A Taxonomy of Propositional Logic Constraint Patterns for the Unified Modeling Language

James P. Davis, *Member, IEEE*, Ronald D. Bonnell, *Senior Member, IEEE*

***Abstract*— The Unified Modeling Language (UML) supports the capture and representation of conceptual semantics in analysis and modeling of arbitrary domains of discourse. It is now being used for modeling of database and agent-oriented applications in addition to its use with object-oriented software. An important activity in conceptual modeling is the capture and expression of domain constraints on the underlying data and object states. UML supports the capture of declarative constraints both directly in the graphical class diagram notation and in the separately denoted Object Constraint Language (OCL) syntax. In this paper, we present a set of constraint patterns that have been formalized for capturing a broader category of semantic constraints directly using the class diagram notation. The objective of this work is to enrich UML static models for greater precision in conceptual modeling applications by formalizing the syntax and semantics of these constraint patterns, thus giving analysts a tool for capturing these common patterns directly in the UML class diagram with a minimum of additional notation—without requiring a separate expression of constraints in the adjunct OCL syntax. We present a taxonomy for propositional constraints, discuss their semantics through the use of logic truth tables and Karnaugh maps, and illustrate their usage in UML class diagrams by way of an example from the Unix security domain.**

***Index Terms*—Semantic Constraints, Boolean Logic, Unified Modeling Language, Conceptual Modeling, Knowledge Engineering, Object-Oriented Analysis, Agent-Oriented Software Engineering.**

I. INTRODUCTION

This paper is concerned with facilitating analysts with a means to represent a richer set of invariant constraints using the UML class diagram notation than is currently defined in the base model of the UML specification [1], yet without requiring analysts to resort to the adjunct OCL syntax [2]. The UML class diagram notation supports some basic constraints as defined in the standard, and provides a means to create additional constraint types—thus allowing an analyst to place restrictions on the allowable states of data and object instances during the natural course of knowledge engineering or domain analysis. These constraints are usually captured along with the information about classes and the relationships between them in the course of modeling of some arbitrary domain of discourse.

This work has been submitted to the IEEE for possible publication. Copyright may be transferred without notice, after which this version may be superseded.

Constraints are restrictions on the states that data (and the objects owning or encapsulating the data) can enter, as reflected by the set of valid values that the data can hold, or by relationship configurations in which instances represented by data or objects may assume in valid model extensions. Such constraining of data values or relationship participation is generally governed by a set of assertions about the domain, captured during business analysis, knowledge engineering, or data engineering activities. Constraint patterns are recurring structures defining certain categories of constraints that have been observed in practice across these analysis activities in many different domains of discourse. In the course of teaching this discipline in academia—as well as practicing it in industry—we have codified a set of such constraint patterns that represent propositional statements that can easily be added to the class diagram.

These propositional statements often coincide with named propositional logic functions of two or more variables, as presented in any Discrete Mathematics text, such as [3]; but in other situations, these propositions represent unnamed—and often misunderstood—logic functions. It has been our intention to codify this set of constraint patterns according to their logic function, thus capturing invariants expressed in propositional logic as a means to represent recurring constraint "patterns" observed in the analysis of applications with object-oriented, agent-oriented and database modeling techniques. We have observed an inconsistent and imprecise use of many of these propositional functions in the UML literature, and we set out to correct this by presenting the semantics of these propositional patterns precisely, using the tools commonly used for such formulations in Boolean logic—namely the truth table [3] and the Karnaugh map (K-map), used extensively in the design of digital circuits [17].

In this paper, we present a set of proposition-based invariant constraint patterns for the UML class diagram, and discuss the semantics of the patterns in which these extensions are used. We restrict our focus in this paper to one particular type of invariant known as *inter-relationship constraints*; this category of constraint has alternately been referred to as interlink constraints [4] or relative constraints [5]. This type of constraint involves restricting the meaning, i.e., the number of possible interpretations, which may be ascribed to the presence of different relationships emanating from one class to other classes to which it is connected in a UML class diagram, as prescribed in the OMG specification [1] and discussed as a "constraint between associations" in [20].

Often, it is the superposition of assertions implied by these relationships that can cause uncertainty in how to interpret portions of a UML class diagram—where such constraining decisions are often made arbitrarily during the programming activity. Usually, this ambiguity exists because the presence of multiplicities on the links of relationships is insufficient to fully capture the required semantics [6]. It is our wish to furnish analysts and knowledge engineers with a richer, more refined set of conceptual tools for capturing certain categories of constraints, and to do so using the "palette" of the UML class diagram.

In our research and use of UML (and other notations such as the Entity-Relationship model and its variants used by Teorey et al. [5], Davis et al. [4,6], and Elmasri et al. [16]), we have found that many common types of invariant constraints exist across domains in identifiable patterns, and that a collection of these constraint patterns can be effectively, easily and compactly expressed directly on the UML class diagram with a minimal amount of additional notation. In fact, according to the OMG standard [1], the extension mechanisms embodied in UML were designed for this very purpose.

We begin with a discussion of *inter-relationship* constraint assertions—how they are formulated, and at what point they are captured—leading into a description of the taxonomy of constraint patterns expressible as propositions of some number of Boolean variables. We present six specific constraint patterns in this paper, using a domain example to show how the constraint patterns can be recognized and annotated on the UML class diagram. We also contrast and compare the various means for expressing the constraint patterns with those that are currently in use with UML notation, pointing out some of the inconsistencies that exist with the current UML formulations when compared with the semantics of propositional logic.

II. OVERVIEW OF SEMANTIC CONSTRAINTS

A. Defining Constraints

In general, invariant constraints are logical assertions stated about some domain of discourse that typically restrict or constrain the set of allowable states available to an object or piece of data, and the relationships in which object instances participate. Constraints are defined as a result of making observations about the domain—such as when reviewing documents to formulate business rules, or when interviewing a subject-matter expert by a knowledge engineer.

