

make creating, analyzing, and understanding the models difficult. A further drawback with this approach is the limited tool support which is available.

In “integrated” informal/formal techniques, imprecise models are transformed to models expressed in a suitable formal modeling notation. A number of integrated OO and formal notations have been proposed (e.g., see [3, 14, 18]). Most works focus on the generation of formal specifications from variants of popular, but less formal OO models (e.g., OMT, Fusion, Shlaer-Mellor). Most works on integrated formal and OO modeling notations fail to state whether their techniques are applicable to requirements, high-level (architectural) or detailed design modeling. We have found it useful to distinguish between problem-oriented modeling (requirements modeling) and solution-oriented modeling (design). The concepts and supporting semantics can vary according to the perspective taken. For example, in a problem-oriented view the generalization/specialization relationship between classes can be cleanly modeled in terms of set/subset relationships. From a solution-oriented perspective the notion of inheritance includes the notion of generalization/specialization and more (e.g., use of inheritance to support reuse of implementation). The set/subset semantics that provides a clean semantics at the problem-oriented level becomes inappropriate when applied to inheritance. In this case a programming language-level type/subtype semantics may be more appropriate. For example, Bordeau and Cheng’s treatment of inheritance in their formalization of OMT Class Diagrams is more appropriate for solution-oriented models [3]. Our work differs from other works on integrated techniques in that we specifically developed a semantics based on a problem-oriented perspective of UML models.

A tool called *RoZeLink* (see www.calgary.shaw.wave.ca/headway/use.htm) developed by *Headway Software* provides support for linking Z specifications to Class Diagrams. The tool provides limited generation capabilities: For each class in a Class Diagram a Z schema is generated that declares the class attributes. The generated Z schemata are extended by the modeler to reflect constraints on attribute values and other class properties. Our techniques provides support for transforming Class Diagrams to Z schemata that reflect more semantic content. This is possible because our technique is based on a well-defined semantics for Class Diagrams. Human interaction is also needed in our approach, but this is restricted to the entry of application-specific constraints and to the simplification of generated schemata.

The interpretations that we use as the basis for the CD-to-Z transformation rules were developed from the informal descriptions of semantics provided in the OMG-UML V1.3 standard document [27] and from our previous work on generating formal specifications from Use Cases and Fusion models [14, 15]. In developing the interpretations for the UML Class Diagram constructs we tempered the informal descriptions given in the UML standard document (OMG-UML V1.3) with a problem-oriented view of OO concepts. Development of the problem-oriented interpretation of UML Class Diagrams required an in-depth analysis of the UML semantics document. This analysis resulted in the identification of inconsistent, ambiguous, and vague statements, as well as incomplete descriptions of semantic concepts. This is not surprising given the informal nature of the semantic description. This paper does not discuss the problems we uncovered in detail, rather it focuses on the interpretation we developed for requirements-level Class Diagrams and how it is used to transform Class Diagrams to formal models. In the presentation of the technique we relate our interpretation of UML constructs with the informal descriptions given in the OMG-UML V1.3 document and briefly discuss the rationale behind our interpretation. The reader can use these informal links to judge the adequacy of our interpretations.

Our approach also differs from other works on integrated techniques in that it uses an incremen-

tal approach to formalizing Class Diagrams. The formalization of complex, large CDs is broken down into manageable increments. Each phase builds upon the formal model produced in the previous phase by formalizing a more detailed view of the Class Diagram.

In our previous works (and other works on integrated formal/informal techniques), the focus was on producing formal specifications from informal models and little was said about how the formal models could be used to analyze the informal models. In this paper we demonstrate how one can use the Z specifications produced from a CD to establish properties about the CD.

2.2 UMLtranZ Object Analysis Models

UMLtranZ uses UML Class Diagrams and a variation of the Fusion Operation Models. The models are created using the Fusion OOA model development guidelines. Fusion is an object-oriented software development methodology that combines and extends some of the more mature OO techniques, for example, Rumbaugh's Object Modeling Technique (OMT) [29], Booch's technique [2], Wirfs-Brock's Class Responsibility Collaborator [33] (CRC) technique, and Jacobson's Objectory [20]. Fusion claims to take the best ideas from these methods and incorporate them into a single coherent method that covers analysis, design, and implementation. A soon to be released revised version of the Fusion methodology will be based on the OMG-UML V1.3 (see, <http://www.hpl.hp.com/fusion/index.htm>).

The UMLtranZ Class Diagrams are expressed in the UML [27] rather than the notation used in the Fusion book [7]. The UML is an Object Management Group (OMG) standard for OO modeling notations. It is expected that major OO modeling techniques and methods will utilize the UML notation.

In this subsection we give an overview of the Class Diagrams and Operation Models used in UMLtranZ.

2.2.1 UMLtranZ UML-based Class Diagrams

A *Class Diagram* is a conceptual model of a system that is expressed in terms of classes and the relationships among them. At the requirements level, Class Diagrams reflect problem-oriented concepts. At the design level, Class Diagrams reflect solution-oriented concepts. In this paper we discuss only requirements-level Class Diagrams. In the remainder of this paper a *requirements-level Class Diagram* is referred to as a CD. An example of a CD for a clinical laboratory system is given in Fig. 1 [26]. The example CD has been reproduced with slight modification, so as to highlight aspects of the CDTranZ formalization steps. In a CD 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 a CD. Associations are depicted as adorned lines between classes. The name of the association is placed at or near the center of the line. Associations are associated with cardinalities that restrict the number of class instances (objects) that can be linked to an object. A multiplicity is a set of non-overlapping natural number ranges. A range can have one of the following forms: $m..n$ (m to n), $n..n$ (also written as n), $0..*$ (0 or more; also written as $*$), $m..*$ (m or more). For example, in Fig. 1 the cardinalities on the association *result_of* state that a *Test* object is either not linked or is linked to one *TestResult* object, and a *TestResult* object is linked to exactly one *Test* object.

A special form of association called *aggregation* can be distinguished in CDs. In an aggregation structure there are two types of classes: The *aggregate* (whole) class and the *component* (part)

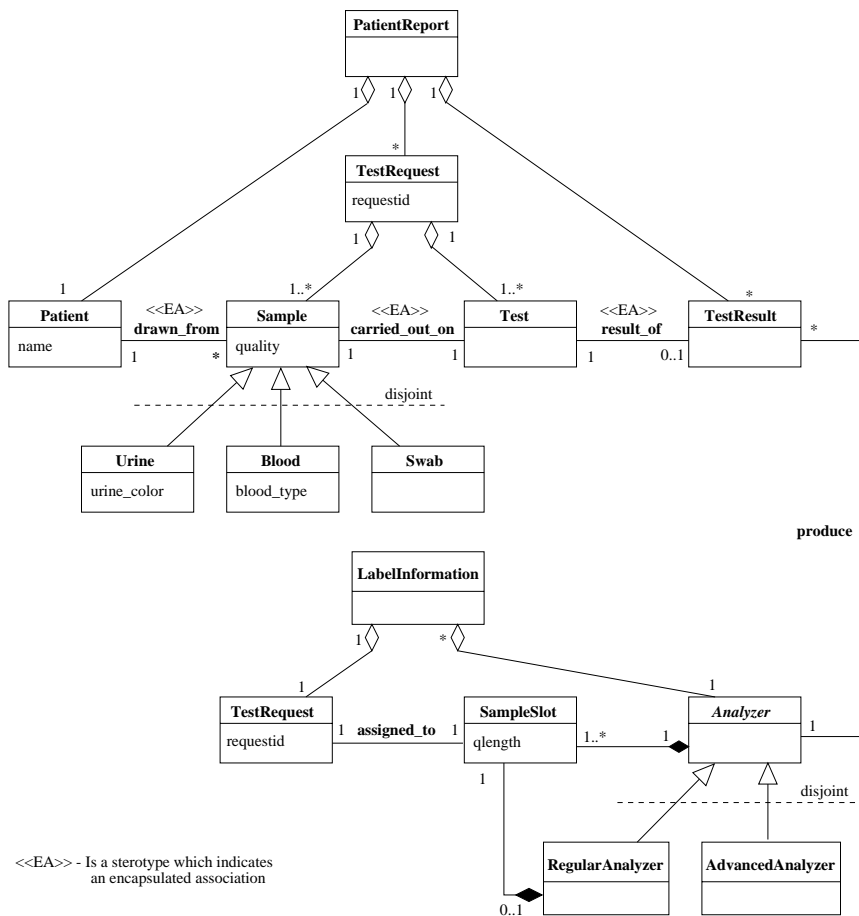


Figure 1: The Clinical Class Diagram

class. An aggregate object can be viewed as consisting of objects of the component classes. The UML defines two forms of aggregation: Strong and weak aggregation. Strong aggregation is called *composition*. In a composition, the components of an aggregate object can belong to at most one aggregate object. In a *weak aggregation*, the components can belong to zero or more aggregate objects.

Aggregation can also be used as a *structuring* mechanism that allows one to encapsulate binary associations. Associations whose links are encapsulated in such aggregation structures are called *encapsulated associations*. UML allows encapsulated associations in compositions, but it is not clear whether it is also allowed in weak aggregations. If a composition contains an encapsulated association between part classes *Comp1* and *Comp2*, then an object of the composition that contains *Comp1* and *Comp2* parts must include links between these parts (this is so because it is the links of the association which is encapsulated in the aggregation) and the links must be consistent with the cardinalities of the association. UMLtranZ allows encapsulated associations in both compositions and weak aggregations. Aggregations with encapsulated associations are called *encapsulating aggregations*. In our models, encapsulating aggregations are indicated by annotating

their encapsulated associations with the string `<< EA >>`.

In Fig. 1, a *PatientReport* object is an encapsulating weak aggregation with a single *Patient* object, zero or more *TestRequest* objects (which are also encapsulating aggregate objects), and zero or more *TestResult* objects. From our understanding of the requirements for the CD in Fig. 1 this encapsulating aggregation *should* be modeled as an UML composition, i.e. *strong* aggregation but we retain the original specification given in [26]. If a *PatientReport* object is composed of a *TestRequest* object then the *Sample* component must be linked to the *Patient* component of *PatientReport*. Similarly, if a *PatientReport* object has *TestRequest* and *TestResult* components then each *TestResult* component must be linked to a *Test* component of a *TestRequest* object in the *PatientReport* object, and vice versa. *Sample* and *Test* objects in a *TestRequest* object must be linked, and the links must respect the many-to-many multiplicity of the *carried_out_on* association.

Generalization/Specialization relationships are denoted by arrows emanating from specializations and directed towards their generalizations. The *Sample* class is a generalization of disjoint *Urine*, *Blood* and *Swab* classes in Fig. 1.

Additional constraints on class structure can be expressed as textual annotations in a CD. The UML developers have introduced a language, the OCL (*Object Constraint Language*[27]), for expressing such notations. The current version of UMLtranZ does not utilize OCL expressions in the transformations, but will in the future.

2.2.2 Operation Models

A Fusion *Operation Model* (OM) characterizes the observable effects of system operations. A system operation is one that can be invoked by users (external agents) of the system. This description of Fusion OM is analogous to that of the Use Case model given in [1] and we take the view of congruency between the two descriptions, with the note that the the OM is more structured and provides more information than the basic use case. Each system operation is described by a Fusion *Operation Schema* (OS) that consists of the following sections:

Description: an informal and concise description of the operation.

Reads: a list of the items that the operation reads.

Changes: a list of the items the operation changes.

Sends: a list of the events the operation sends to other objects in the environment.

Assumes: a condition that describes what is assumed true at the start of the operation.

Result: describes what is true after the operation has completed its execution.

We modify the Fusion OS in order to more reflect the terminology of our UMLtranZ process as follows:

Description: an informal and concise description of the operation.

Inputs: a list of the items that the operation accesses.

Modifies: a list of the items the operation *may* change.

Outputs: a list of the items the operation sends to other objects in the environment.

Pre-conditions: a condition that describes what is true at the start of the operation.

Post-conditions: a condition that describes what is true at the end of the operation.

2.3 The Z notation

In this section we introduce only the parts of the Z notation necessary to understand the specifications given in this paper (see [31] for more details).

The primary structuring construct in Z is the *schema*. A schema has two parts: a declaration and a predicate part. The *declaration* part consists of variable declarations of the form $w : Type$, where w is a variable name and *Type* is a type name. The preceding declaration means that the value of w is a member of the set named by *Type* (types are sets in Z). The *predicate* part consists of a predicate that defines the relationships among the declared variables. A schema can be written as follows:

<i>Schema</i>
<i>Declaration</i>
<i>Predicate</i>

Types in Z can be either *basic* or *composite*. Elements of basic types (or *given sets*) are used as the basic building blocks for more complex elements (elements with composite types). A basic type is declared as follows:

[*BASIC_TYPE_NAME*]

The internal structure of the elements of basic types are abstracted out (i.e., they are not of interest). There are three kinds of composite types in Z: *set types*, *cartesian product types*, and *schema types*. Example declarations involving composite types are given in the schema below:

<i>Decls</i>
$s : \mathbb{P} S$
$t : A \times B$
$u : Schema$

s is a *set of elements* from S.

t is a *pair of elements* in which the first element of the pair is an element of A and the second is an element of B .

u is a *binding of values* to the variables declared in the schema *Schema*.

Using these types one can also define functions and relations as sets of pairs in which the first element is from the domain and the second element is from the range of the function/relation.

Z schemata are used to model both the structural and dynamic aspects of a system. A schema that captures the structural aspect of a system will be referred to as a *state schema*, and a schema that captures the dynamic aspect will be referred to as an *operation schema*. In a state schema the components of a system's state are declared in the declaration section and the constraints on the state are given in the predicate part. An example of a state schema is given below:

[*SUBS, TELEPHONES*]

<i>TelService</i> <i>subs</i> : $\mathbb{P} SUBS$ <i>directory</i> : $SUBS \rightarrow TELEPHONES$
$\text{dom } directory \subseteq subs$

SUBS and *TELEPHONES* are the sets of registered subscribers and registered telephones, respectively. *TelService* describes the state of a telephone system consisting of a set of subscribers, *subs*, and a set of pairs, *directory*, modeled as a partial function to reflect the constraint that a subscriber can only be assigned to a single telephone. The predicate in the predicate part states that only subscribers can be associated with a telephone in this system (i.e., the domain of *directory* is a subset of *subs*).

An operation schema defines the relationship between the state at the start and at the end of an operation's execution. The declaration part of an operation schema declares variables representing the before and after states, inputs, outputs, and any other variables needed to define the relationship. The predicate part of the schema defines the relationship between the before and after states. The following conventions are used for variable names in operation schemata:

- unprimed variable (e.g., *w*) - value of variable before operation execution;
 - primed variable (e.g., *w'*) - value of variable after operation execution;
 - variable ending in '?' - an input to the operation; and
 - variable ending in '!' - an output from the operation.
- ΔS denotes a possible change in state *S*.
 $\exists S$ denotes that the state *S* does not change.

An example of an operation schema is given below:

<i>AddSub</i> $\Delta TelService$ <i>sub?</i> : <i>SUBS</i>
<i>sub?</i> $\notin subs$ <i>subs'</i> = $subs \cup \{sub?\}$ <i>directory'</i> = <i>directory</i>

This operation schema defines an operation that adds a new subscriber to the telephone system whose state is defined by *TelService*. The predicate part states that the input subscriber (*sub?*) is not currently a registered subscriber, and, after the operation is completed, the set of registered subscribers includes the new subscriber, and the directory set is left unchanged.

3 UMLtranZ: An Overview

In this section we give an overview of UMLtranZ and describe the semantics underlying the OOA models used in UMLtranZ. We also present the example that will be used to illustrate the application of UMLtranZ.