When we use the term "constraining assertion", we mean either some declarative statement about what constitutes "allowable" information in the domain, or some statement about explicit information patterns that are allowed (meaning all others that do not match the assertion are violations). There are two means for conveying a constraining assertion, as follows [24]: (1) *extensional* - explicitly enumerating all of the allowable states that may be held by an artifact of data; or, (2) *intensional* - implicitly restricting the artifact to those states covered under some logical expression of what constitutes a valid interpretation of the domain semantics.

When we use the term "correct state", we mean the states of the information system in operation that are consistent with some *a priori* specified definition of what states or sequences of states are valid for the instances of classes in the model domain when quiescent. The information system's state may be represented in its collection of stored values, such as in program variables or database tuples. This data is operated on by executing artifacts—objects, stored procedures, software agents, and the like. The states of the information system in operation are consistent with *a priori* specified definitions when there are no transitions that are possible that place the system at odds with the specified definitions.

We use the term semantic constraint to mean that we wish to constrain the possible interpretations of declarative statements written in some representation that may be written and read by humans. Thus, given a declarative specification—such as created for an arbitrary domain using UML—we wish to achieve the following objectives: (1) limit the number a ways that someone reading the model can interpret what is stated; and, (2) provide a basis for preventing the operationalized system built on such a model from reflecting some state of the domain that was not intended by the parties specifying the system's underlying semantics. The execution semantics of programs is not discussed further in this paper.

The problem of constraint capture and representation is common to database engineering/conceptual modeling [6, 16,19], object-oriented analysis and design [20,21], and agent-oriented software engineering [22, 23] activities, and has been discussed in many contexts in the literature of these areas in recent years.

B. Semantic Clarity to Avoid Ambiguity

Our work in constraint modeling and its taxonomic treatment is based on method research using the Entity-Relationship (ER) method and its variants. The work in this paper is motivated by our background in using static, class-oriented modeling formalisms to capture conceptual structure and semantics in a variety of domains. Many modeling problems we have observed in teaching the ER and UML notations, as well as our experience in using these methods in industrial analysis and modeling projects, point to the issue of representation ambiguity as a principal cause of problems when using these diagrammatic notations to convey the semantic intent, and consensus "world view", within a given enterprise or domain of discourse.

Simply put, semantic ambiguity occurs when an assertion, by its absence in the statement of assertions about a domain, in whatever form expressed, can be interpreted as either being "true" or "false". It is in this inability to discern precise meaning where many systems and software program failures occur—in that a system programmer, in the absence of explicit information otherwise, non-deterministically makes a programming assumption one way or the other when presented with an ambiguity—often by default and often

without realizing they are doing so—thus affecting the correctness of the system.

Our objective is to mitigate the ambiguity problem in this modeling and analysis endeavor by using a pre-defined set of constraint patterns—with their associated notation and semantics—to deepen the level of precision afforded to the domain modeler/analyst when capturing domain semantics and requirements. We seek to add precision and representational power through a minimal cost of new notation and syntax. In addition, we have found that through a simple set of pattern cues—using the tools of Boolean logic—we can improve the analysts' ability to recognize such patterns, so as to better guide them in the analysis, interviewing and knowledge engineering processes. The UML constraint extension mechanism in the OMG standard has been designed with this objective in mind [1].

The rationale for this objective is simple: the use of OCL and other formal specification languages is a trade-off in richness of representation (the epistemology of what can be stated) versus the ease of use of these languages as a "mediating" means to capture and express domain semantic details between humans—such between a domain expert and a knowledge engineer [7]. OCL provides a means to express arbitrary constraints in a full predicate calculus or in other logics in precise terms—for purposes that seem to favor their use in automated tools for theorem-proving [8] or automatic constraint checking [9]. Although the OMG specification [1] and Warmer et al. [2] state that OCL has been designed specifically for ease of use, as opposed to specification languages such as Z [10], it has been noted that it is often difficult for humans to use OCL or general logic notations, which limits their use in practice [11,12,13].

Some researchers attempt to overcome this problem through defining a task-specific model with specialized tool interfaces to capture certain types of constraints prevalent in the task domain [14]. Other work focuses on adding additional first-class meta-model elements to the class diagram in order to capture a greater set of invariants through visual depiction in Constraint Diagrams built on set-theoretic notions, with appropriate meta model extensions, rather than through specification using a formal logic language notation alone [12, 15]. Finally, some research has been done in using UML notation available in other views of a model under development, such as Collaboration diagrams, to capture constraints in a mix of OCL text and graphics [13].

C. Taxonomy of Constraint Categories

There are different categories of invariant constraints, where a category has to do with the types of logical statements that are made. In [6], we presented a partial taxonomy of these, stating that the use of cardinality (or multiplicity) constraints, inter-relationship constraints, correspondence constraints, and value integrity constraints have different formulations. Many of the examples we observe in the literature for using OCL [1,2] and other formulations [12,13,15] are based on stating what we have referred to as *value integrity* constraints, which might limit the range of values that can validly be held by attributes, that define dependencies among attributes with certain values, or that restrict an attribute to a subset of values for either inclusion or exclusion, for example.

Most discussions of invariants in the OCL literature for constraining complex correspondences within class diagrams are in terms of the navigational syntax of the language [2,18]. While such complex formulations of invariants requires this capability, we are interested in formulating the semantics and notation of a weaker, yet widely prevalent, form of invariant using the tools of propositional logic: Boolean truth tables, K-maps, and Boolean algebra. In the examples of *inter-relationship* constraints that follow, we will show how the patterns can be easily visualized using the K-map, and also how use of a simple truth table by an analyst insures that the proper constraint pattern is selected, and correct meaning is insured.

In Figure 1, we depict the meta-model of the type of constraint that is the focus of this paper—specifying a set of constructs and recognition cues for representing constraints between UML relationships of all types, to help the conceptual analyst in capturing domain knowledge. The *inter-relationship* constraint category includes patterns of relationships related to an arbitrary class—where important semantics involving more than a single relationship between one or more classes exist whose interpretation should be formally yet easily constrained.