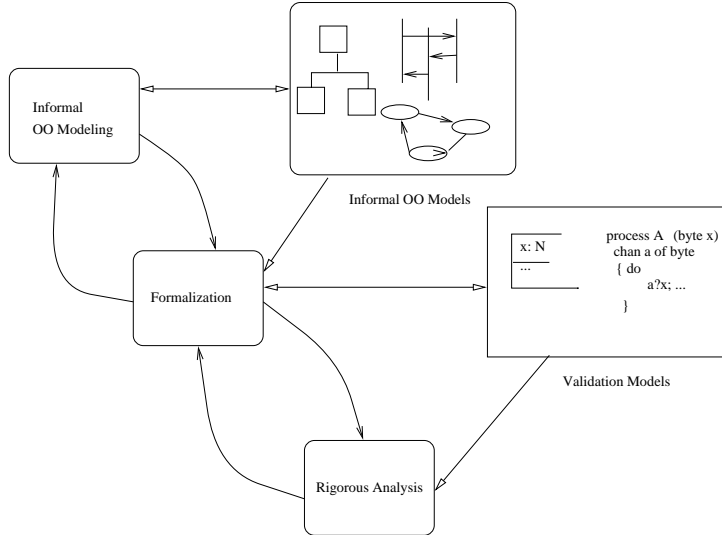


Figure 2: The Explore/Elaborate/Analyze Process Model

3.1 The UMLtranZ Transformation Process

UML and a variation of Fusion modeling techniques are used in UMLtranZ because they incorporate some of the best OO modeling experiences. The use of Z reflects our desire to use a stable, mathematically precise, structured notation that can express OOA modeling concepts in a straightforward manner, and that is supported by computer-based analysis tools. Hall has shown that Z can be used to express OO modeling concepts [18], and Z is supported by typecheckers (e.g., ZTC [22]), animators (e.g., ZANS [21]), and theorem proving/checking environments (e.g., Z/EVES [9]). Fig. 2 illustrates the *Explore/Elaborate/Analyze* (EEA) process model that supports the application of the UMLtranZ technique. The Exploratory phase (Informal OO Modeling) is concerned with imposing an initial structure on the problem space that is being explored. During this phase the modeler is developing an understanding of the problem and its context through “mapping” of the problem space. The use of intuitively-appealing informal techniques is favored in this phase because they provide the needed flexibility (in terms of ease of change and tolerance towards incompleteness) and ease of use. In UMLtranZ, the primary models (maps) that are produced in this phase are CDs and Operation Models. Often the development of these models from vague statements of requirements requires use of intermediary modeling techniques. We advocate the use of *Use Cases* as intermediary models [1, 7].

The Elaboration phase (Formalization) is concerned with making the models developed in the Exploratory phase more precise. The precision enables rigorous validation of properties, thus the formal models produced in this phase are referred to as *validation models*. UMLtranZ provides rules and guidelines for transforming CDs and Operation Models to validation models expressed in the Z notation. While some of the transformations that take place in the formalization activity can be automated, there are others that require human interaction, for example, tasks that involve transforming vague statements of requirements to precise expressions, and adding information needed to obtain precise statements. These tasks provide opportunities for uncovering ambiguities, omissions,

and inconsistencies in the informal models. The insights gained during the formalization activity as a result of human interaction can lead to a deeper understanding of the requirements, that, in turn, can lead to the modeler improving the informal models (e.g., by restating vague statements more precisely in natural language, or adding information to resolve ambiguities). For this reason, complete automation of the formalization activity is not a goal of our work.

The Analyze phase is concerned with rigorously analyzing the formal models produced in the Elaboration phase. The rigorous analysis activity involves discharging proof obligations for the formal models (e.g., checking that the initial state satisfies the state invariant and checking that the preconditions/postconditions of operations determine valid states), and proving the presence of properties. UMLtranZ supports the transformation of properties expressed in terms of the informal OOA models to properties expressed in the formal notation. UMLtranZ also supports the use of Z typecheckers, and the Z animator ZANS. Currently, proofs are done by hand, but we will be integrating the Z theorem prover/checker, Z/EVES, into a prototype analysis environment that is under development.

The formal models produced by the formalization activity are intended to provide a precise specification of functional requirements that are imprecisely stated in the informal models. If the informal models are created without knowledge of the semantics imposed by the formalization activity then there is the danger that the formal models produced by the formalization activity will not reflect the interpretation intended by the model's creator. If this is the case, then there is a strong possibility that faults uncovered during the formalization and analysis activities are more apparent than real. To avoid this, the informal models should be created with knowledge of the semantics imposed on the informal notations by the transformation process.

3.2 Interpreting OOA Models in UMLtranZ

In UMLtranZ, a CD is a characterization of valid, externally observable application states. An (externally observable) *application state* is a structure consisting of all the objects and links that can be observed by external agents at some point in time. A *formalized CD* expresses an invariant property that valid application states must have. The Z specification produced from an informal CD in UMLtranZ is a representation of the formalized CD. In this paper we refer to the valid application states that possess the properties expressed in a CD as *configurations*. The semantic domain for CDs is a collection of sets of configurations, and the meaning of a CD is a set of configurations.

Fig. 3 illustrates the relationship between a CD and its meaning using a simple case. The CD A_Rel_B has as its meaning the shown set of configurations (an element of the CD semantic domain). The configurations in the set (e.g., *config1*, *config2*, *config3*) must satisfy the constraints depicted in the CD. In this case the constraints are expressed as cardinalities on the association *Rel*. The multiplicity constraints state that each instance of *A* in a configuration must be linked to exactly one instance of *B* in the configuration, and an instance of *B* in the configuration can be linked to 0 or more instances of *A* in the configuration. For example, in *config1* the two instances of *A*, *a1*, *a2* are linked to the instance of *B*, *b*, via the links (instances of associations) *rel1*, *rel2*.

The instance-based semantics of CDs we use is consistent with the UML object interpretation of a Class Diagram. The OMG-UML V1.3 document states (extracted from [27], page 2-59, Instantiation):

The purpose of a model is to describe the possible states of a system and their behavior.

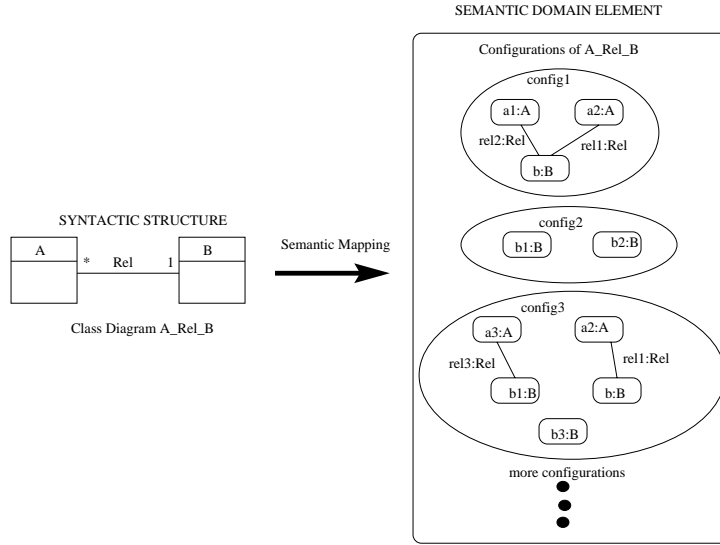


Figure 3: An Example of a Class Diagram Interpretation

The state of a system comprises objects, values, and links. ... The state of a system is a valid system instance if every instance in it is a direct instance of some element in the system model and if all of the constraints imposed by the model are satisfied by the instances.

In the UMLtranZ context, references to “model” and “system model” in the above quote refer to CDs, and a “valid system instance” is a configuration.

A formalized Fusion Operation Schema (OS) characterizes a 4-tuple configuration set $\langle input, output, pre-condition, post-condition \rangle$. Each 4-tuple in such a set denotes the before- and after-effect of the operation on the application state, the necessary inputs and the resulting outputs of the operation. In UMLtranZ, a formalized OS is expressed as a Z operation schema. This is consistent with the intended use of Operation Models in Fusion as described in the Fusion book (extracted from [7], page 269, C-3.2 Operation Model)

The operation model is a precondition/postcondition style specification, as seen in Z or VDM.

The UMLtranZ interpretation is also consistent with the UML interpretation of behavior which is stated as follows (extracted from [27], page 2-59, Instantiation)

The behavioral parts of UML describe the valid sequences of valid system instances that may occur as a result of ... external ... behavioral effects.

In UMLtranZ a transformation is referred to as a *formalization* of the informal model. Invariably, the informal model does not have all the information needed to generate a formal expression of desired properties, or the information is expressed in notation that is open to multiple interpretations. Human intervention is needed to provide additional information and to provide formal expressions of informal statements. The intervention forces the modeler to examine the informal models more closely and helps him/her formulate appropriate questions about its content.

3.3 The Clinical Laboratory System

We illustrate the UMLtranZ technique by applying it to the CD of a Clinical Laboratory System based on the model given in [26]. Below is a statement of the problem with respect to the portion of the CD we formalize.

In the Clinical Laboratory System tests are carried out on patient samples and analyzed using analyzers. Samples can be drawn only from patients, and can be either blood, swab or urine. The results of tests are recorded in the patient's report. A label holds information about the requested test and the analyzer assigned to carry out the test. The Clinical System CD is shown in Fig. 1.

A full formalization of the CD is given in the Appendix.

4 Formalizing Requirements-Level Class Diagrams

CD constructs can possess two types of properties: *static* and *dynamic*. Static properties are used to characterize configurations, while dynamic properties are used to constrain application behavior. An example of a static property is the multiplicity of a class. If a class has a multiplicity property, $m..p, m \leq p$, then in any valid state (configuration) there can be no less than m and no more than p objects of the class. An example of a dynamic property is the notion of *addonly* attributes. An addonly attribute is one that can hold more than one value, but once a value is added it cannot be removed. In UMLtranZ, dynamic properties of CD constructs are used to constrain the effects of operations and will be discussed in the next section. The Z specification characterizing configurations of a CD reflects the static properties of the CD. This section focuses on the static properties of CDs.

The formalization of a CD is done in three phases:

- *Formalization of basic classes:* A *basic class* is one that is not a specialization (subclass) in a generalization hierarchy. This phase produces a Z schema for each basic class, called the *instance schema*, that defines the properties that each object of the class must possess when it is present in a configuration. The set of Z specifications produced in this phase is referred to as the *Basic Class Model* (BM).
- *Formalization of generalization/specialization structures:* In this phase, the BM is used to create schemata that formalize generalization structures in the CD. In a generalization structure *superclasses* are specialized by *subclasses*. At the requirements level, a subclass inherits the attributes of its superclass and may define additional attributes. For each subclass, an instance schema declaring the subclass as a specialization of the superclass is created. For each generalization structure a Z schema called a *gen-schema* is created. A gen-schema expresses relationships among subclasses and superclasses. For example, properties such as 'subclasses are disjoint' and 'superclasses are abstract' are expressed in the gen-schema. The BM and the gen-schemata are collectively referred to as the *Specialization Model* (SM).
- *Formalization of associations:* In this phase, the SM is used to create formalizations of associations (including aggregations). The Z specifications characterizing associations are combined with the specifications in the SM to produce a Z specification called the *configuration schema*. A configuration schema is a representation of the *formalized* CD.

A problem with the CD-to-Z transformation process is that it can sometimes produce lengthy Z specifications. Properties that are concisely and clearly expressed in natural language (e.g., “the association is many to many”) are sometimes transformed to lengthy formal statements. To tackle this problem we developed a set of parameterized Z definitional constructs, we call *Z generators*, that can be used to generate formal expressions of CD properties. An instantiated Z generator can be viewed as an abbreviated form of a Z expression. Z generators will be defined when they are used for the first time in this paper.

4.1 Naming convention

In the following subsections we give the UML-to-Z mapping rules used in each phase of the CD transformation process. The parts of the Z specifications that must be supplied by the user during the transformation are distinguished from those parts that can be automatically generated:

- User-supplied parts are written in **typewriter style**, and
- Parts that can be produced automatically are written *emphasized style*.

4.2 Phase 1: Formalizing Basic Classes

In the current UML standard (OMG-UML V1.3) a class is described as follows (OMG-UML V1.3, pg. 2-26):

A class is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics.

The attributes, relationships and semantics of a class are referred to as *object properties* in UMLtranZ. Operations and methods (i.e., operation implementations) are not associated with classes in an analysis phase CD. We take the Fusion view that allocation of operations to classes is a design activity [7].

The set of all possible objects belonging to a class forms the object space of that class. The set of objects (in the object space of a class) which are present in a configuration of the class is called the *configuration set* of the class, and the objects in the set are referred to as *configuration objects* of the class. In the context of a UMLtranZ CD a class defines its configuration objects (a subset of its object space). The UMLtranZ interpretation of a class implies that constraints associated with basic classes in a CD are constraints on their configuration sets. For example, class multiplicity restricts the number of objects that can be in a configuration set of the class. This is consistent with the use of class constraints that we have encountered in UML and other OO models.

In the Clinical System CD (see Fig. 1) the basic classes are *PatientReport*, *TestRequest*, *Patient*, *Sample*, *Test*, *TestResult*, *LabelInformation*, *SampleSlot*, and *Analyzer*.

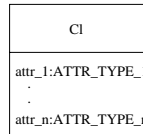
The UMLtranZ transformation maps a basic class to a Z specification that characterizes the configuration set of the class. The Z specification is called an *instance schema*. The characterization can include constraints on the number of instances that can appear in a configuration (multiplicity constraints), and define relationships that must be maintained among attribute values within and across configuration objects of the class.

The configuration set characterized by an instance schema must be a subset of the class’s *object space*, where the object space of a class is defined as the set of all possible objects of the class.

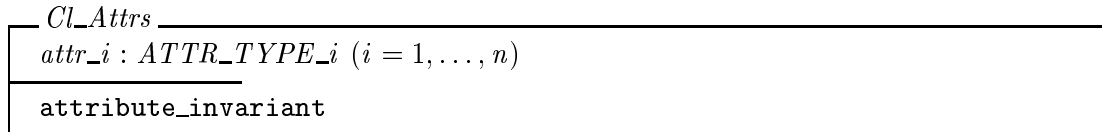
The object space of a class is represented by a Z basic type in the formalization of the class. The following basic types represent the object spaces of the basic classes in the Clinical System:

[*PATIENTREPORT*, *TESTREQUEST*, *PATIENT*, *SAMPLE*, *TEST*]
 [*TESTRESULT*, *LABELINFO*, *SAMPLESLOT*, *ANALYZER*]

If a basic class has attributes, then an *attribute schema* is created for the class. In UMLtranZ, all class attributes are object-scope. An attribute schema defines the attribute values that can be associated with objects of a class. The attribute schema for a basic class, *Cl*,



with n attributes, $attr_i$ ($i = 1, \dots, n$), and an optional, user-supplied invariant that constrains attribute values, **attribute_invariant**, is given below (the indexing used in the Z schemata in this paper is not part of the Z notation; it is used to make the presentation of schemata concise):



In the above schema, n variables, ($attr_1, \dots, attr_n$), are declared. The names $ATTR_TYPE_i$ ($i = 1, \dots, n$) are attribute type names. Attribute types are declared as Z basic types if the built-in Z types are not appropriate. The formalization requires a type to be associated with each attribute; if one is not given in the CD then the capitalized attribute name is used as the type and declared as a Z basic type.

In OMG-UML V1.3 an attribute can also have a multiplicity. The multiplicity indicates the number of distinct values the variable can store at any point in time. For example, an attribute declared as $list[1..5] : Integer$ is interpreted as a variable that can hold one to five integers at any point in time. Attributes that can store more than one value are called *containers* and they are formally interpreted as sequences in UMLtranZ. The *list* attribute given above is represented in an attribute schema as follows:



The following basic types denote attribute type spaces for the Clinical System (the attribute *qlength* is of type integer, which is predefined in Z and thus no explicit type declaration for *qlength* is made):

[*NAME*, *QUALITY*, *COLOR*, *TYPE*]

The attribute schema for the basic class *Sample* in the Clinical System CD is given below:

$\begin{array}{l} \textit{Sample_Attrs} \\ \textit{sample_quality} : \textit{QUALITY} \end{array}$
--

If a class has no attributes then an attribute schema is not created for it.

In general, an object is mapped to its property values via relations in UMLtranZ. This is consistent with the following statement in OMG-UML V1.3 (pg. 2-26):

Each Object instantiated from a class contains its own set of values corresponding to the Structural Features declared in the full descriptor (of the class).

The phrase in parentheses is added by the authors. In UMLtranZ, an instance schema includes the definition of a mapping between configuration objects and their attributes (if any). The following is the rule for generating an instance schema from a basic class:

Basic Instance Schema (BIS) Rule

A basic class, Cl , associated with

- an optional multiplicity $m..p$, where m is a natural number and p is a natural numbers or '*',
- an optional attribute schema Cl_Attrs ,
- an optional, user-supplied invariant on configuration sets of the class, $objs_invariant$,

is transformed to the following Z specification

$[CL]$ [Object space of Cl]

Instance Schema

$\begin{array}{l} \textit{Cl} \\ \textit{cl} : \mathbb{P} \textit{CL} \text{ [configuration set of Cl]} \\ \textit{cl_attrs} : \textit{CL} \leftrightarrow \textit{Cl_Attrs} \text{ [maps config. objects to attributes]} \end{array}$
$\begin{array}{l} m \leq \#\textit{cl} \leq p \\ \text{dom } \textit{cl_attrs} = \textit{cl} \\ \textit{objs_invariant} \end{array}$

The variable cl in the Cl schema given in the BIS rule is called a *configuration set variable*.