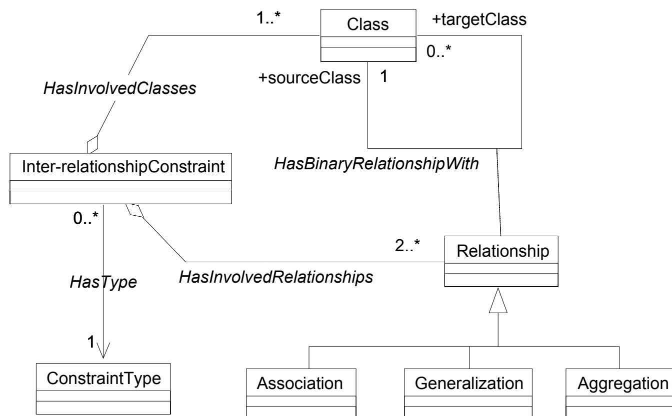


Fig. 1. UML meta-class diagram for the inter-relationship constraint pattern.

Inter-relationship constraints indicate propositions existing between a given source class and one or more target classes to which it is related. It is recognized as an important category of invariant in [1,20,21,], but it is not formalized. These govern the set of object instances that may participate in a relationship, based on whether an instance also participates in other class relationships at the same time.

However, in our formulation of this type of invariant, the occurrence of constraints across relations in a class diagram is independent of the type of relationship; it applies equally well to association, generalization, aggregation, composition and

shared-subclass types of relationships, as defined in the context of the UML meta-model [1]. This separation of constraint formulation using Boolean propositions across multiple relationships from the type of relationship links involved will separate the semantics of the inter-relationship constraint from that of the type of relationship. A relationship is not a formal meta-class in the UML meta-model [1]; however, we have found it instructive to show it explicitly as such in our meta-model, to illustrate the utility of the inter-relationship constraint pattern for all types of relationships—including associations, generalization and aggregation relationships.

We can gain insight into the general structure of the inter-relationship constraint pattern through depiction on a meta-class diagram (Fig. 1). Generally speaking, classes participate in relationships with other classes. If we were to take any arbitrary class in the domain, we'd be interested in looking for occurrences of inter-relationship constraints that indicate hidden semantic assertions across relationships in which the given class participates. As such, we'd proceed through all the classes in the diagram—looking at the relationships it has with other classes—to identify occurrences of these patterns in the domain.

From Fig. 1, it makes sense to think of the originating class (for purposes of examining the relationships) as the "source" class, and to think of all the classes to which this given class is related (regardless of the type of relationship) as "target" classes. Each relationship link between the source class and its target classes (which could include itself in a recursive association) is a specified relationship that is of interest in the domain. Each relationship could have role names and multiplicity values specified for it (although some of these constructs are implicit—such as multiplicity on a Generalization relationship—and are not depicted).

D. *Process of Capturing Constraints*

During the modeling and analysis process, the human analyst seems to progress through a series of steps, as depicted in the Fig. 2.

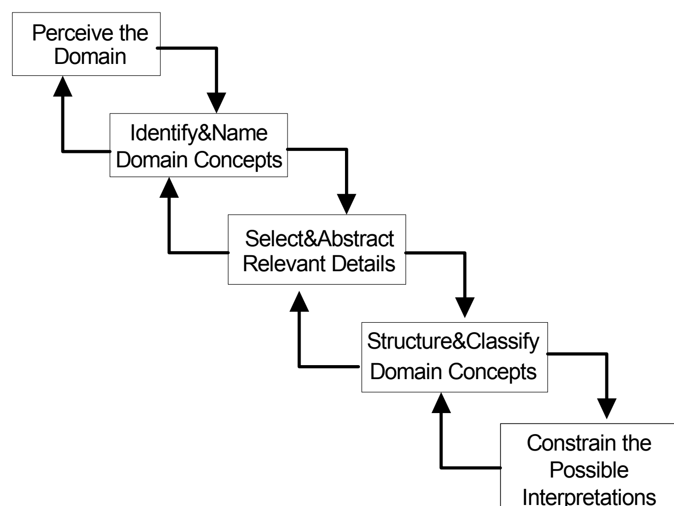


Fig 2. Sequence of Conceptual Tasks in Domain Modeling.

We can think of the process of constructing a class model in the following terms. The analyst observes the domain of discourse, as a consequence of interviewing subject matter experts, reading documents, and performing Use Case analysis. The analyst constructs various mental and written lists of the important concepts that present themselves as a result of analysis. Special attention is paid to terminology and meaning. The relevant concepts and relationships are selected from the broader set of artifacts uncovered, based on the scope of the system. The analyst structures and organizes the concepts and relationships into a series of class diagrams. These are used as input to other parts of a UML-based analysis process—such as Use Case refinement, construction of sequence diagrams, and the like.

One additional step that we have found valuable is the explicit examination of a class diagram for the occurrence of constraint patterns lying hidden among the classes and their relationships of the class diagram. Since we have a reasonably finite set of patterns for the inter-relationship constraint category, this activity can be easily managed during the interviewing process. We could, however, have complex formulations of such constraints, based on having many relationship links relating a source class to one or more target classes—corresponding to an inter-relationship proposition consisting of some number *n* propositional variables, where *n* is the number of relationships involved in the constraint formulation. In the next section of this paper, we walk through each type of Inter-relationship constraint pattern.

III. INTER-RELATIONSHIP CONSTRAINT PATTERNS

In this section, we discuss the particular patterns for the Inter-relationship category of constraints. The example used in this discussion is of a specialized security management system for a UNIX®[1] server environment, on which we have a vertical application system, such as that built using the Oracle®[2] DBMS. As such, we have an environment where the principal artifacts of the UNIX® operating system are augmented by specialized artifacts defined in the vertical turnkey application environment. Many of these artifacts are concerned with implementation of access control policies that are part of a security model. This model could be encapsulated by a set of objects and operated on by a collection of stored procedures, or software agents, for instance. The specifics of this environment are outside the scope of this paper. What is relevant, however, is that it is an interesting, real-world and non-trivial domain about which we may discuss the constraint patterns.