Following are formalizations of the basic classes *Analyzer* and *Sample*:

$\begin{array}{l} \textit{Analyzer} \\ \textit{analyzer} : \mathbb{P} \textit{ANALYZER} \end{array}$
--

$\begin{array}{l} \textit{Sample} \\ \textit{sample} : \mathbb{P} \textit{SAMPLE} \\ \textit{sample_attribute} : \textit{SAMPLE} \leftrightarrow \textit{Sample_Attrs} \\ \text{dom } \textit{sample_attribute} = \textit{sample} \end{array}$

4.3 Phase 2: Formalizing Specializations

The type space of a specialization hierarchy can be viewed as a carving up of the root superclass object space into subsets, where each subset is the object space of a subclass. In a configuration, the subclasses of a superclass are subsets of the superclass configuration set. How a superclass configuration set is divided into subclass configuration sets can be constrained as follows:

- *Overlapping and Disjoint Subclasses:* A set of subclasses is said to be *disjoint* if there are no objects that are instances of more than one subclass in the set. Whereas the set is said to consist of *overlapping* subclasses if an object maybe simultaneously an instance of more than one subclass (OMG-UML V1.3 pg. 2-36).
- *Abstract and Non-Abstract Superclasses:* An *abstract* superclass is one in which each superclass configuration object is also a configuration object of at least one depicted subclass. A superclass that can have configuration objects that are not configuration objects of any depicted subclass is said to be *non-abstract*.

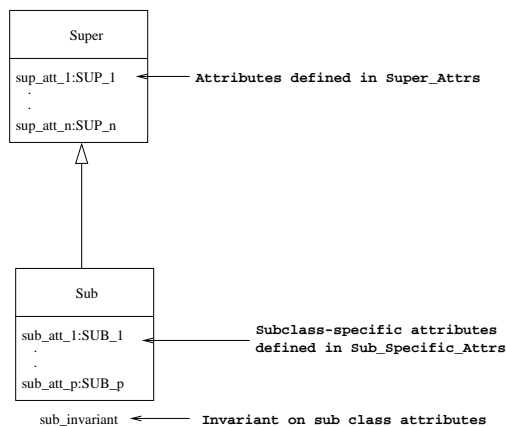
In UMLtranZ, the configuration set of a subclass in an application state is a subset of the configuration set of its superclass.

One also has to assign a meaning to a specialization structure that reflects the generalization/specialization relationships between the classes and the constraints on the relationships. Given a configuration, the set of configuration sets for all classes in a specialization structure is called the *specialization configuration* of the structure. In UMLtranZ, a specialization structure is a characterization of its specialization configurations.

The formalization of a specialization structure is done in two steps. In the first step the instance schemata for the subclasses are generated. In the second step a Z schema called a *gen-schema* is developed for each specialization structure. A gen-schema characterizes the specialization configurations of the structure.

4.3.1 Generating subclass instance schemata

If a superclass is associated with an attribute schema then all its subclasses must have attribute schemata that include the attribute schema of the superclass. The attribute schema of a subclass includes the superclass attribute schema, a schema defining the subclass-specific attributes (if any), and an optional, user-supplied invariant that defines the relationship that must be maintained across superclass and subclass-specific attribute values. For example, a subclass, *Sub*, of a superclass with an attribute schema *Super_Attrs*, and with subclass-specific attributes defined in the schema, *Sub_Specific_Attrs*,



has an attribute schema of the form:

<i>Sub_Attrs</i>
<i>Super_Attrs</i>
<i>Sub_Specific_Attrs</i>
sub_invariant

The (optional) user-supplied invariant **sub_invariant** expresses relationships that must be maintained between superclass attributes and subclass-specific attributes. As an example, the attribute schema for the *Blood* subclass in the Clinical System CD is given below:

<i>Blood_Attrs</i>
<i>Sample_Attrs</i>
<i>blood_type</i> : TYPE

A subclass instance schema consists of a configuration set variable, a variable representing the object space of the subclass (a subset of the superclass object space), an optional function mapping configuration objects to their attributes, and an optional, user-supplied invariant that must be satisfied by configuration sets of the subclass. The rule for generating a subclass instance schema follows.¹

¹In the rule we use the following shorthand for a conjunction of predicates:

$$\bigwedge_{i=1, \dots, p} P_i$$

where P_i ($i = 1, \dots, p$) are predicates. The above expands to:

$$P_1 \wedge P_2 \wedge \dots \wedge P_p$$

The Subclass Instance Schema (SIS) Rule

Let

- Sub be a subclass of a superclass characterized by an instance schema $Super$,
- $supers$ be the configuration set variable defined in $Super$,
- Sub_Attrs be the optional attribute schema for Sub ,
- $ROOT$ be the object space of the root superclass,
- $super_attr_i$, ($i = 1, \dots, p$) be superclass attributes defined in the superclass attribute schema (included in the schema $Super$), and
- **sub_objs_invariant** be the optional, user-supplied invariant that must be satisfied by the subclass configuration sets.

The instance schema for the subclass Sub is given below (see Fig. 4 for an illustration of the above subclass structure):

Sub <hr/> $Super$ [inheritance of superclass properties] $subs, SUB : \mathbb{P} ROOT$ [subclass config. set and object space] $sub_attrs : ROOT \rightarrow Sub_Attrs$
<hr/> $subs = \{x : supers \mid x \in SUB\}$ [a] $\forall s : subs \bullet (\bigwedge_{i=1, \dots, p} sub_attrs(s).super_attr_i = super_attrs(s).super_attr_i)$ [b] $dom\ sub_attrs = subs$ [c] sub_objs_invariant [d]

Conjunct [a] states that the configuration set of the subclass is precisely those objects in the configuration set of $Super$, $supers$, that are in the object space of the subclass (SUB). Conjunct [b] states that the subclass inherits the attribute values of its superclass. Conjunct [c] states that the mapping sub_attrs maps only configuration objects of Sub to their attribute values, and conjunct [d] is the optional invariant on the configuration sets of Sub .

The instance schema produced by application of the SIS rule to the subclass $Blood$ is given below:

Blood

Sample
 $blood, BLOOD : \mathbb{P} SAMPLE$
 $blood_attribute : SAMPLE \rightarrow Blood_Attrs$

$blood = \{b : sample \mid b \in BLOOD\}$
 $\forall b : blood \bullet$
 $(blood_attribute\ b).sample_quality$
 $= (sample_attribute\ b).sample_quality$
 $dom\ blood_attribute = blood$

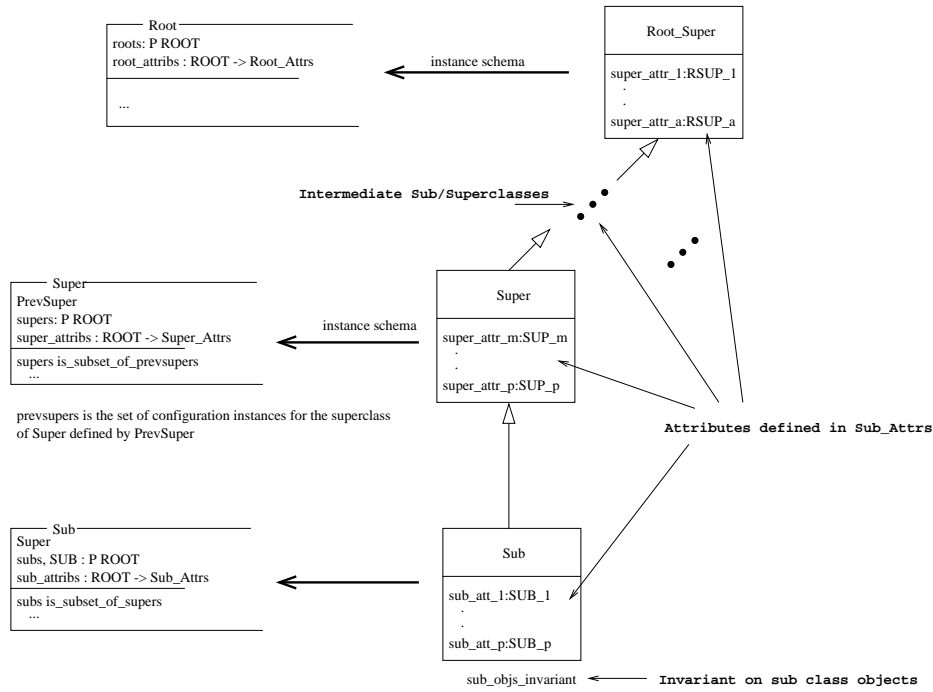


Figure 4: CD for SIS Rule

4.3.2 Characterizing Specialization Configurations

A gen-schema characterizes the specialization configurations of a specialization structure, that is, it defines the relationships between the root superclass and subclass configuration sets. The disjoint property of subclass object spaces and the abstract property of superclass configuration sets are expressed in the gen-schema of a specialization structure. The following are Z expressions of the disjoint and abstract properties:

- **Disjoint Property (on object spaces):** Subclass object spaces, $SUBS_1, \dots, SUBS_n$ are disjoint (non-overlapping):

$$\text{disjoint}\langle SUBS_1, \dots, SUBS_n \rangle$$

It is also possible to specify the Disjoint Property only on subclass configuration sets (allowing object spaces to overlap). We suspect that the need for this type of constraint does not occur often. In UMLtranZ we interpret the presence of the disjoint annotation on subclasses as a constraint on the object spaces. If the disjoint property holds for configuration sets only and not for the object spaces, then the UML-provided disjoint annotation is not used. Instead, an annotation stating that the configuration sets are disjoint must be created by the modeler.

- **Abstract Property:** A superclass configuration set *supers* is abstract with respect to its subclass configuration sets $subs_1, \dots, subs_n$:

$$\bigcup \{subs_1, \dots, subs_n\} = supers$$

The set of instance schemata for the leaf subclasses of a specialization hierarchy include all the instance schemata for the classes in the hierarchy. This means that the gen-schema can be defined in terms of elements in the instance schemata of leaf subclasses. The following rule is used to produce a gen-schema for a specialization hierarchy:

Gen-Schema (GS) Rule

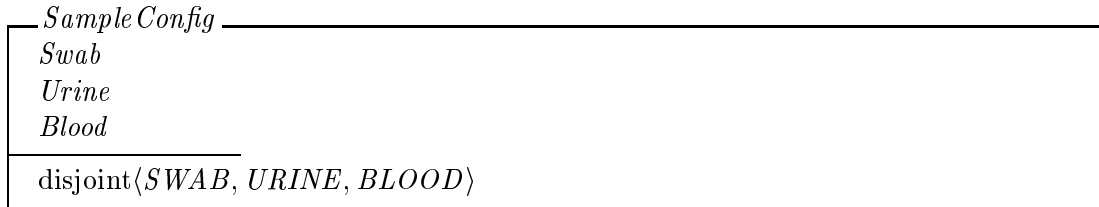
For a specialization structure with p subclasses, Sub_i ($i = 1, \dots, p$), let

- *ROOT* be the object space for the root superclass,
- SUB_i , ($i = 1, \dots, p$) represent the object spaces of the subclasses Sub_i , ($i = 1, \dots, p$),
- $Subleaf_i$, ($i = 1, \dots, n$; $n \leq p$) be the leaf subclass instance schemata,
- *root* be the configuration set variable defined in the instance schema of the root superclass,
- $subs_i$, ($i = 1, \dots, p$), be configuration set variables defined in the instance schemata of the subclasses Sub_i , ($i = 1, \dots, p$),
- $gen_invariant(SUB_1, \dots, SUB_p, root, subs_1, \dots, subs_p)$ be the optional predicate expressing disjoint properties of subclass object spaces or abstract super class properties,
- **other_invariant** other properties which are not defined automatically in the specialization structure.

The gen-schema for the above structure is given below:

$GenConfig$
$Subleaf_i$ ($i = 1, \dots, n$)
$gen_invariant(SUB_1, \dots, SUB_p, root, subs_1, \dots, subs_p)$
other_invariants

Application of the GS Rule to the specialization structure rooted by the *Sample* class in the Clinical System CD results in the following gen-schema (the class *Sample* is not abstract):



4.4 Phase 3: Formalizing Associations

The UMLtranZ formalization of associations is currently restricted to binary associations. This is not a serious limitation because n-ary (for n greater than 2) associations can be reduced to binary associations using encapsulating aggregations [7]. UMLtranZ also does not currently support the formalization of so-called association classes. Again, encapsulating aggregations can be used to model such situations in most cases.

In the context of a configuration, a binary association is interpreted as a mathematical relation between the configuration sets of the related classes that satisfies the multiplicity and other stated constraints on the association. The mathematical relations characterized by an association are defined by a Z schema called an *association schema*.

Two types of associations are formalized in this phase of the UMLtranZ CD formalization process: general binary associations and aggregations. In both cases, the formalization of associations that appear at both superclass and subclass levels of a specialization structure requires special treatment because of the need to relate the associations. An example of such an association can be found in the Clinical System CD (see Fig. 1): The aggregation between *Analyzer* and *SampleSlot* also appears at the subclass level as an aggregation between *RegularAnalyzer* and *SampleSlot*. In the UMLtranZ interpretation of aggregation in the context of specialization structures, these two aggregations are not independent: The aggregation at the subclass level must be a subset of the aggregation at the superclass level. The aggregation at the subclass level is said to be a specialization of the aggregation at the superclass level.

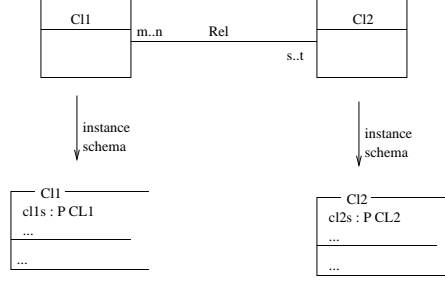
4.4.1 Formalizing general (non-specialized) binary associations

We created a parameterized Z generator (see ref:FMOODS96), called *Assoc-Gen*, that, when properly instantiated, characterizes a set of mathematical relations that are constrained by the cardinalities depicted on an association. Assoc-Gen has the following form:

$$p \longleftrightarrow q[source, target]$$

In the above, p , q , $source$ and $target$ are parameters, where p and q are cardinalities (i.e., sets of natural number ranges), $source$ denotes the source configuration set and $target$ denotes the target configuration set. Assoc-Gen can only be used in a context in which the types of $source$ and $target$ can be determined, that is, Assoc-Gen can be used in the predicate part of a Z schema in which $source$ and $target$ are declared variables, or are well-defined expressions. The choice of which class in a binary association is the source and which is the target is arbitrary at the requirements level.