A. *The XR Constraint Pattern*

Figure 3 depicts a portion of an enterprise schema of the UNIX® system with the Unix File class and its specialized

[1] UNIX is a registered trademark of SCO, Inc. Solaris is a registered trademark of Sun Microsystems, Inc.
[2] Oracle is a registered trademark of Oracle Corporation.

subclasses: Directory, Regular File and Special File. We can capture the following assertions, represented in the multiplicities defined on the links of the subclass relationships.

each Directory IS_A exactly_one UnixFile.
each UnixFile IS_A (may_be) at_least_zero and at_most_one Directory.
each RegularFile IS_A exactly_one UnixFile.
each UnixFile IS_A (may_be) at_least_zero and at_most_one RegularFile.
each SpecialFile IS_A exactly_one UnixFile.
each UnixFile IS_A (may_be) at_least_zero and at_most_one SpecialFile.

Fig 3. Class hierarchy for UNIX file types - XR constraint pattern example.

This implies that every Directory, Regular File, and Special File instance must be a Unix File instance. However, a given Unix File may be a Directory, a Regular File, a Special File, or any combination thereof, or none of the three. However, in Unix®, the actual semantics of this scenario is that a given file instance is one and only one of the three types. Furthermore, the set of files is partitioned into three disjoint sets--directories (which contain other files), "regular" files (source or executable programs, text, etc.), and "special" files (UNIX abstractions for physical devices, such as disk, TTY or other thin-client terminals, etc.). This assertion cannot be captured using the multiplicity information alone. We define the XR constraint pattern to capture the additional constraining assertions, indicated as follows.

An instance of UnixFile must_be a Directory, or a RegularFile, or a SpecialFile.
a UnixFile may_not_be_a Directory and a RegularFile and a SpecialFile at the same time.
the UnixFile class consists of at_least one Directory, and at_least one RegularFile, and at_least one SpecialFile.

The XR constraint pattern represents a collection of propositional logic assertions, of which only the two-variable one is effectively named and identifiable without use of a truth table (i.e., the "exclusive-or" function [3]). The formal definition of this 3-variable pattern is shown in the truth table below. Each relationship link has an assertion and each is represented as a proposition with a given variable name. A truth table can be constructed based on whether the given combination of propositions is valid in the UNIX domain. If so, then the logic function with those inputs is "True"; otherwise it is "False".

p = UnixFile IS_A Directory
q = UnixFile IS_A RegularFile
r = UnixFile IS_A SpecialFile

$$XR(p,q,r) = ((p \wedge \neg q \wedge \neg r) \vee (\neg p \wedge q \wedge \neg r) \vee (\neg p \wedge \neg q \wedge r))$$

p	q	r	{XR}
F	F	F	F
F	F	T	T
F	T	F	T
F	T	T	F
T	F	F	T
T	F	T	F
T	T	F	F
T	T	T	F

Note that the combinations of inputs having a "True" response correspond to the semantic assertions presented in this example. This propositional function for XR with three variables is also shown as an equation. Note that the XR constraint pattern has a different logic equation, depending on the number of propositional variables.

$P \oplus Q \Leftrightarrow P$ {XR} Q "Exclusive-OR"
For the two-variable proposition case.

P {XR} Q {XR} R
For the three or more variable proposition case, the {XR} constraint is no longer equivalent to the propositional Exclusive-OR.

$P \oplus Q \oplus R$ "Exclusive-OR"
For the three-variable proposition case, this forms the "checkerboard pattern".

P {XR} Q {XR} R {XR} S

$P \oplus Q \oplus R \oplus S$ "Exclusive-OR"

Fig 4. K-maps for the 2, 3 and 4-variable {XR} constraint patterns.

For instance, the XR function for 2 variables (also shown in Fig. 3, for the Block Device class and subclasses) corresponds to the "exclusive-or" mentioned earlier. Although the meaning of the proposition doesn't change with the number of specific assertions, the specific logic function itself changes—based on the number of propositional variables [3]. The nature of this pattern becomes clear when we represent the propositions graphically using K-maps [25].

In Fig. 4, we show the K-map depiction for the 2, 3, and 4-variable propositions corresponding to the "exclusive-or". In the 2-variable case, the {XR} is the "exclusive-or" function of propositional logic. However, when we get to three or more variables, this is no longer the case, as shown in the K-maps for 3 and 4 variable XR patterns. The implication of this difference is as follows: a 3-variable "exclusive-or" for an association, using the {xor} constraint as presented in [20], doesn't yield the same logic semantics. In fact, we might even state that, because of this difference in possible semantic interpretations, that the {xor} constraint as defined in [20] is ambiguous.

The pattern of ones in the K-map cells for the 3-variable case corresponds to logic expressions and the truth table shown previously for this example. The K-map has the additional advantage that it can be used to find minimal coverings for arbitrarily complex propositional assertions [25], leading to more-compact, yet equivalent expressions for semantic constraints—an aspect of our work that is outside the scope of this paper.

Visually, the pattern of ones and zeros in the K-Map (zeroes are not shown for clarity), corresponding to true and false values of the {xor} proposition in UML, given the combination of terms related to UML relations, is referred to as a "checkerboard" pattern in digital logic [27]; however, the corresponding K-Map pattern for the {XR} constraint consists only of the first row and first column of the checkerboard. This pattern of ones in the K-map implies that the subclasses are mutually exclusive, whereas the zero in the left hand corner of the K-map implies that the subclasses are collective exhaustive, i.e., if you take the set union of all the objects in the subclasses, you obtain all the objects in the superclass.

We can formulate this constraint using other means, as defined in the UML meta-model [1] or by using OCL directly [21]. The prescribed means of capturing this constraint pattern in UML with predefined constraints is by using the *complete* and *disjoint* constraints together on the subclass link [1]. However, a formulation using Boolean logic provides the tools to more easily recognize the pattern. Furthermore, applying the propositional function is orthogonal to the types of relationships in the class diagram—generalization or otherwise. The semantics are more precise and consistent when using the {XR}, and are immediately verifiable using techniques that are manageable by any analyst with a background in discrete math or digital logic. Furthermore, using the visual K-Map provides an easy means of verifying the precise semantics of compound propositions placed on links across relationships.

B. The Inclusive OR {OR} Constraint

In Fig. 5, we have several important domain classes identified, along with their implied semantics, as follows: users of the UNIX system and its vertical applications; processes started and owned by users in this environment; special subclasses of users having certain access privileges (application developers, system administrators and application end users); installed applications of the vertical business suite that have identified application users; and, directory files that serve as the base directories for the installed applications.

There are three assertions represented in the generalization. Every Developer, System Administrator, and Application User instance must also be a User instance, and a User may be one of these, none of these, or any combination thereof. The familiar {OR} constraint is added to this structure, in terms applying the "inclusive-or" logical operator. This can be expressed using the *overlapping* constraint for generalization [1], with the *complete* or *incomplete* constraint specified, depending on whether all the subclasses have been specified or not. However, the {OR} constraint captures the full semantics as a single invariant, rather than as two.

Note also that an {or} constraint is sometimes defined and used across arbitrary association links in UML class diagrams [20, 21], but it seems that—for such examples cited—the real intent is to describe the exclusive-OR proposition as opposed to the inclusive-OR proposition [3]. We do not show an example of this contrast in this paper, but simply note further ambiguity in the literature in passing.

The {OR} constraint is shown as an edge placed across the links of the generalization hierarchy in Fig. 5. It is also shown below in terms of proposition form and its corresponding truth table. This constraint has the same logic connotation with any number of propositional variables, and corresponds to the inclusive-OR logic function [3].

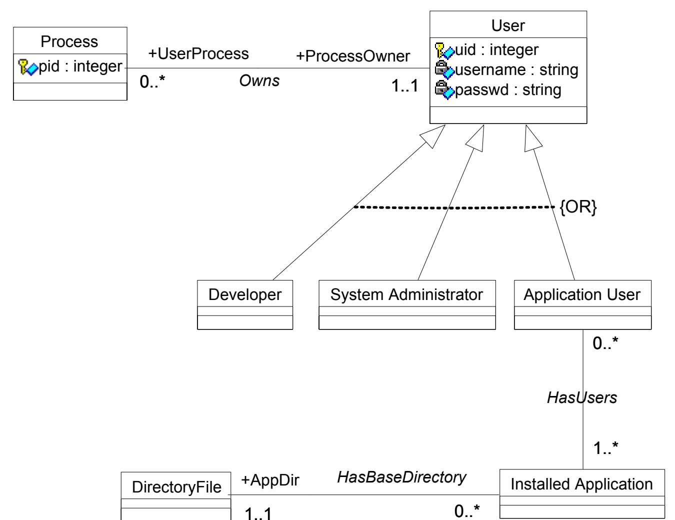


Fig 5. Class hierarchy for UNIX user types - OR Constraint Example.

a User may_be a Developer or a SystemAdministrator or an ApplicationUser, or any combination thereof.
the User class consists of at_least one Developer, one SystemAdministrator, and one ApplicationUser class instances.

p = User *is-a* Developer
q = User *is-a* SystemAdministrator
r = User *is-a* ApplicationUser

OR(p,q,r) = p ∨ q ∨ r

p	q	r	{OR}
F	F	F	F
F	F	T	T
F	T	F	T
F	T	T	T
T	F	F	T
T	F	T	T
T	T	F	T
T	T	T	T

C. The ND Constraint Pattern

Fig. 6 presents the similar class diagram as shown in Fig. 5, except that the constraints across the relationship links are different. Specifically, the User class has the {ND} constraint pattern on the Developer and Application User subclasses, and an {IP} constraint across the Developer and System Administrator subclass links. In this view of the enterprise, a User may be a Developer, Application User, or a System Administrator, but cannot be classified as both a Developer and an Application User simultaneously.

p = User IS_A Developer
q = User IS_A ApplicationUser

ND(p,q) = ¬(p ∧ q)

p	q	{ND}
F	F	T
F	T	T
T	F	T
T	T	F

A developer has different access privileges to the underlying system, and the {ND} constraint is required to succinctly make this assertion. Thus--from the standpoint of a security access control policy rule--a User can be classified as a System Administrator only if he existed also as a Developer. In other words, a system administrator has a developer's access privileges to the system, in addition to other privileges not given to developers--implying the general notion of an access policy "ring"[3] [28].

The constraints depicted in Fig. 6 provide a different interpretation than that described for the same configuration of classes in Fig. 5. Namely, we choose to express a different meaning on the generalization relations between the subclasses and the User class, reflecting a different policy ring constraint.

We have two different constraints being applied to the generalization links. The first is the {ND} constraint, which captures the following assertion not captured using standard generalization. The multiplicity values on the relationship are the same as in Fig. 5, and the semantics of these are the same as discussed for the {OR} constraint. However, when the {ND} constraint construct is applied across these links, we assert the following constraints.

An instance of User *may-be-a* Developer *or* ApplicationUser *or* neither.
A User *may-not-be-a* Developer *and* an ApplicationUser at the same time.

For the {ND} pattern, the logic function for this two-variable case reduces to the NAND (Not AND) function of propositional logic [3, 17]. When the pattern consists of greater than two variables, i.e. a constraint across more than 2 relationships, the logic function derived from this semantic constraint will be different, and is one that has no defined pattern name in Boolean logic as used in discrete math [3] or digital logic design [17].

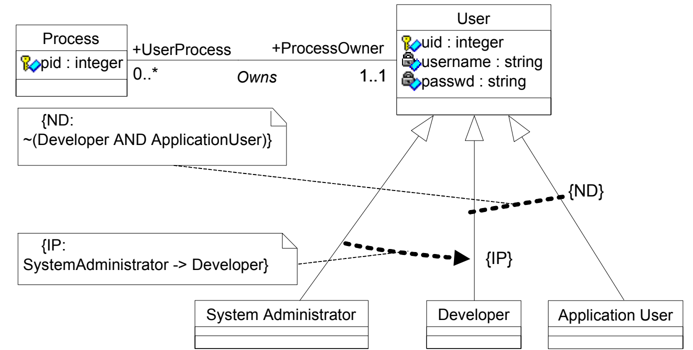


Fig 6. Class hierarchy for UNIX user types – Joint ND and IP constraints example. This shows how orthogonal constraints can be applied across generalization links. For the IP constraint, the direction of the arrow indicates the direction of implication, from antecedent to consequence.