Consider the association *Rel* shown in the diagram below,



where $n \geq m \geq 0$, $t \geq s \geq 0$. The set of mathematical relations defined by the association Rel is given by the following instantiated Assoc-Gen generator:

$$\begin{aligned}
& (m..n) \longleftrightarrow (s..t)[cl1s, cl2s] \equiv \\
& \{ Rel : CL1 \leftrightarrow CL2 \mid \text{dom } Rel \subseteq cl1s \wedge \text{ran } Rel \subseteq cl2s \\
& \wedge (\forall x : cl2s \bullet m \leq \#(Rel \sim (\{x\} \mid)) \leq n) \\
& \wedge (\forall x : cl1s \bullet s \leq \#(Rel(\{x\} \mid)) \leq t) \}
\end{aligned}$$

If $n = *$, that is, the target objects can be linked to an unbounded number of source objects, but no less than m source objects, then the third conjunct in the body of the set comprehension is replaced by:

$$(\forall x : cl2s \bullet m \leq \#(Rel \sim (\{x\} \mid)))$$

Similarly, if $t = *$ then the fourth conjunct is replaced by:

$$(\forall x : cl1s \bullet s \leq \#(Rel(\{x\} \mid)))$$

In general, the instantiated generator:

$$\{(a_1..b_1), \dots, (a_n..b_n), (p_1..*), \dots, (p_m..*)\} \longleftrightarrow \{(s_1..t_1), \dots, (s_k..t_k), (q_1..*), \dots, (q_j..*)\}[cl1s, cl2s]$$

expands to the following Z expression:

$$\begin{aligned}
& \{ Rel : CL1 \leftrightarrow CL2 \mid \text{dom } Rel \subseteq cl1s \wedge \text{ran } Rel \subseteq cl2s \wedge \\
& (\forall x : cl2s \bullet (\bigvee_{i=1, \dots, n} (a_i \leq \#(Rel \sim (\{x\} \mid)) \leq b_i)) \vee \\
& (\bigvee_{i=1, \dots, m} (p_i \leq \#(Rel \sim (\{x\} \mid)))) \wedge \\
& (\forall x : cl1s \bullet (\bigvee_{i=1, \dots, k} (s_i \leq \#(Rel(\{x\} \mid)) \leq t_i)) \vee \\
& (\bigvee_{i=1, \dots, j} (q_i \leq \#(Rel(\{x\} \mid)))) \}
\end{aligned}$$

The association schema for an association in a CD is obtained using the following rule:

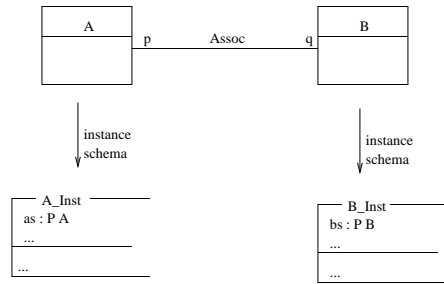


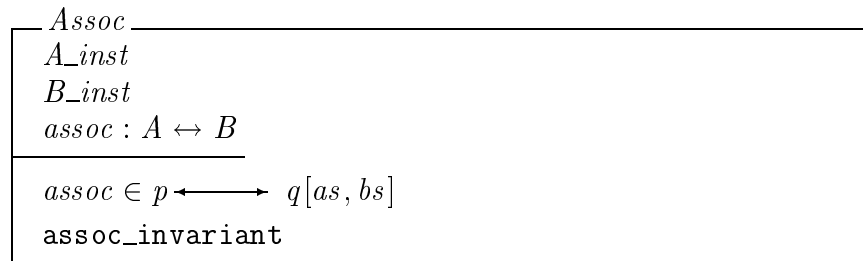
Figure 5: CD for AS Rule

Association Schema (AS) Rule

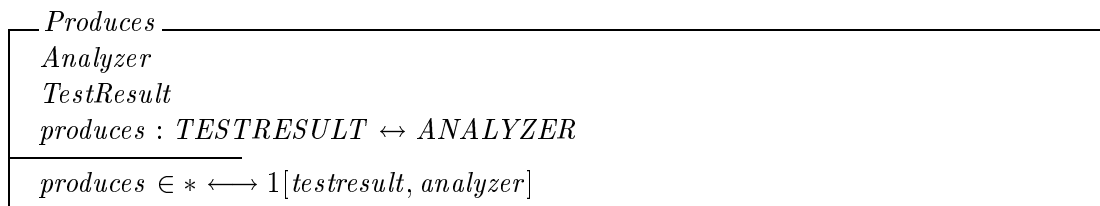
Let *Assoc* be an association between classes *A* and *B* that have instance schemata:

- *A_inst*, declaring a configuration set variable, $as : \mathbb{P} A$,
- *B_inst*, declaring a configuration set variable, $bs : \mathbb{P} B$.

Let *Assoc* have the multiplicity *p* at the *A* end and the multiplicity *q* at the *B* end, and let *assoc_invariant* be an optional, user-supplied invariant that further constrains the association. The association is characterized by the following *association schema* (see Fig. 5 for an illustration of the association):



Using the AS Rule, the following association schema is produced for the association *produces* shown in the Clinical System CD:



When the generator in the above schema is expanded and simplified the following schema is obtained:

<i>Produces</i> <i>Analyzer</i> <i>TestResult</i> <i>produces</i> : <i>TESTRESULT</i> \leftrightarrow <i>ANALYZER</i>
<i>produces</i> \in <i>testresult</i> \rightarrow <i>analyzer</i>

4.4.2 Formalizing specialized associations

A specialized binary association is one that involves a subclass, *Sub*, and another class, *A*, such that there exists an association with the same name between a class, *Super*, that is an ancestor of *Sub*, and the class *A*. An example of a specialized association, *Rel1*, is shown in Fig. 6.

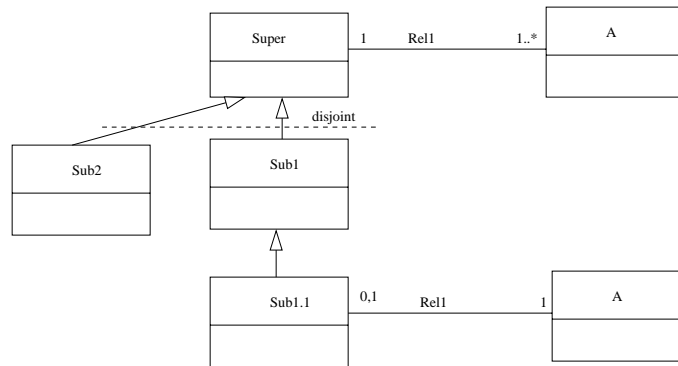


Figure 6: CD for AS Rule

The cardinalities on an association at the superclass level are constraints on the links that can be formed between objects of the superclass and objects of the associated class. Given that instances of subclasses are also instances of superclasses, the cardinalities at the superclass level also constrain the links that can exist between subclass objects and objects of the associated class. A modeler can further restrict the links at the subclass level by explicitly stating cardinalities on the association at the subclass level. These cardinalities must be consistent with the cardinalities given at the superclass level. This is the case for *Rel1* in Fig. 6. At the superclass level, a *Super* object must be associated with one or more *A* objects. This restriction is tightened for *Sub1.1* objects: A *Sub1.1* object must be associated with exactly one *A* object. Similarly, an *A* object must be associated with exactly one *Super* object (this can be a *Sub1* or *Sub2* object). If we restrict our attention to objects of *Sub1.1* then an *A* object can be associated with at most one *Sub1.1* object. The cardinalities of *Rel1* at the subclass level are thus consistent with the cardinalities of *Rel1* at the superclass level.

The rule for formalizing a specialized association is an extension of the AS Rule. The extended association schema includes the association schema for the superclass association, and a conjunct in the predicate part that states that the subclass association is the subset of the superclass association in which the domain elements are objects of the subclass. If the cardinalities at the superclass and subclass levels are consistent with each other then the result is a consistent association schema, else the predicate part is equivalent to false. The *Specialized Association Rule* follows:

Specialized Association Schema (SAS) Rule

Let

- *Assoc* be the association between a subclass *Sub* and a class *B*, where:
 - *Sub_inst* is the instance schema for *Sub* with a configuration set variable, $subs : \mathbb{P} SUB$,
 - *B_inst* is the instance schema for *B* with a configuration set variable, $bs : \mathbb{P} B$.
- *Super_Assoc* be the schema defining the corresponding association, *super_assoc*, between an ancestor of *Sub* called *Super*, and *B*. *Super_Assoc* includes the schema *B_inst*.
- *ROOT* be the object space of the root superclass.

Also, let *Assoc* have the multiplicity *p* at the *Sub* end and the multiplicity *q* at the *B* end, and let **assoc_invariant** be an optional, user-supplied invariant that further constrains the association. The specialized association is characterized by the following association schema:

$\frac{\begin{array}{l} \textit{Assoc} \\ \textit{Sub_inst} \\ \textit{Super_Assoc} \\ \textit{sub_assoc} : \textit{ROOT} \leftrightarrow \textit{B} \end{array}}{\begin{array}{l} \textit{sub_assoc} \in p \longleftrightarrow q[\textit{subs}, \textit{bs}] \\ \textit{sub_assoc} = \textit{subs} \triangleleft \textit{super_assoc} \\ \textbf{assoc_invariant} \end{array}}$

The formalization of the structure shown in Fig. 6 is given below:

Applications of the BIS Rule

$\frac{[SUPER, A] \quad \textit{Super}}{\textit{supers} : \mathbb{P} SUPER}$
--

$\frac{\textit{A_Class}}{\textit{as} : \mathbb{P} A}$
--

Applications of the SIS Rule

$\frac{\begin{array}{l} \textit{Sub1} \\ \textit{Super} \\ \textit{SUB1}, \textit{sub1s} : \mathbb{P} SUPER \end{array}}{\textit{sub1s} = \{s : \textit{supers} \mid s \in \textit{SUB1}\}}$
--

$\frac{\begin{array}{l} \textit{Sub2} \\ \textit{Super} \\ \textit{SUB2}, \textit{sub2s} : \mathbb{P} SUPER \end{array}}{\textit{sub2s} = \{s : \textit{supers} \mid s \in \textit{SUB2}\}}$
--

$Sub11$ $Sub1$ $SUB11, sub11s : \mathbb{P} SUPER$
$sub11s = \{s : supers \mid s \in SUB11\}$

Application of the GS Rule

$SuperConfig$ $Sub2$ $Sub11$
$disjoint(SUB1, SUB2)$

Application of the AS Rule

$Super_Rel1$ $Super$ A_Class $super_rel1 : SUPER \leftrightarrow A$
$super_rel1 \in 1 \longleftrightarrow (1..*)[supers, as]$

Application of the SAS Rule

Sub_Rel1 $Super_Rel1$ $Sub11$ $sub_rel1 : SUPER \leftrightarrow A$
$sub_rel1 \in (0, 1) \longleftrightarrow 1[sub11s, as]$
$sub_rel1 = sub11s \triangleleft super_rel1$

4.4.3 Formalizing (non-specialized) aggregations

In this section we give the rules for formalizing aggregations, including one form of encapsulating aggregations.

Formalizing non-encapsulating UML aggregation As was done for general association, we created a parameterized generator for expressing compositions. The generator for a composition is called *Comp-Gen* and it has the following form:

$$q \bowtie p[comp, agg]$$

In the above, the parameter p is the multiplicity at the aggregate-end, parameter q is the multiplicity at the component-end, parameter $comp$ is the configuration set variable for a component class, and parameter agg is the configuration set variable for the component's aggregate (whole) class. In general, the instantiated generator:

$$\{(a_1..b_1), \dots, (a_n..b_n), (p_1..*), \dots, (p_m..*)\} \bowtie p[comp, agg]$$

where p is 1 or 0..1, expands to:

$$\{s : comp \leftrightarrow agg \mid (\forall x : \text{ran } s \bullet (\forall_{i=1, \dots, n} (a_i \leq \#(s \sim (\{x\}))) \leq b_i)) \vee (\forall_{i=1, \dots, m} (p_i \leq \#(s \sim (\{x\}))))\}$$

The rule for transforming compositions to Z specifications follows:

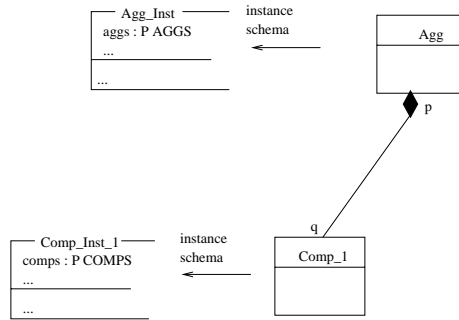


Figure 7: CD for CS Rule

Composition Schema (CS) Rule

Let

- *Agg* be the aggregate class of a composition that has an instance schema *Agg_Inst* in which the configuration set variable, *aggs*, is declared,
- *Comp* be a component of the composition with instance schemata *Comp_Inst* in which configuration set variable, *comps*, is declared,
- *q* be the multiplicity at the *Comp* end and *p* be the multiplicity at the aggregate class end, where *p* is either 0..1 or 1, and
- **agg_invariant** be an optional, user-supplied invariant that defines further constraints on the composition.

The association schema for the above composition is (see Fig. 7 for a CD with the above structure):

$Comp_Agg$ Agg_Inst $Comp_Inst$ $comp_agg : COMPS \leftrightarrow AGGS$
$comp_agg \in q \bowtie p [comps, aggs]$ agg_invariant

In the Clinical System CD, the aggregation between the *Analyzer* and the *SampleSlot* classes is an example of composition. Application of the CS Rule to this part of the CD results in the following specification (the generator is expanded and simplified in what follows):

<i>Analyzer_Agg</i> <i>Analyzer</i> <i>SampleSlot</i> <i>analyzer_agg</i> : <i>SAMPLESLOT</i> \leftrightarrow <i>ANALYZER</i>
<i>analyzer_agg</i> \in (<i>sampleslot</i> \rightarrow <i>analyzer</i>)

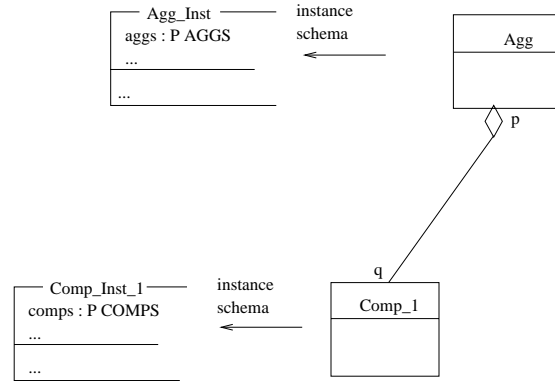


Figure 8: CD for WAS Rule

The weak form of aggregation weakens the functional relationship between components and their aggregates to a general relation (allowing for the sharing of components). Weak aggregation between a component class and an aggregate class is structurally equivalent to a general association between the classes. The generator *Assoc-Gen* is used to define weak aggregation, as is shown in the following rule:

Weak Aggregate Schema (WAS) Rule

Let

- *Agg* be the aggregate class of an aggregation that has an instance schema *Agg_Inst* in which the configuration set variable, *aggs*, is declared,
- *Comp* be a component of the aggregation with instance schemata *Comp_Inst* that declare configuration set variable, *comps*,
- *q* be the multiplicity at the *Comp* end and *p* be the multiplicity at the aggregate class end, and
- **agg_invariant** be an optional, user-supplied invariant that defines further constraints on the aggregation structure.

The schema characterizing the above weak aggregation structure is (see Fig. 8 for an illustration of the above structure):

$\frac{\begin{array}{l} \textit{Weak_Agg} \\ \textit{Agg_Inst} \\ \textit{Comp_Inst} \\ \textit{comp_agg} : \textit{COMPS} \leftrightarrow \textit{AGGS} \end{array}}{\begin{array}{l} \textit{comp_agg} \in q \longleftrightarrow p[\textit{comps}, \textit{aggs}] \\ \textit{weak_agg_invariant} \end{array}}$

An example of the formalization of weak aggregation structures will be given later in this section.

Formalizing encapsulating weak aggregation In the example shown in Fig. 9, the encapsulating aggregate *Agg* consists of two component classes and an encapsulated association. The following are the constraints on *Agg* objects and their component objects in a configuration:

- Each *Agg* object must consist of three *Comp_1* objects and one or more *Comp_2* objects.
- All *Comp_1* and *Comp_2* objects in an *Agg* object *must* be linked and the links must respect the multiplicity of *Rel*. This means that each *Comp_1* object in the aggregate must be linked to *one* or more *Comp_2* objects in the aggregate, and each *Comp_2* object must be linked to one to three *Comp_1* objects.

An encapsulating weak aggregation is transformed in UMLtranZ to an extended Weak Aggregate Schema. The extensions include a specification of associations between component classes and a predicate that stipulates that objects of associated component classes must be linked in a manner consistent with the cardinalities of the association.

If *comp1* and *comp2* are two component classes in an encapsulating weak aggregate structure with a whole class *whole*, and *comp1* and *comp2* are related via an encapsulated association *rel*, then the constraint on how objects of *comp1* and *comp2* can be linked in a whole instance can be represented diagrammatically as follows (note that the diagram does not commute for all component and whole objects):

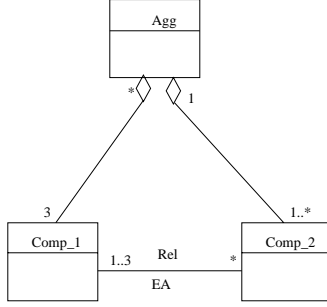
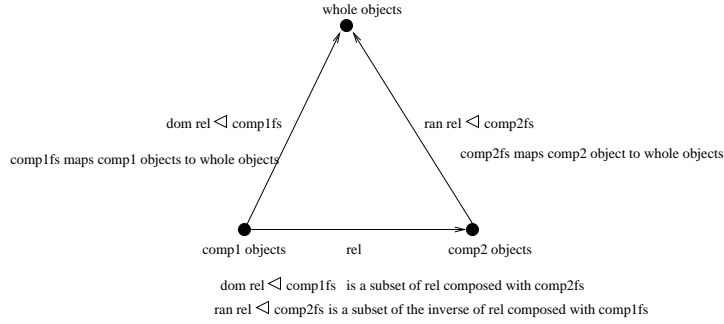


Figure 9: An encapsulating weak aggregation



A consequence of this constraint is that a *comp1* object that is linked via *rel* to *comp2* objects cannot appear unlinked in a *whole* object: A subset of its linked *comp2* objects must also be components of the *whole* object it is a part of. This means that the cardinalities at the component ends must be ranges that do not have 0 as a lower bound.