In the 2-proposition variable case, a User specializing into one of the two subclasses is *mutually exclusive* (a given User instance can be sub-classed to only one of the classes at a time under the constraint), and *non-empty* (neither of the subclasses is empty). We show how this {ND} pattern is different than the logical NAND for more than the 2-variable case in the K-maps of Fig. 7.

From the K-map, note that the pattern for the {ND} logic proposition consists of the same first row and first column configuration of logic-ones as the "checkerboard" identified with the {XOR} propositional constraint, implying that the subclasses of {NAND} links are mutually exclusive. However, the placement of a logic-one in the upper left hand corner of the NAND K-map (unlike the XOR K-map) implies that the subclasses are not collective exhaustive with respect to the User superclass.

D. The IP Constraint Pattern

Continuing our discussion with patterns using Fig. 6 again, the {IP} constraint shown makes a different business rule assertion or security policy statement that is to be captured in the class diagram for later enforcement in the object model, as

[3] The notion of an access control policy "ring" is based on providing levels of protection during execution of processes in an operating system, where each executing process runs at a particular ring level. The lower the ring level, the more access a process has to system resources, yet its actions are less protected. The higher ring numbers are more restrictive, are also more protected, and subsume the privileges of policy rules residing at lower ring levels [28].

stated in the discussion for this additional proposition adorning Fig. 6.

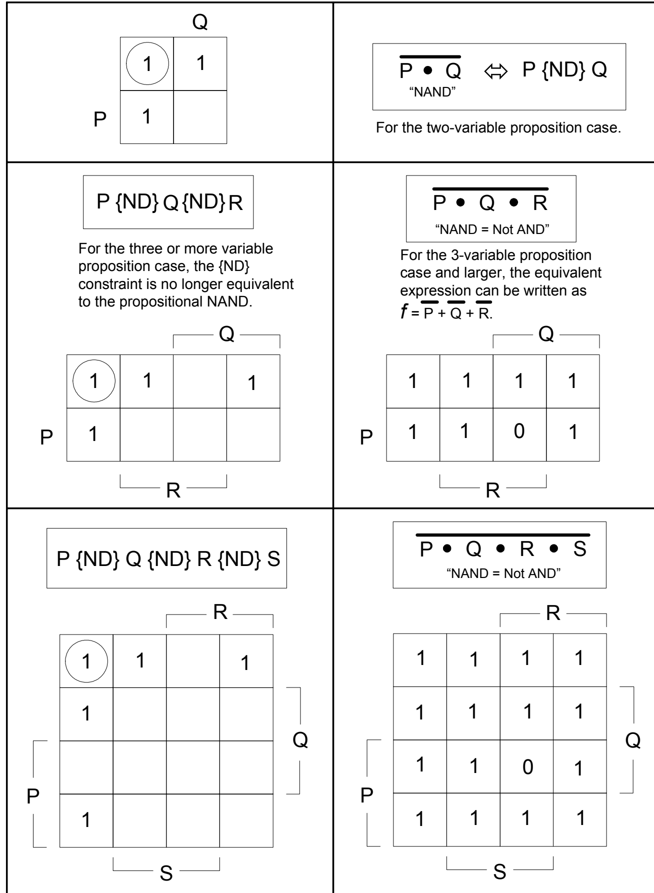


Fig 7. K-maps for the 2, 3 and 4-variable {ND} constraint patterns.

In the implementation of this system's access control policy, a User can be classified as a System Administrator only if that instance of User is first classified as a Developer (i.e., a system administrator has a developer's access to the system and its resources as a precursor to it having other access privileges). The corresponding propositional representation for the IP constraint for two variables is shown below.

IF a User *is-a* System Administrator *THEN* User is-a Developer

p = User *is-a* SystemAdministrator
q = User *is-a* Developer

IP(p,q) = p ⇒ q

p	q	{IP}
F	F	T
F	T	T
T	F	F
T	T	T

The two variable case of the IP constraint reduces to the *implication* operator in propositional logic. The case for more than two variables has the general form that only one of the assertions is the conclusion, and all other assertions connected in the UML diagram with the {IP} marker are the rule antecedents. The antecedents are, by default, assumed to be logically AND'ed together. However, explicit logical operators could be used to express more subtle semantics. In our use of this IP constraint pattern in teaching and industrial object modeling, we have not encountered modeling situations where such connectives were required in *inter-relationship* constraint capture.

E. Constraint Patterns Applied to Attributes

In Fig. 8, we see two attributes defined on the UnixFile class. These convey the information that each UNIX file has "inode" and "file header" structures, and establish a dependency condition of an object being a file on whether it has inodes and file headers (which are identified by respective 'file_header' and 'inode' attributes).

However, it might be possible to create the instance of the UnixFile before instances of the component File structure (with its file header) and the inode in the directory structure are both created. Therefore, we might choose to indicate that the 'file_header' and 'inode' attributes require a logical EQ constraint between them.

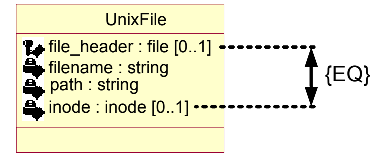


Fig 8. The EQ constraint pattern between two class attributes.

In this case, we would add a constraint across the attributes of the UnixFile class as an {EQ}, with the following meaning:

a UnixFile has exactly one Inode *if and only if* the UnixFile has exactly one FileHeader.

p = UnixFile HasA Inode
q = UnixFile HasA FileHeader

EQ(p,q) = (p ∧ q) ∨ (¬p ∧ ¬q)

p	q	{EQ}
F	F	T
F	T	F
T	F	F
T	T	T

This {EQ} constraint expression could conceivably be used to specify triggers or other operations within the system, such that if one or the other type of component is created on invoking a method to create it, the other one should be invoked in turn, regardless of ordering. This EQ of 2-variables is the "equality" propositional function, sometimes also referred to as the "IFF" (if and only-if) function. In this case, we have specified a constraint between the two attributes of the class rather than across relations among different classes.

F. Constraint Patterns Applied to Categories

In the database literature, there is an additional structural modeling construct used in conceptual modeling known as the *Category* [16,26], an extension to the E-R model that is currently not part of UML.

Using categories, an analyst can arbitrarily group instances of different classes into a common class based on the shared role the instances of these classes play in context of relationships with another class. This is shown by example in

Fig. 9, where we model the assertion that both users and the system itself may be owners of processes executing in the system (as might be shown by entering the "ps –lae" command at the Unix C-shell prompt).

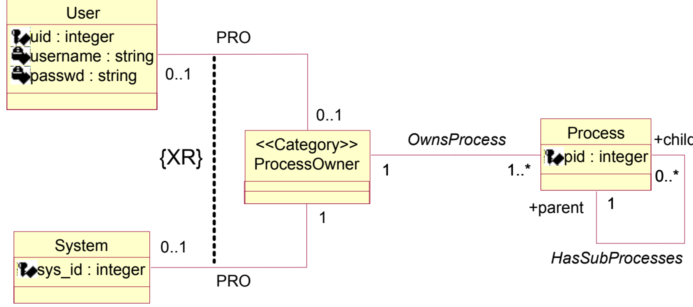


Fig 9. Placing constraints across a category.

A User has *partial participation* in the category Process_Owner (0..1)
The System has *total participation* in the category Process_Owner (1)
The {XR} constraint states that the role-playing classes User and System are *mutually-exclusive*, and the multiplicities to the category state that the role-playing classes are not *collective-exhaustive*.

Extending the concept as defined in [16], the category is created such that its instances comprise a subset of the union of the instances of the constituent classes forming the category. We represent categories as stereotypes of Class and the *PRO* (*plays role of*) links as relations, and model the relationships between the role-playing classes and the category class.

The use of categories is subject to the same semantics defined in the OMG standard [1] with regards to the construction of the identifier set, its defined domain, and the constraints that exist regarding selective inheritance along a hierarchy to which the category is bound.

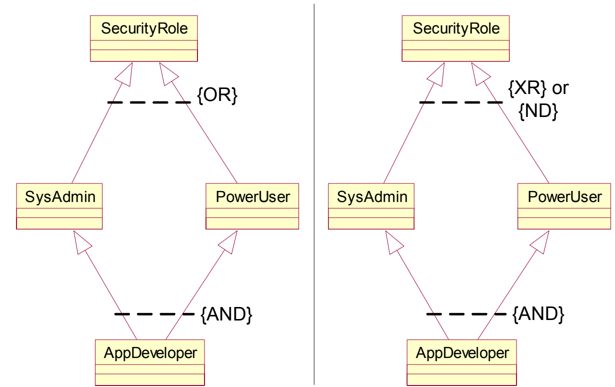
In the case of defining a category to model the role of being a "process owner" (and thus, some shared responsibilities that would likely be captured in other views of the Unix domain model), we also place an occurrence of an {XR} constraint, to indicate that 'user' and 'system' occurrences in the category are disjoint.

Note that we might choose to model this particular example using the standard {xor} constraint. There are several reasons why we would use the {XR} propositional constraint instead: (1) the constraint, when combined with the denoted multiplicities on the PRO links, completely specifies the semantics, (2) there would be no ambiguity in the meaning if we were to increase the number of role-playing classes for the category beyond the 2-variable case (which would yield incorrect semantics for the {xor} constraint, as discussed earlier).

G. Constraint Patterns Applied to Shared Subclasses

In the object-oriented literature, there is an additional structural modeling situation to which the inter-relationship constraint patterns can be applied. When modeling a generalization hierarchy, there is the possibility for elucidating subtle semantics by applying the constraint patterns to a shared subclass structure. This is illustrated in Figs. 10a&b, showing a shared parent class of two subclasses, which also, themselves, share a subclass.

Object-oriented programming languages such as C++ and interface specification languages such as CORBA's IDL support some measure of multiple inheritance. Without getting into the details of inheritance and polymorphism for a given language formalism, we can say that it is possible to represent a hierarchy in which we have a base class from which subclasses are drawn, and that these subclasses have a shared subclass.



Figs 10a&b. Placing constraints across a shared subclass hierarchy.

In some Unix security domain, this might be modeled as follows: SysAdmin and PowerUser are derived classes from the SecurityRole base class, and that the AppDeveloper class is a derivation of these two classes. The semantics being captured is that we wish to have a derived class inherit a common set of properties (attributes and behaviors) from its two parent classes, possibly adding some unique extensions that are relevant in the domain of discourse.

We model inter-relationship constraints along the subclass links from the base class, SysAdmin, to its derived classes, specifying the invariant of participation of instances of the base class as instances pf its subclasses. In Fig 10b, we have an instance of the {XR} constraint placed across the subclass links, indicating that a SecurityRole may be a SysAdmin or a PowerUser, or both, but not neither (i.e., the base class instance cannot exist independently of one or more of its subclass instantiations, as created via a *constructor*, *factory* or other means).

Also, we have placed an additional {AND} constraint across the shared subclass AppDeveloper, indicating that instances of that derived class must be instances of the type sets for the two subclasses derived from the base class. Thus, instances of the shared subclass will have the accumulation of inheritable properties of both its parents.