For example, the aggregation structure shown in Fig. 9 is transformed to the following aggregation schema (in what follows, the source of *Rel* is *COMPS_1* and the target is *COMPS_2*):

<p><i>Agg Config</i></p> <p><i>Agg_Inst</i> [Instance Schema for <i>Agg</i>] <i>Rel_Sc</i> [Association Schema for <i>Rel</i>] <i>comp1_agg</i> : <i>COMPS_1</i> \leftrightarrow <i>AGGS</i> <i>comp2_agg</i> : <i>COMPS_2</i> \leftrightarrow <i>AGGS</i></p> <hr/> <p>$comp1_agg \in 3 \longleftrightarrow *[comps_1, aggs]$ $comp2_agg \in (1..*) \longleftrightarrow 1[comps_2, aggs]$ $(dom\ Rel) \triangleleft comp1_agg \subseteq Rel \circledast comp2_agg$ $(ran\ Rel) \triangleleft comp2_agg \subseteq Rel^{\sim} \circledast comp1_agg$</p>

In the above schema *comps_1*, *comps_2* are configuration set variables declared in the instance schemata *Comp_1* and *Comp_2*, respectively. These instance schemata are included in the association schema *Rel_Sc*.

The following is the encapsulating weak aggregate transformation rule:

Encapsulating Weak Aggregate Schema (EWAS) Rule

- Let *was* be the EWAS (Weak Aggregate Schema) that captures the weak aggregate properties of an encapsulating weak aggregation structure. Included in *Fwas* are the relations $agg_i (i = 1, \dots, p)$ mapping component objects to their aggregate objects.
- Let $Rel_i, (i = 1, \dots, t)$ be the association schemata defining encapsulated associations between component classes. These schemata include instance schemata for the associated component classes and define the relations $rel_i, (i = 1, \dots, t)$, between the objects of the classes.
- Finally, let **fuagg_invariant** be an optional, user-supplied invariant that defines further constraints on the aggregation structure.

The EWAS characterizing the aggregation is constructed as follows:

- In the declaration part include the schemata *Fwas* and Rel_i .
- In the predicate part, include conjuncts that constrain components that are involved in encapsulated associations. The conjuncts that constrain components that take part in encapsulated associations have the following form (in what follows the objects of the component classes *Comp_1* and *Comp_2* are constrained by the encapsulated association rel_i):

$$\begin{aligned} (\text{dom } rel_i) \triangleleft comp1_agg \subseteq rel_i \ ; \ comp2_agg \\ (\text{ran } rel_i) \triangleleft comp2_agg \subseteq rel_i \sim comp1_agg \end{aligned}$$

- In the predicate part include the conjunct **fuagg_invariant**.

Applying the EWAS Rule to the aggregate structure *TestRequest* results in the following specification:

Application of the WAS Rule:

$TRWagg$ <i>TestRequest</i> <i>Sample</i> <i>Test</i> $tragg1 : SAMPLE \leftrightarrow TESTREQUEST$ $tragg2 : TEST \leftrightarrow TESTREQUEST$
$tragg1 \in (1..*) \longleftrightarrow 1[sample, testrequest]$ $tragg2 \in (1..*) \longleftrightarrow 1[test, testrequest]$

Application of the AS Rule to the Carried_out_on association:

$Carried_Out_On$ $Sample$ $Test$ $carried_out_on : SAMPLE \leftrightarrow TEST$
$carried_out_on \in 1 \longleftrightarrow 1[sample, test]$

Application of the EWAS Rule:

$TestRequestAgg$ $TRWagg$ $Carried_Out_On$
$(dom\ carried_out_on) \triangleleft tragg1 \subseteq carried_out_on \ ;\ tragg2$ $(ran\ carried_out_on) \triangleleft tragg2 \subseteq carried_out_on^{\sim} \ ;\ tragg1$

A rule for encapsulating compositions can be similarly defined.

4.5 Formalizing specialized aggregations

An aggregation association between a subclass, SUB , and a class, B , is referred to as a *specialized* aggregation if there exists a corresponding aggregation between an ancestor of SUB and B . The formalization of a specialized aggregation is handled in much the same way as a specialized association: The superclass association schema is included in the subclass association schema and a conjunct that restricts the subclass aggregation to a subset of the superclass aggregation is added to the predicate part of the schema defining the subclass aggregation. For example, the rule characterizing a specialized composition is given below. Similar rules can be defined for other types of specialized aggregations.

Specialized Composition Schema (SCS) Rule

Let:

- *Agg* be a subclass that is also an aggregate class in an aggregation, where *Agg_Inst* is its instance schema that declares a configuration set variable, *aggs*,
- *ROOT* be the object space of the root superclass,
- *SComp* be a component that is linked to the aggregate *Agg* via a specialized association, where *SComp* has an instance schema *SComp_Inst* that declares a configuration set variable, *scomps*,
- *s* be the multiplicity at the *SComp* end and *t* be the multiplicity at the aggregate class, *Agg*, end, where *s* is either 0..1 or 1,
- *Super_SCompAgg* be the aggregation schema that defines the superclass aggregation *super_sagg* that is specialized at the subclass level, and
- **agg_invariant** is an optional, user-supplied invariant that defines further constraints on the aggregation structure.

The schema characterizing the above aggregation structure is:

$ \begin{array}{l} \textit{Comp_Agg} \text{-----} \\ \textit{Agg_Inst} \\ \textit{Super_SCompAgg} \\ \textit{SComp_Inst} \\ \textit{sub_sagg} : \textit{SCOMP} \leftrightarrow \textit{ROOT} \\ \hline \textit{sub_sagg} \in s \bowtie t[\textit{scomps}, \textit{aggs}] \\ \textit{sub_sagg} = \textit{super_sagg} \triangleright \textit{aggs} \\ \textit{agg_invariant} \end{array} $

4.6 Obtaining the configuration schema

The configuration schema is formed by including the association schemata and gen-schemata produced by the transformation process. Some renaming of variables may be needed if unique variable names were not used for different variables across the schemata. The configuration schema for the Clinical System is given in the Appendix.

5 Formalizing Required Behavior

In this section we describe how the dynamic properties of CD constructs and Fusion Operation Schemata are formalized.

5.1 Formalizing Dynamic Properties of CDs

5.1.1 Dynamic Properties of Classes

The attributes of a class can be *changeable*, *frozen*, or *addonly*. A changeable attribute is one on which no restrictions are placed on how its value(s) can be changed (this is the default in UML). A frozen attribute is one whose value “may not be altered after the object is instantiated and its values initialized” (OMG-UML V1.3, pg. 2-24). An addonly attribute is a container with the restriction that new values can be added, but once a value is added to the container it cannot be changed or deleted. Let Cl be a class with frozen attributes $fatt_1, \dots, fatt_m$, and addonly attributes $addatt_1, \dots, addatt_n$, and an instance schema:

Cl
$cls : \mathbb{P} CL$
$cl_attrs : CL \leftrightarrow CL_Attrs$
$m \leq \#cls \leq p$
$\text{dom } cl_attrs = cls$
objs_invariant

The dynamic properties of the class, Cl , are defined in the following schema:

$DynCl$
ΔCl
$\forall o : cls \mid o \in cls' \bullet$
$(\bigwedge_{i=1..m} ((cl_attrs(o)).fatt_i = (cl_attrs'(o)).fatt_i)) \wedge$
$(\bigwedge_{i=1..n} ((1.. \#((cl_attrs(o)).addatt_i)) \triangleleft (cl_attrs'(o)).addatt_i$
$= (cl_attrs(o)).addatt_i))$

5.1.2 Dynamic Properties of Associations

Like an attribute, an association end can be *changeable*, *frozen*, or *addonly*. A changeable association end is one in which no restrictions are placed on how links are set up between objects of the associated classes (the UML default).

If an association end is frozen then the objects at the frozen association end are referred to as *target* objects, and those at the other end are referred to as *source* objects (e.g., see Fig. 10). The OMG-UML V1.3 notion of frozen association ends is expressed as follows (pg. 3-63):

The property {frozen} indicates that no links may be added, deleted, or moved from an object (toward the end with the adornment) after the object is created and initialized.

The above requires that links be created when the source object is created. We feel that this is an unnecessary restriction on link creation. In UMLtranZ we loosen this constraint to allow for the setting up of links after the source object is created, but links must be set up in a single transition (i.e., they must be created during the execution of an atomic operation).

The UML description of frozen association ends is incomplete in that it does not state what impact the frozen property has on the lifetime of the linked objects. For example consider the

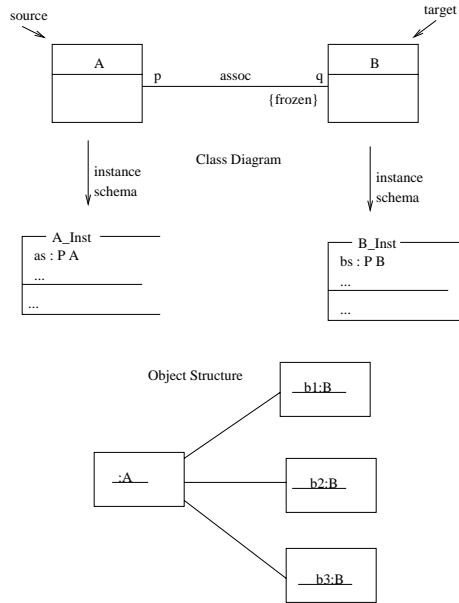


Figure 10: An example of a frozen association end

object structure shown in Fig. 10, in which an A object is linked to three B objects ($b1, b2, b3$). The links between the A object and the B objects are frozen in that the A object cannot be linked to another B object and none of the links can be deleted while their corresponding B objects exist. It is clear that deletion of the A object would result in the destruction of the links (but not necessarily the target objects), but what happens if one of the B objects is deleted before its linked source object is deleted is not discussed in the UML standard document.

UMLtranZ supports two shades of frozen associations: Frozen with lifetime dependencies and frozen with independent lifetimes. If $assoc$ is an association that is *frozen with lifetime dependencies* then the deletion of a B object in the object structure shown in Fig. 10 is not allowed until after the associated A object is deleted. This shade of frozen associations forces a lifetime dependency between source and target objects: A linked target object can be deleted only after all its linked source objects are destroyed (a target object can be linked to more than one source object if permitted by the association end multiplicity). If $assoc$ is frozen with independent lifetimes, then a linked B object can be deleted independently of its linked source object, resulting in the deletion of the corresponding link. In this case an $assoc$ link is frozen as long as either linked object exists.

Let $Assoc$ be a schema defining the static properties of the association $assoc$ shown in Fig. 10:



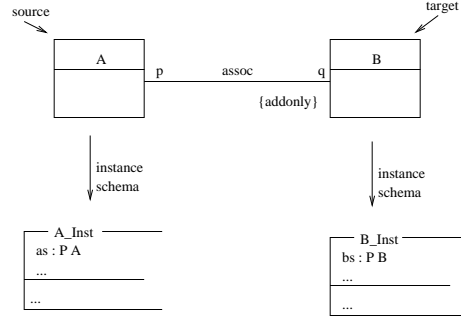


Figure 11: An example of an AddOnly association end

If the association *assoc* is frozen at the *B* end with lifetime dependencies, then the following relationship must be maintained:

$$\begin{array}{|l}
 \hline
 \textit{FrozenDepAssoc} \\
 \hline
 \Delta \textit{Assoc} \\
 \hline
 as' \triangleleft assoc = \text{dom } assoc \triangleleft assoc' \\
 \hline
 \end{array}$$

If the association *assoc* is frozen at the *B* end with independent lifetimes then the following relationship must be maintained:

$$\begin{array}{|l}
 \hline
 \textit{FrozenIndAssoc} \\
 \hline
 \Delta \textit{Assoc} \\
 \hline
 (as' \triangleleft assoc) \triangleright bs' = \text{dom } assoc \triangleleft assoc' \\
 \hline
 \end{array}$$

In UMLtranZ an association end is said to be *addonly* if links can be added to the source object, but none of the previously created links to the target objects can be deleted. Again, we define two shades of the *addonly* property which are formally characterized below for the association shown in Fig. 11:

$$\begin{array}{|l}
 \hline
 \textit{AddDepAssoc} \\
 \hline
 \Delta \textit{Assoc} \\
 \hline
 \forall a : as; b : bs \mid (a, b) \in assoc \bullet (a, b) \in assoc' \vee a \notin as' \\
 \hline
 \end{array}$$

In the case formalized above, once an *assoc* link is created between *a* and *b* it cannot be removed until the *a* element is destroyed. Consequently, the *b* element cannot be destroyed until after the *a* element is destroyed. The other shade of the *addonly* property is defined below for the association shown in Fig. 11:

$$\begin{array}{|l}
 \hline
 \textit{AddIndAssoc} \\
 \hline
 \Delta \textit{Assoc} \\
 \hline
 \forall a : as; b : bs \mid (a, b) \in assoc \bullet (a, b) \in assoc' \vee a \notin as' \vee b \notin bs' \\
 \hline
 \end{array}$$

In the above case a linked *b* object can be deleted before its source *a* object is deleted (in which case the link is deleted).

5.1.3 Formalizing the dynamic aspects of aggregation

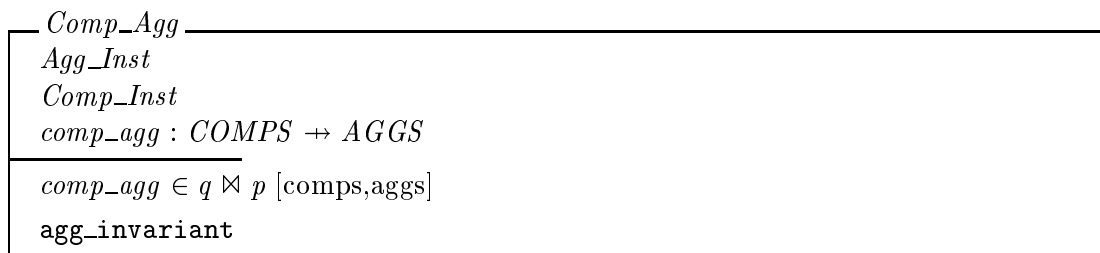
The OMG-UML V1.3 document states the following about compositions (pg. 3-71)

Composition is a form of aggregation with strong ownership and coincident lifetime of part with the whole.

(pg. 2-54)

Composite aggregation is a strong form of aggregation which requires that a part instance be included in at most one composite at a time, although the owner may be changed over time. Furthermore, a composite implies propagation semantics ... For example, if the whole is copied or deleted, then so are the parts as well. A shared aggregation denotes weak ownership (i.e., the part may be included in several aggregates) ...

The UML notion of *strong ownership* seems to be defined in terms of the multiplicity at the whole end (which is restricted to 0..1 or 1 in compositions), and is thus a static property (see previous section). It is not clear to us what “coincident lifetimes” means and this is not elaborated in the UML standard document. A literal translation would result in the following interpretation: The parts are created at the same time the whole is created and they are destroyed when the whole is destroyed (consequently, parts cannot be created or destroyed independently of the whole). We find this interpretation to be too restrictive and it contradicts the intent that a part can change owners (see above) and can have no owners (e.g., in the cases where the whole end multiplicity is 0..1) during its lifetime. In UMLtranZ, a part can be created at anytime, but if the multiplicity indicates that the part must be linked to a whole (multiplicity of 1 at the whole end) then the part must be linked to a whole object when it is created. Furthermore, if a UMLtranZ composition is deleted then all the parts currently in the composition are also deleted. We express this constraint in terms of delete composition operations. For example, consider the composition shown in Fig. 7. The composition schema for this structure is given below:



Deletion of *Agg* objects is specified below:

$\Delta \text{Comp_Agg}$ $\text{delaggs?} : \mathbb{P} \text{AGGS}$
$\text{delaggs?} \subseteq \text{aggs}$ $\text{aggs}' = \text{aggs} \setminus \text{delaggs?}$ $\text{comps}' = \text{comps} \setminus \{c : \text{comps}; a : \text{delaggs?} \mid \text{comp_agg}(c) = a \bullet c\}$

In the previous chapter we noted that weak aggregation is structurally equivalent to general association. Unfortunately, the UML document does not provide enough information to make a distinction between the two concepts from either a static or behavioral perspective. What they do state is that a weak aggregation does not imply propagation semantics (e.g., deletion of the whole does not imply deletion of the parts in weak aggregation). In UMLtranZ the distinction between weak aggregation and general binary association is made from the dynamic perspective. The following dynamic property is associated with weak aggregations (and not general associations) in UMLtranZ: If a whole in a weak aggregation is deleted then all its parts *that are not shared* are also deleted. This interpretation is consistent with our general notion of aggregation in which death of the whole implies death of unshared parts. There are other ways of distinguishing weak aggregation from general binary association that we are currently considering [25].

Like composition, the dynamic property on weak aggregations is expressed in terms of operation schemata. For example, consider the weak aggregation shown in Fig. 8. Its WAS is given below:

$\Delta \text{Weak_Agg}$ Agg_Inst Comp_Inst $\text{comp_agg} : \text{COMPS} \leftrightarrow \text{AGGS}$
$\text{comp_agg} \in q \iff p[\text{comps}, \text{aggs}]$ weak_agg_invariant

Deletion of objects in *aggs* is specified below:

$\Delta \text{DelWAgg}$ $\Delta \text{Weak_Agg}$ $\text{delaggs?} : \mathbb{P} \text{AGGS}$
$\text{delaggs?} \subseteq \text{aggs}$ $\text{aggs}' = \text{aggs} \setminus \text{delaggs?}$ $\text{comps}' = \text{comps} \setminus \{c : \text{comps}; a : \text{delaggs?} \mid \#(\text{comp_agg}(\{c\})) = 1$ $\quad \wedge (c, a) \in \text{comp_agg} \bullet c\}$

Deletion of encapsulating aggregates also implies deletions of linked objects that are not shared. In the case where one linked object is shared and the other is not shared, then only the unshared object is deleted when the whole is deleted.

When developing formalizations of Operation Schemata one is obligated to prove that the dynamic properties of CD constructs are preserved. This is done as part of the rigorous analysis

activity (see section 6). Alternatively one can include the properties in the formalizations of the operation schemata (this is not the approach taken in this paper).

5.2 Formalizing an Operation Schema

A Fusion Operation Model consists of a set of operation schemata. Fusion’s operation schemata are expressed mostly in natural language. It is not likely that a tool that automatically produces a formal specification using only the informally stated requirements in the operation schema can be built currently. A tool that produces a partial specification is a more realistic possibility. We use the following guidelines to obtain a partial Z operation schema from the variant Fusion operation schema, which was defined in Section 2.2.2.

1. The operation name is used as the name of the Z operation schema.
2. If the **Modifies** section is non-empty, then the expression $\Delta SystemConfig$, where *SystemConfig* is the configuration schema for the CD, is placed in the declaration part of the Z operation schema. This expression indicates that a change in the configuration *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 **Modifies** section is empty, then the declaration $\exists SystemConfig$, that introduces a predicate equating the before and after configurations, is used.
3. In the **Inputs** section, **supplied** elements are inputs to the operation, and the other elements are elements in the ‘before’ application state that are manipulated by the operation. For each supplied element an input variable (with a name ending in ‘?’) is declared in the Z schema. The other elements can be specified as local variables. If this is done, the local variables are references to the corresponding state elements and such correspondence must be specified in the predicate part of the schema.
4. In the **Modifies** section, **new** elements are instances created by the operation. A local variable is declared for each of these elements. For every **new** element, a conjunct stating that the element is not in the configuration before execution, but is in the configuration after execution, is required in the predicate part of the operation schema. The other elements can be specified as local variables. If this is done, the local variables are references to the corresponding state elements and such correspondence must be specified in the predicate part of the schema.
5. The data sent in the **Outputs** section are declared as output variables in the Z schema (with a name ending in ‘!’).
6. The description in the **Pre-conditions** section is reexpressed as a conjunct in the predicate part of the Z schema. Because of the natural language used, this reexpression requires human effort. The resulting conjunct is part of the precondition of the operation. Local variables can be added to the declaration part if this results in clearer formalizations of the assumptions in Z.

- The description in the **Post-condition** section is reexpressed as a conjunct describing the effect of the operation in the predicate part of the Z schema. Because of the natural language used in the UMLtranZ style operation schemata this reexpression requires human effort. The resulting conjunct is part of the postcondition of the operation. Local variables can be added to the declaration part if this results in clearer formalizations of the operation effects in Z. If the effects involve deleting aggregate objects then one must make sure that the unshared parts of the aggregation are also deleted.

The UMLtranZ operation schema for the *enter test request* operation of the Clinical System is given below:

Operation	: enter test request
Description	: Enter information about a sample and the test to be performed on it.
Inputs	: supplied <i>sample_info</i> supplied <i>test_info</i> analyzer
Modifies	: new <i>sample</i> new <i>test</i> new <i>test_request</i> new <i>label_info</i> <i>current_patient_report</i> <i>sample_slots</i> of <i>analyzers</i>
Outputs	: <i>labeler</i> : <i>send_label_info</i>
Pre-conditions	: There is a patient report that has been selected as current, <i>current_patient_report</i>
Post-conditions	: A new test request has been created to hold information about the requested test type and sample; [a] A slot on an analyzer has been selected for use in processing the requested test; [b] The length of the queue for the selected analyzer slot has been incremented by 1; [c] A description of the sample, test to be performed, and the analyzer slot to be used has been created and sent to the labeler; [d] The new test request has been added to the <i>current_patient_report</i> . [e]

The following is an initial attempt at producing a Z schema for the *enter test request*. The declarations that are marked by ‘*’ in the schema are local variables that are added to ease the task of writing conjuncts in the predicate part. The parts marked by ‘??’ indicate that not enough information is available in the above UMLtranZ OS to complete the expression. The schema *ClinicalConfig0* is the configuration schema for the Clinical System.

enter_test_request

Δ *ClinicalConfig*

si? : *Sample_Info*

ti? : *Test_Info*

current_patient_report : *PATIENTREPORT*

t : *TEST*

tr : *TESTREQUEST*

s : *SAMPLE*

l : *LABELINFO*

pat : *PATIENT* *

slot : *SAMPLESLOT* *

slotattr : *SampleSlot_Attrs* *

send_label_info! : *TESTREQUEST* \times *SAMPLESLOT*

Formalization of [a]

$(tr \notin \text{testrequest} \wedge \text{testrequest}' = \text{testrequest} \cup \{tr\} \wedge$

$t \notin \text{test} \wedge \text{test}' = \text{test} \cup \{t\} \wedge$

$s \notin \text{sample} \wedge \text{sample}' = \text{sample} \cup \{s\} \wedge$

$\text{sample_attribute}' = \text{sample_attribute} \cup \{(s, ??)\} \wedge$

$\text{carried_out_on}' = \text{carried_out_on} \cup \{(s, t)\} \wedge$

$\text{sample_testreq}' = \text{sample_testreq} \cup \{(s, tr)\} \wedge \text{test_testreq}' = \text{test_testreq} \cup \{(t, tr)\})$

Formalization of [b]

$(\exists s : \text{sampleslot} \bullet s = \text{slot} \wedge$

$\text{assigned_to}' = \text{assigned_to} \cup \{(tr, slot)\})$

Formalization of [c]

$(\text{slotattr}.qlength = (\text{sampleslot_attribute}(\text{slot})).qlength + 1 \wedge$

$\text{sampleslot_attribute}' = \text{sampleslot_attribute} \oplus \{(slot, \text{slotattr})\})$

Formalization of [d]

$\text{send_label_info}' = (tr, slot)$

Formalization of [e]

$(\exists rep : \text{patientreport} \bullet \text{current_patient_report} = rep \wedge$

$\text{patient_patrep} \sim (\{rep\}) = \{pat\} \wedge$

$\text{drawn_from}' = \text{drawn_from} \cup \{(pat, s)\} \wedge \text{testreq_patrep}' = \text{testreq_patrep} \cup \{(tr, rep)\})$

All other state variables are unchanged

$(\text{testresult}' = \text{testresult} \wedge \text{patient}' = \text{patient} \wedge \dots)$

The initial formalization uncovered the following problems with the original Fusion operation schema:

- In the **Reads** section, the elements *sample_info* and *test_info* are mentioned, but they do not appear in the CD. The intent is that *Sample_Info* be a value for the *Sample* attribute *quality*. The class *Test* did not have any attributes, but the above operation indicates that it should. To address this problem an attribute called *test_info* with type *TEST_INFO* is added to the class *Test*. If more information is available about the attributes of *Test* then a more refined listing of attributes is possible. The above modification to the *Test* class is shown in Fig. 12.

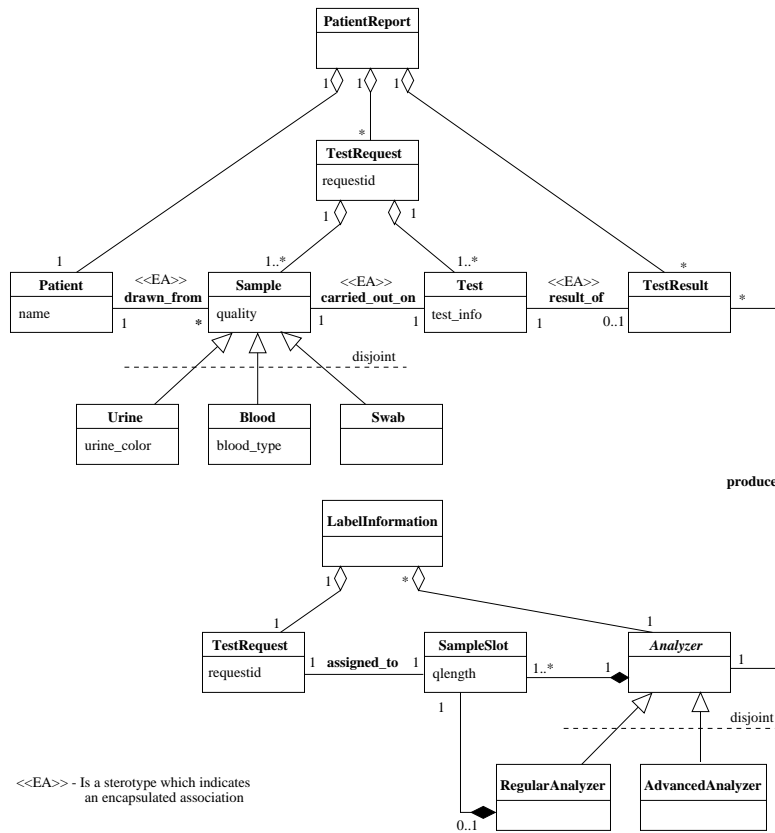


Figure 12: Modified Clinical Class Diagram

- In the **Changes** section, the element *new labelInfo* is introduced, but it was not referenced in the Assumes or Results section. We interpreted this statement as indicating the creation of an object of *LabelInformation*, but this object is not created by this operation; the information for building this instance is passed to the *labeler*, who then creates the object. This interpretation is consistent with the description given in the **Result** section, and with the Object Interaction Graph for the operation given in [26].

The modified formalization of the Fusion operation is given below (the schema *ClinicalConfig* is fully defined in the appendix):

enter_test_request

Δ *ClinicalConfig*

si? : *QUALITY*

ti? : *Test_Info*

current_patient_report : *PATIENTREPORT*

t : *TEST*

tr : *TESTREQUEST*

s : *SAMPLE*

l : *LABELINFOMATION*

pat : *PATIENT* *

slot : *SAMPLESLOT* *

slotattr : *SampleSlot_Attrs* *

testattr : *Test_Attrs* [*Test_Attrs* is type of test attribute]

send_label_info! : *TESTREQUEST* \times *SAMPLESLOT*

Formalization of [a]

$(tr \notin \text{testrequest} \wedge \text{testrequest}' = \text{testrequest} \cup \{tr\} \wedge$

$t \notin \text{test} \wedge \text{test}' = \text{test} \cup \{t\} \wedge$

$\text{testattr.test_info} = ti? \wedge \text{test_attribute}' = \text{test_attribute} \cup \{(t, \text{testattr})\} \wedge$

$s \notin \text{sample} \wedge \text{sample}' = \text{sample} \cup \{s\} \wedge$

$\text{sampleattr.quality} = si? \wedge \text{sample_attribute}' = \text{sample_attribute} \cup \{(s, \text{sampleattr})\} \wedge$

$\text{carried_out_on}' = \text{carried_out_on} \cup \{(s, t)\} \wedge$

$\text{sample_testreq}' = \text{sample_testreq} \cup \{(s, tr)\} \wedge \text{test_testreq}' = \text{test_testreq} \cup \{(t, tr)\})$

Formalization of [b]

$(\exists s : \text{sampleslot} \bullet s = \text{slot} \wedge$

$\text{assigned_to}' = \text{assigned_to} \cup \{(tr, \text{slot})\})$

Formalization of [c]

$(\text{slotattr.qlength} = (\text{sampleslot_attribute}(\text{slot})).qlength + 1 \wedge$

$\text{sampleslot_attribute}' = \text{sampleslot_attribute} \oplus \{(\text{slot}, \text{slotattr})\})$

Formalization of [d]

$\text{send_label_info!} = (tr, \text{slot})$

Formalization of [e]

$(\exists rep : \text{patientreport} \bullet \text{current_patient_report} = rep \wedge$

$\text{patient_patrep} \sim (\{rep\}) = \{pat\} \wedge$

$\text{drawn_from}' = \text{drawn_from} \cup \{(pat, s)\} \wedge \text{testreq_patrep}' = \text{testreq_patrep} \cup \{(tr, rep)\})$

All other state variables are unchanged

$(\text{testresult}' = \text{testresult} \wedge \text{patient}' = \text{patient} \wedge \dots)$

Information in the more formal representation of the operation schema can be used to make the informal descriptions more precise. A more precise UMLtranZ operation schema based on the above formalization follows:

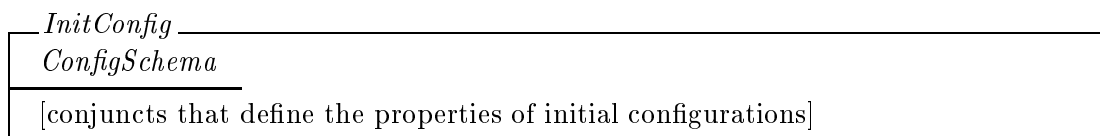
Operation	: enter test request
Description	: Enter information about a sample and the test to be performed on it.
Inputs	: supplied <i>sample_info</i> : <i>QUALITY</i> supplied <i>test_info</i> : <i>Test_Attrs</i> <i>sampleslot</i> : <i>SAMPLESLOT</i> where <i>sampleslot</i> is a slot in an existing analyzer
Modifies	: new <i>sample</i> : <i>SAMPLE</i> new <i>test</i> : <i>TEST</i> new <i>test_request</i> : <i>TESTREQUEST</i> new <i>label_info</i> : <i>LABELINFOMATION</i> <i>current_patient_report</i> : <i>PATIENTREPORT</i> <i>sampleslot</i>
Outputs	: <i>labeler</i> : <i>send_label_info</i>
Pre-conditions	: There is a patient report that has been selected as current, <i>current_patient_report</i>
Post-conditions	: A new test request has been created to hold information about the requested test type and sample; [a] A slot, <i>sampleslot</i> , on an analyzer has been selected for use in processing the requested test; [b] The length of the queue for <i>sampleslot</i> has been incremented by 1; [c] <i>sample</i> , <i>test</i> , and <i>sampleslot</i> are sent to the labeler; [d] The new test request has been added to the <i>current_patient_report</i> . [e]

6 Rigorous Analysis in UMLtranZ

In this section we give an overview of the types of rigorous analyses that can be carried out on formalized CDs and Operation Models.

6.1 CD Analysis

The configuration schema produced by the UMLtranZ transformation approach can be used to check the consistency of the corresponding CD. Furthermore, if a schema characterizing initial configurations is defined, then one is obligated to show that the configurations it characterizes are configurations characterized by the CD's configuration schema. A schema characterizing initial configurations has the following form:



In the above, *ConfigSchema* is the configuration schema and the predicate part consists of conjuncts that characterize the initial configuration sets for objects, links between objects of these sets, and mappings from the objects to their initial attribute values. Given such an initial configuration

schema, one has to show that:

$$\vdash \exists \text{ ConfigSchema} \bullet \text{ InitConfig}$$

that is, that there exists a configuration that satisfies the initial configuration schema.

The precise characterization of a CD can also be used to infer structural properties of the modeled application. The need to establish these properties may arise out of the need to show that the model conforms to certain requirements, or out of challenges posed by reviewers of the models. Rigorous analysis may also be required to tackle questions for which answers are not explicitly given in the model. For example, in the Clinical system one may ask whether an aggregation between *AdvanceAnalyzer* and *SampleSlot* is implied by the aggregation between *Analyzer* and *SampleSlot* and if so what are the constraints on the aggregation. An informal analysis of the CD leads to the conjecture that is expressed in UML terms in Fig. 13. The diagram expresses the conjecture that the

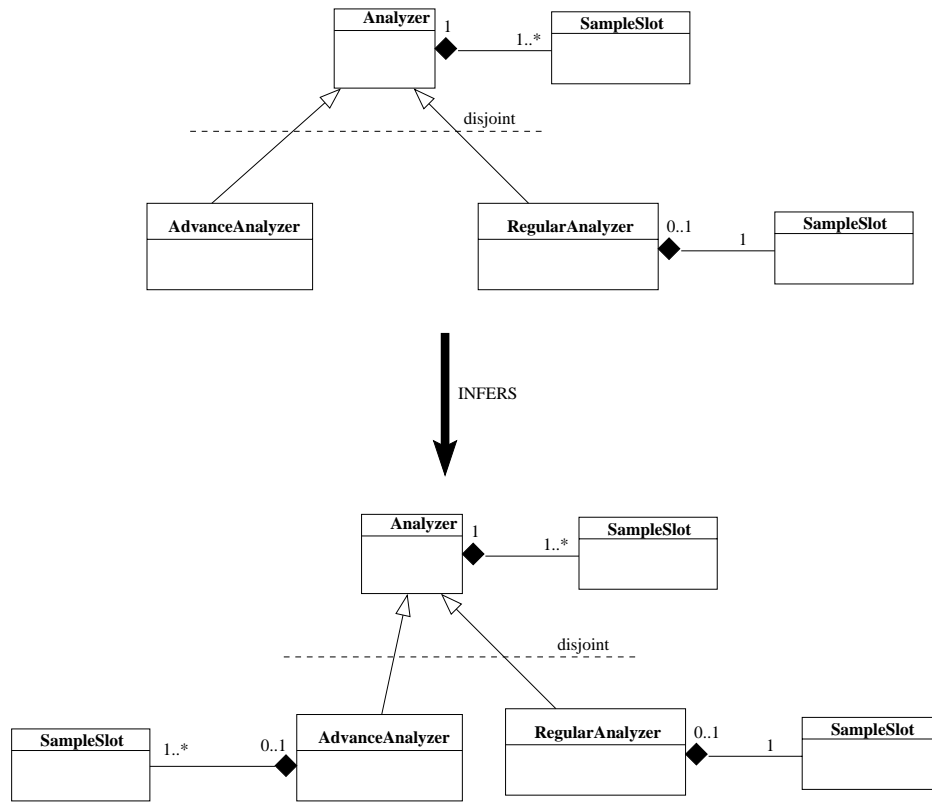


Figure 13: UML Inference Diagram

aggregation at the *Analyzer* level implies that an aggregation also holds between *AdvanceAnalyzer* and *SampleSlot*, where the multiplicity at the *SampleSlot* end is 1..* and the multiplicity at the *AdvanceAnalyzer* end is 0..1. The informal reasoning that produced this conjecture follows:

An advanced analyzer is an analyzer, hence it can be associated with one or more sample slots. We note that this is true in the absence of further information that can further constrain the number of slots associated with an advanced analyzer. It is known

that regular analyzers have sample slots and that regular analyzers are distinct from advanced analyzers (the respective classes are disjoint). From this we can conclude that a sample slot either belongs to a regular analyzer or to an advanced analyzer, but not to both at the same time. This implies that a sample slot is either associated with one advanced analyzer or it is not associated with any analyzer.

The specification characterizing the CD at the top of the “inference” diagram in Fig. 13 is given below (the following specification is not in the modular form that would result from application of the transformation process outlined in this paper; the following collapsed and simplified form is used to make the presentation concise):

$$\begin{array}{c}
\boxed{\begin{array}{l} [ANALYZER, SLOT] \\ \textit{Analyzer} \text{-----} \\ \textit{analyzers} : \mathbb{P} ANALYZER \end{array}} \quad \boxed{\begin{array}{l} \textit{Slot} \text{-----} \\ \textit{slots} : \mathbb{P} SLOT \end{array}} \\
\\
\boxed{\begin{array}{l} \textit{Analyzer Config} \text{-----} \\ \textit{Analyzer} \\ \textit{Slot} \\ \textit{ADVS}, \textit{REGS}, \textit{adv}, \textit{regs} : \mathbb{P} ANALYZER \\ \textit{super_agg}, \textit{sub_agg} : SLOT \twoheadrightarrow ANALYZER \\ \hline \textit{adv} = \{x : \textit{analyzers} \mid x \in \textit{ADVS}\} \\ \textit{regs} = \{x : \textit{analyzers} \mid x \in \textit{REGS}\} \\ \textit{ADVS} \cap \textit{REGS} = \emptyset \\ \textit{adv} \cup \textit{regs} = \textit{analyzers} \\ \textit{super_agg} \in \textit{slots} \twoheadrightarrow \textit{analyzers} \\ \textit{sub_agg} \in \textit{slots} \twoheadrightarrow \textit{regs} \\ \textit{sub_agg} = \textit{super_agg} \triangleright \textit{regs} \end{array}}
\end{array}$$

The conjecture that a sample slot is a part of zero or one advanced analyzer can be formally stated as follows:

$$\textit{AnalyzerConfig} \vdash \forall s : \textit{slots} \bullet ((\exists a : \textit{adv} \bullet \textit{super_agg}(s) = a) \not\equiv (\forall a : \textit{adv} \bullet \textit{super_agg}(s) \neq a))$$

An outline of the proof of the above conjecture is given below:

- From the definition of *super_agg* we get:

$$\forall s : \textit{slots}; \exists an : \textit{analyzers} \bullet \textit{super_agg}(s) = an$$

- Given that $\textit{analyzers} = \textit{adv} \cup \textit{regs}$, the above is equivalent to:

$$\forall s : \textit{slots}; \exists an : (\textit{adv} \cup \textit{regs}) \bullet \textit{super_agg}(s) = an$$

which, in turn, is equivalent to:

$$\forall s : \textit{slots} \bullet ((\exists an : \textit{adv} \bullet \textit{super_agg}(s) = an) \vee (\exists an : \textit{regs} \bullet \textit{super_agg}(s) = an))$$

- Given that $regs \cap advs = \emptyset$ the above implies:

$$\forall s : slots \bullet ((\exists a : advs \bullet super_agg(s) = a) \not\equiv (\forall a : advs \bullet super_agg(s) \neq a))$$

End of Proof Sketch

The conjecture that an advanced analyzer can have one or more sample slots (in the absence of information that can further constrain this property) can be expressed as follows:

$$AnalyzerConfig \vdash \forall a : advs \bullet \#(super_agg \sim (\{a\})) \geq 1$$

The proof is similar to the previous proof:

- From the definition of *super_agg* we get:

$$\forall a : analyzers \bullet \#(super_agg \sim (\{a\})) \geq 1$$

- Given that $analyzers = advs \cup regs$, the above is equivalent to:

$$\forall a : (regs \cup advs) \bullet \#(super_agg \sim (\{a\})) \geq 1$$

which, in turn, is equivalent to:

$$\forall a : regs; b : advs \bullet \#(super_agg \sim (\{a\})) \geq 1 \wedge \#(super_agg \sim (\{b\})) \geq 1$$

End of Proof Sketch

Another approach to establishing the conjectures is to formalize the two CDs in the “inference” diagram and show that one infers the other.

6.2 Operation Analysis

The Z operation schema produced from a UMLtranZ OS allows one to utilize Z proof obligations for operation schemata as a mechanism for determining the adequacy of the OS description. The following proof obligations are relevant to our formalization of UMLtranZ Operation Models:

- Each operation must be shown to preserve the dynamic properties of classes and associations (if any). Specifically, one is obligated to establish that frozen and addonly attribute properties, and frozen and addonly association end properties are preserved by each operation.
- If an operation deletes an aggregation object, one must establish that the unshared parts are also deleted.
- Each operation must be able to start in at least one configuration. In Z, the precondition of an operation schema, *OpSchema* can be obtained as follows:

$$Pre OpSchema = \exists State' \bullet OpSchema \setminus outputs$$

where *State'* is the configuration schema with all variables primed and *outputs* is the set of output variables. If the precondition of an operation is false this means that there is no configuration in which it can start.

7 Towards CASE Tool Support for UMLtranZ

We built a prototype tool, FuZE [5], that uses an early version of a set of transformation rules to automatically generate Z specifications from Fusion class models. Z analysis tools (ZTC and ZANS) can be called from within the tool to analyze the generated Z specifications. The tool was built as an extension of the CASE tool Paradigm Plus[©] using the available scripting language. The tool has been applied by graduate students at Florida Atlantic University on nontrivial projects and case studies (e.g., see [13]). In general, our experiences indicate that formalization and analysis of informal models can uncover problems with the informal models, and can lead to a deeper understanding of the problem. Our applications of the integrated techniques on case studies also uncovered problems with our previous rules. The rules described in this paper are an improved version of the rules implemented in FuZE.

We are currently modifying FuZE and incorporating the new rules into the environment. The modifications we plan will make FuZE independent of CASE tools. We are currently developing a simple, web-based, tool-independent representation of UML Class Diagrams that will be used as the input to the module that implements the UMLtranZ transformation process. The notation is called HTUML and an overview of its current form can be found at the site <http://www.cs.colostate.edu/~carheden/uml/index.html>. Bruel's group in France is developing a translation from the Rational Rose[©]CASE tool to HTUML, and France's team in the USA is developing a Java implementation of the UMLtranZ transformation process.

8 Conclusion and future works

Popular software specification techniques can be roughly categorized as formal and informal. The *informal, structured techniques* (ISTs) tend to emphasize ease-of-use and understandability, often at the cost of rigor. Examples of ISTs are Structured Analysis and Design Techniques (e.g., see [10, 34]) and Object-Oriented (OO) techniques such as Fusion [7] and OMT [29]. *Formal specification techniques* (FSTs) [6, 30] emphasize formality, sometimes at the cost of ease-of-use and understandability. Examples of popular FSTs are Z [31], VDM [23], and Larch [17].

There is evidence that works on ISTs and FSTs are slowly converging. In the FST domain there is work on making formal techniques more practical and understandable by introducing graphical elements and specification structuring mechanisms that are typically found in ISTs (e.g., see [11, 24]) and by providing some tolerance for informality (e.g., SDL [12]). In the IST domain there is work on making informal modeling notations and concepts more precise (e.g., see [16, 18]) and on transforming informal models to formal models to enable rigorous semantic analyses (e.g., see [3, 14, 19]). The work described in this paper takes this approach.

The act of formalizing an informal model can reveal significant problems that are easily missed in less rigorous analyses of the models. Furthermore, the formal specifications produced can be rigorously analyzed, providing another opportunity for uncovering defects in models. Another benefit of precisely defining the mapping between informal and formal constructs is that it can uncover problems with the informal modeling notations. For example, it can help identify ambiguous structures and structures with parts that have inconsistent properties.

After transformation, the informal models can be made more precise by using the insights gained during formalization and analysis to restate vague statements. Use of an annotation such as the *Object Constraint Language* (OCL) can also improve the precision of the models. When this is

done, the resulting OOA models can be viewed as more readable representations of the underlying formal models. We are currently developing techniques for generating graphical OOA models with precise annotations from appropriately structured Z specifications.

From a technology-transfer perspective, an organization that has made significant investment in object technologies, is more likely to adopt FSTs if they are packaged as a value-added extension to the OO techniques they are developing or are using.

The primary goal of our work is to evolve informal OOA techniques to formal techniques. This can be done by defining precise semantics for the OOA modeling notations and developing mechanisms that allow developers to rigorously analyze the OOA models without the need to generate specifications in another formal notation. We have started to develop techniques that support this vision for the UML. This work is being carried out as part of a collaborative effort to define a precise semantics for the UML. The group that we are working with is called the *precise UML* (pUML) group. The approach taken by the pUML group is to use formal techniques to explore and define appropriate semantic foundations for OO concepts and UML notations, and to use the foundations to develop rules for transforming models to enable rigorous analysis. More information about the pUML effort can be found on the following website: <http://www.cs.york.ac.uk/puml/>

References

- [1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
- [2] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Menlo Park, CA, Second edition, 1994.
- [3] Robert H. Bourdeau and Betty H.C. Cheng. A formal semantics for object model diagrams. *IEEE Transactions on Software Engineering*, 21(10):799–821, October 1995.
- [4] 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 20th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'96)*, Santa Barbara, California, July 29–August 2 1996.
- [6] B. Cohen, W. T. Harwood, and M. I. Jackson. *The specification of complex systems*. Addison-Wesley, 1986.
- [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] Steve Cook and John Daniels. Let's get formal. *Journal of Object-Oriented Programming (JOOP)*, pages 22–24 and 64–66, July–August 1994.

- [9] 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.
- [10] T. DeMarco. *Structured Analysis and System Specification*. Prentice-Hall, 1978.
- [11] 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.
- [12] J. Ellesberg, D. Hogrefe, and A. Sarma, editors. *SDL: Formal Object-Oriented Language for Communication Systems*. Prentice Hall, 1997.
- [13] 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.
- [14] Robert B. France, Jean-Michel Bruel, and Maria M. Larrondo-Petrie. An Integrated Object-Oriented and Formal Modeling Environment. *Journal of Object-Oriented Programming (JOOP)*, 10(7), 1997.
- [15] Robert B. France, Jean-Michel Bruel, Maria M. Larrondo-Petrie, and Emanuel Grant. Rigorous object-oriented modeling: Integrating formal and informal notations. In *Proceedings of the 6th International AMAST Conference*, 1997.
- [16] Robert B. France, Andy Evans, and Kevin Lano. The UML as a formal modeling notation. In Institut für Informatik der Technischen Universität München TUM-19737, editor, *Proceedings of the OOPSLA'97 Workshop on Object-Oriented Behavioral Semantics*, 1997.
- [17] J. V. Guttag, J. J. Horning, and J. M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5), 1985.
- [18] J. Anthony Hall. Using Z as a specification calculus for object-oriented systems. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 290–318. VDM-Europe, Springer-Verlag, New York, 1990.
- [19] Jonathan A. R. Hammond. Producing Z Specifications from Object-Oriented Analysis. In Jonathan P. Bowen and J. Anthony Hall, editors, *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 316–336. Springer-Verlag, New York, 1994.
- [20] I. Jacobson. *Object oriented software engineering*. Addison-Wesley, 1992.
- [21] Xiaoping Jia. *An Approach to Animating Z Specifications*. Division of Software Engineering, School of Computer Science, Telecommunication, and Information Systems, DePaul University, Chicago, IL 60604, USA, 1995. ZANS is a Z animator available on Internet via anonymous ftp at `ise.cs.depaul.edu`.

- [22] 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 60604, USA, May 1995. The ZTC tool and documentation are available on Internet via anonymous ftp at `ise.cs.depaul.edu`.
- [23] C. Jones. *Systematic software development using VDM*. Prentice-Hall, 1986.
- [24] Kevin C. Lano. Z^{++} , an object-orientated extension to Z. In John E. Nicholls, editor, *Z User Workshop, Oxford 1990*, Workshops in Computing, pages 151–172. Springer-Verlag, 1991.
- [25] R. B. France M. Saksena and M. M. Larrondo-Petrie. A characterization of aggregation. In *5th. International Conference on Object Oriented Information Systems*. Springer, 1998.
- [26] Ruth Malan, Reed Letsinger, and Derek Coleman. *Object-Oriented Development At Work: Fusion in the Real World*. Hewlett-Packard professional books. Prentice Hall, Upper Saddle River, NJ, November 1995.
- [27] The Object Management Group (OMG). Unified Modeling Language. Version 1.3, OMG, <http://www.omg.org>, June 1999.
- [28] The UML Partners. Unified Modeling Language. Version 1.1, Rational Software Corporation, Santa Clara, CA-95051, USA, January 1997.
- [29] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
- [30] Hossein Saiedian. An Invitation to Formal Methods. *Computer*, 29(4):16–30, April 1996.
- [31] J. Michael Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, Englewood Cliffs, NJ, Second edition, 1992.
- [32] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.
- [33] R. Wirfs-Brock and B. Wilkerson. *Designing object oriented software*. Prentice-Hall, 1990.
- [34] E. Yourdon. *Modern Systems Analysis*. Prentice-Hall, 1989.

A Applying the UMLTranZ Technique

We illustrate our formalization approach by applying it to the Clinical Laboratory System [26] shown in Fig. 12.

A.1 Building the Basic Model

The basic classes are *Patient*, *PatientReport*, *TestRequest*, *TestResult*, *Sample*, *Test*, *LabelInformation*, *Analyzer* and *SampleSlot*.

A.1.1 Basic Class Object Spaces

[*PATIENT*, *PATIENTREPORT*, *TESTREQUEST*, *TESTRESULT*]

[*LABELINFO*, *ANALYZER*, *SAMPLESLOT*, *SAMPLE*, *TEST*]

A.1.2 Attribute Type Spaces

[*NAME*, *QUALITY*, *COLOR*, *TYPE*, *TEST_INFO*]

A.1.3 Attribute schemata

The following are attribute schemata for *Patient*, *Sample*, *Test* and *SampleSlot*.

Patient_Attrs _____
patient_name : *NAME*

Sample_Attrs _____
sample_quality : *QUALITY*

SampleSlot_Attrs _____
qlength : \mathbb{N}

Test_Attrs _____
test_info : *TEST_INFO*

A.1.4 Basic Class Instance Schemata

Applications of the BIS rule to *PatientReport*, *TestResult*, *Sample*, *Patient*, *LabelInformation*, *Test*, *TestRequest*, *Analyzer*, and *SampleSlot*.

PatientReport _____
patientreport : \mathbb{P} *PATIENTREPORT*

TestResult _____
testresult : \mathbb{P} *TESTRESULT*

Sample _____
sample : \mathbb{P} *SAMPLE*
sample_attribute : *SAMPLE* \leftrightarrow *Sample_Attrs*
 $\text{dom } \textit{sample_attribute} = \textit{sample}$

Patient _____
patient : \mathbb{P} *PATIENT*
patient_attribute : *PATIENT* \leftrightarrow *Patient_Attrs*
 $\text{dom } \textit{patient_attribute} = \textit{patient}$

<i>LabelInformation</i>	<i>Test</i>
$labelinformation : \mathbb{P} LABELINFORMATION$	$test : \mathbb{P} TEST$
	$test_attribute : TEST \rightarrow Test_Attrs$
	$dom\ test_attribute = test$

<i>TestRequest</i>	<i>Analyzer</i>
$testrequest : \mathbb{P} TESTREQUEST$	$analyzer : \mathbb{P} ANALYZER$

<i>SampleSlot</i>
$sampleslot : \mathbb{P} SAMPLESLOT$
$sampleslot_attribute : SAMPLESLOT \rightarrow SampleSlot_Attrs$
$dom\ sampleslot_attribute = sampleslot$

A.2 Building the Specialization Model

The gen-schemata for the *Sample* and *Analyzer* specialization hierarchies are developed in this section. The class *Sample* is non-abstract, and the *Analyzer* class is abstract. In both structures the object spaces of the subclasses are disjoint.

A.2.1 Specialization class attribute schemata

Attribute schemata (AS) are created for specialized classes, *Urine* and *Blood*.

<i>Urine_Attrs</i>	<i>Blood_Attrs</i>
<i>Sample_Attrs</i>	<i>Sample_Attrs</i>
$urine_color : COLOR$	$blood_type : TYPE$

A.2.2 Specialization instance schemata

Subclass instance schemata are created for classes *Swab*, *Urine*, *Blood*, *RegularAnalyzer* and *AdvanceAnalyzer*. Specialized classes with no attributes have no attribute function.

Application of the SIS rule to Swab

<i>Swab</i>
<i>Sample</i>
$SWAB, swab : \mathbb{P} SAMPLE$
$swab = \{x : sample \mid x \in SWAB\}$

Application of the SIS rule to Urine

<p><i>Urine</i></p> <p><i>Sample</i></p> <p>$URINE, urine : \mathbb{P} SAMPLE$</p> <p>$urine_attribute : SAMPLE \mapsto Urine_Attrs$</p> <hr/> <p>$urine = \{x : sample \mid x \in URINE\}$</p> <p>$\forall u : urine \bullet$</p> <p style="padding-left: 2em;">$(urine_attribute(u)).sample_quality$</p> <p style="padding-left: 2em;">$= (sample_attribute(u)).sample_quality$</p> <p>$dom\ urine_attribute = urine$</p>

Application of the SIS rule to Blood

<p><i>Blood</i></p> <p><i>Sample</i></p> <p>$BLOOD, blood : \mathbb{P} SAMPLE$</p> <p>$blood_attribute : SAMPLE \mapsto Blood_Attrs$</p> <hr/> <p>$blood = \{x : sample \mid x \in BLOOD\}$</p> <p>$\forall b : blood \bullet$</p> <p style="padding-left: 2em;">$(blood_attribute(b)).sample_quality$</p> <p style="padding-left: 2em;">$= (sample_attribute(b)).sample_quality$</p> <p>$dom\ blood_attribute = blood$</p>

Application of the SIS rule to Regular

<p><i>RegularAnalyzer</i></p> <p><i>Analyzer</i></p> <p>$REGS, regularanalyzer : \mathbb{P} ANALYZER$</p> <hr/> <p>$regularanalyzer = \{x : analyzer \mid x \in REGS\}$</p>

Application of the SIS rule to AdvanceAnalyzer

<p><i>AdvanceAnalyzer</i></p> <p><i>Analyzer</i></p> <p>$ADVS, advanceanalyzer : \mathbb{P} ANALYZER$</p> <hr/> <p>$advanceanalyzer = \{x : analyzer \mid x \in ADVS\}$</p>

A.2.3 Gen-Schemata for specialization structures

Application of the GS rule to the Sample and Analyzer structures.

$\frac{\text{SampleConfig}}{\text{Swab}} \quad \frac{\text{SampleConfig}}{\text{Urine}} \quad \frac{\text{SampleConfig}}{\text{Blood}}}{\text{disjoint}\langle \text{SWAB}, \text{BLOOD}, \text{URINE} \rangle}$	$\frac{\text{AnalyzerConfig}}{\text{RegularAnalyzer}} \quad \frac{\text{AnalyzerConfig}}{\text{AdvanceAnalyzer}}}{\text{ADV S} \cap \text{REG S} = \emptyset}$ $\text{regularanalyzer} \cup \text{advanceanalyzer} = \text{analyzer}$
--	--

A.3 Formalizing general associations and aggregations

The following are the properties used in developing the formalization that follows:

- The *TestRequest* and *PatientReport* structures are Fusion-specific aggregations.
- The *drawn_from*, *carried_out_on* and *result_of* associations are encapsulated associations.
- The aggregation between *RegularAnalyzer* and *SampleSlot* is a specialization of the composition between *Analyzer* and *SampleSlot*.

A.3.1 Formalizing Associations

Applications of the AS rule to *drawn_from*, *carried_out_on*, *result_of*, *produces*, *assigned_to*.

$\frac{\text{Drawn_From}}{\text{Patient}} \quad \frac{\text{Drawn_From}}{\text{Sample}}}{\text{drawn_from} : \text{SAMPLE} \leftrightarrow \text{PATIENT}}$ $\text{drawn_from} \in \text{sample} \rightarrow \text{patient}$	$\frac{\text{Carried_Out_On}}{\text{Sample}} \quad \frac{\text{Carried_Out_On}}{\text{Test}}}{\text{carried_out_on} : \text{SAMPLE} \leftrightarrow \text{TEST}}$ $\text{carried_out_on} \in \text{sample} \rightsquigarrow \text{test}$
$\frac{\text{Result_Of}}{\text{Test}} \quad \frac{\text{Result_Of}}{\text{TestResult}}}{\text{result_of} : \text{TEST} \leftrightarrow \text{TESTRESULT}}$ $\text{result_of} \in \text{test} \rightsquigarrow \text{testresult}$	$\frac{\text{Produces}}{\text{Analyzer}} \quad \frac{\text{Produces}}{\text{TestResult}}}{\text{produces} : \text{TESTRESULT} \leftrightarrow \text{ANALYZER}}$ $\text{produces} \in \text{testresult} \rightarrow \text{analyzer}$
$\frac{\text{Assigned_To}}{\text{TestRequest}} \quad \frac{\text{Assigned_To}}{\text{SampleSlot}}}{\text{assigned_to} : \text{TESTREQUEST} \leftrightarrow \text{SAMPLESLOT}}$ $\text{assigned_to} \in \text{testrequest} \rightsquigarrow \text{sampleslot}$	

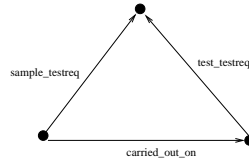
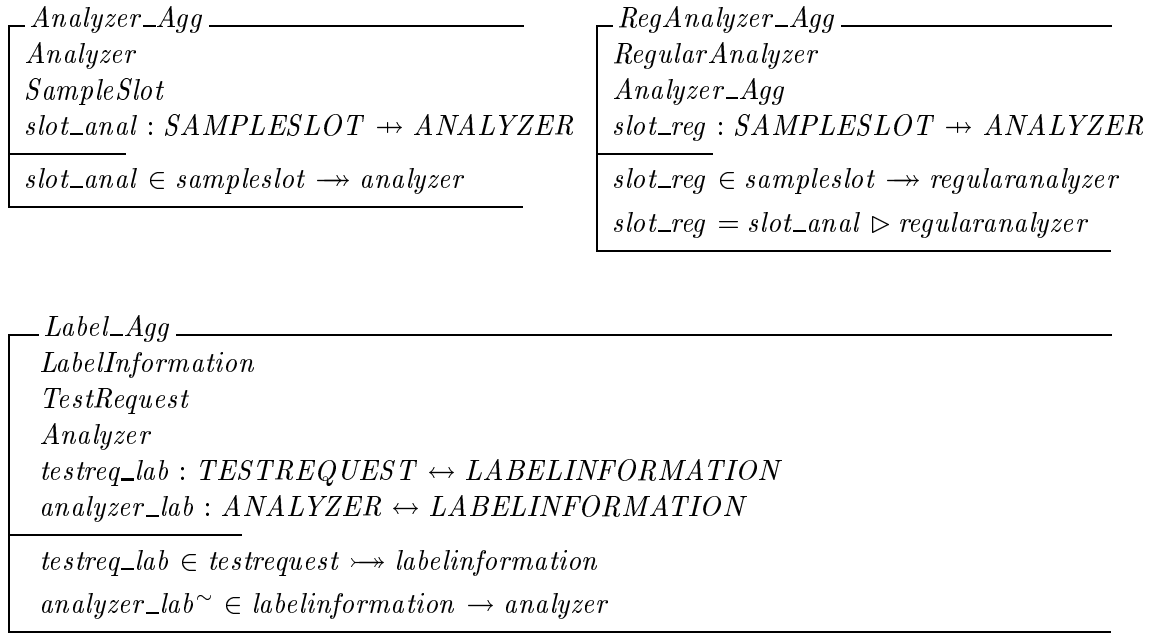


Figure 14: Encapsulated association diagram for carried_out_on

A.3.2 Formalizing UML aggregations

Application of the CS rule to the *Analyzer* aggregation, of the SCS (specialized composition) rule to the *RegularAnalyzer* aggregation, of the WAS rule to the *Labelinformation* aggregation.



A.3.3 Formalizing encapsulated aggregations

Application of the EWAS to the *TestRequest* Aggregation

The diagram we used to develop the constraint on the encapsulated association is shown in Fig. 14.

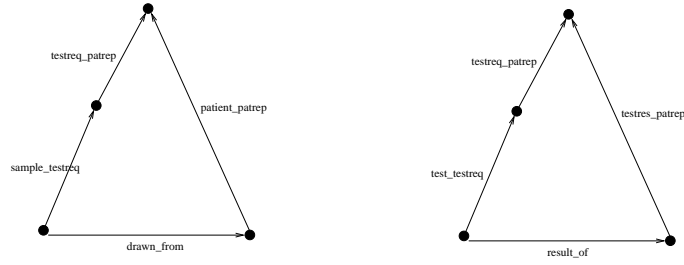
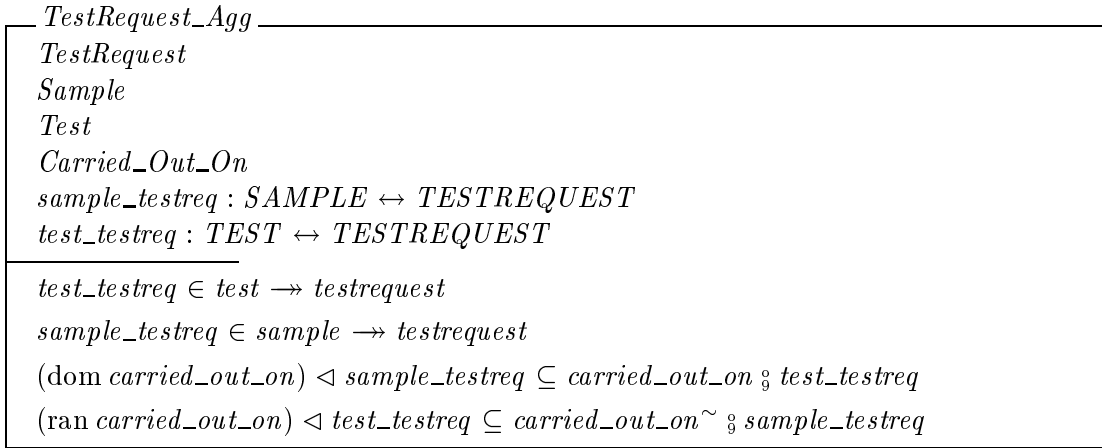
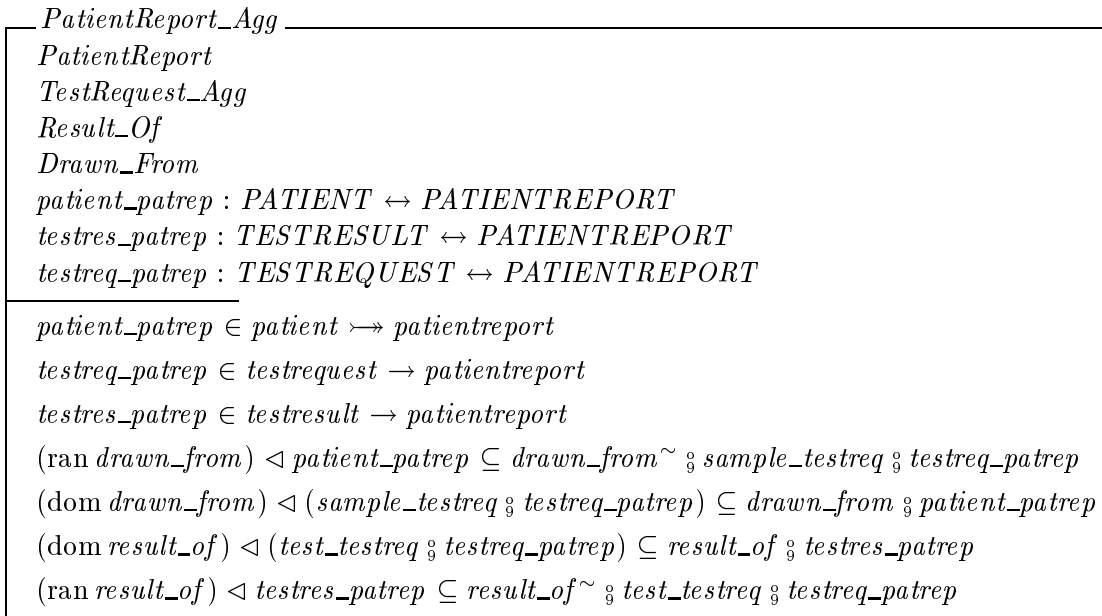


Figure 15: Encapsulated association diagrams for drawn_from and result_of



Application of the EWAS to the PatientReport Aggregation

The diagrams that we used to help formulate the constraints for the encapsulated associations are shown in Fig. 15.



A.4 The Clinical System configuration schema

The following is a representation of the formalized Clinical System CD.

ClinicalConfig
Sample Config
AnalyzerConfig
PatientReport_Agg
Label_Agg
Assigned_To
Analyzer_Agg
RegAnalyzer_Agg
Produces