In Fig 10a, the two constraints—the {OR} across the base class subclass links and the {AND} across the links from the shared subclass back to its parent classes—complement one another and do not conflict. However, in Fig 10b, if we had specified one of the other possible constraint patterns across the base class to derived class links, namely the {XR} or {ND} patterns, we would have created a conflict in the propositions as stated. Specifically, we would be stating that only one or the other of the derived classes from the

SecurityRole base class are possible for a given type set instance, which violates the invariant that the AppDeveloper must inherit properties from instantiations of both its parent classes which, if only one or the other is possible (the {XR} constraint), or if none is possible (the {ND} constraint), then we have a propositional contradiction when considering the assertions posed by both logical expressions, as shown below.

```
p = SecurityRole may be a SysAdmin
q = SecurityRole may be a PowerUser
T <= XR (p, q)
r = AppDeveloper is_a SysAdmin
s = AppDeveloper is_a PowerUser
T <= AND (r, s)
```

We have found the use of the {AND} constraint pattern to be useful in aiding students and clients to more clearly understand the nature of the semantics across some inter-relationship modeling scenarios—although in many instances, the information added through this constraint is already implicit in the multiplicities of the generalization links. However, the general pattern of shared subclasses has been a place where its use, when coupled with other propositional constraint patterns, is beneficial to the domain analyst or knowledge engineer to identify and point out conflicts between constraint assertions that subject matter experts might wish to make about the domain of discourse.

We should note that any discussion on the semantics of inheritance and subclasses should take into consideration factors such as variability in regards to property cancellation [29], appropriate use of polymorphism [30], and other aspects of how the alignment of domains [29] can best be achieved between the analysis model and the programming model. However, we only note in this paper that such issues would need to be considered in a broader context for presenting a complete solution regarding application of invariants in complex subclassing patterns, and leave this treatment for future work.

We should also note that, whereas it is possible to derive a theory of composite constraints such as the shared subclass pattern, and how to evaluate them formally for consistency violations, such a treatment is beyond the scope of this paper. However, such treatment would be consistent with that of theorem proving of propositional assertions [31], as carried out in languages such as Prolog [32].

H. The Limits of Expressiveness

Now, we briefly address the limits of expressiveness using *inter-relationship* constraints. This is manifested in the inability to constrain specific class instances *by value* with propositional constraints, pointing out the primary limitation in capturing semantics in this fashion. It is here that we must resort to a more powerful means than propositional logic to express value-oriented constraints, such as using the predicate logic of OCL [2], with its direct support of quantified variables.

Fig. 11 presents a view of the domain indicating a user's ability to assume another user's identity as an "effective user" via the UNIX `su` command (thus acquiring that user's privileges and access control). A user may assume any other user's identity, provided the other user's password is known. In this view, a given user may assume "su" of any number of other users at a given time (provided they have given the correct authentication password for the user whose identity they are trying to assume).

Fig. 11 has an {IP} constraint asserting that a User is able to assume SuperUser privileges of some “effective” user (i.e., another known user’s identity) on a given TTY terminal device *only if* the User doing the “assuming” of identity is also logged onto a TTY device.

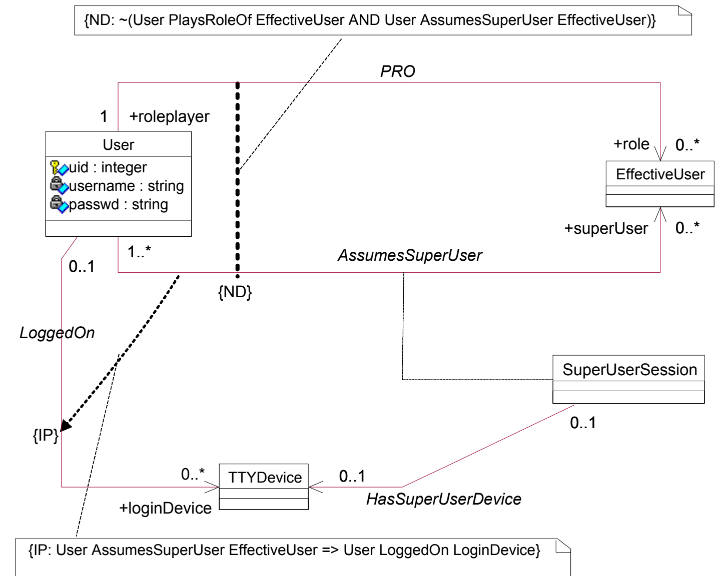


Fig 11. Example of the limits of propositional constraints.

Note that the constraint as interpreted does not state specifically that the TTY device the user is logged onto and the one on which the user has assumed SuperUser privileges are one and the same TTY device. The inability to constrain specific instances *by value* with propositional constraints points out an important limitation in capturing invariant semantics using propositional patterns. It is here that we must resort to a more powerful means than propositional logic to express value-oriented constraints, such as using OCL or other predicate logic-based formalisms, with direct support of quantified variables. It is through using universal and existential quantification that we would be able to state the necessary equality property of the invariant—namely, that the “su” session is occurring on the same terminal device onto which the “assuming” user is logged on.

IV. DISCUSSION AND COMPARISON

In this section, we discuss the constraint patterns in terms of how they compare with base UML constraint types defined in the standard and depicted in various sources [1,2,20,21]. The setup scenario for the following comparison is defined as follows:

A class X participates in relationship R1 with class Y, and participates in relationship R2 with class Z. Individual instances of the class X may be related to instances of classes Y and Z through these relationships R1 and R2.

In Table 1, we take each of the cases for interpreting the above statement as a 2-variable proposition, and we compare the existing means to represent the constraint in UML with the approach presented in this paper.

The benefits of using the *inter-relationship* constraint patterns presented herein in the capture of domain semantics are as follows:

(1) The use of these constraints is independent of the particular category (i.e., UML meta-class) of relationship, and can be applied equally to association, generalization, aggregation and composition, attributes, categories and shared subclasses. Furthermore, the patterns can be applied in modeling situations where constraints are dictated across relationships of different types.
(2) The use of these propositional logic-based patterns allows more precise semantics to be expressed directly on the class diagram, where they can be reviewed and evaluated relative to construction of objects and behaviors to enforce the constraints. In addition, analysts with truth tables and K-maps can easily check the consistency and correctness of the constraints, so placed.
(3) The use of these constraints--along with a process of explicitly looking for constraint patterns—affords the opportunity to achieve a much deeper level of domain analysis as well as creation of a much richer, more precise class model with few additional adornments. When the analyst incorporates a special analysis pass through the constructed class model, looking for such patterns as presented in this paper, he is likely to uncover semantic assertions that might have remained hidden until implementation.

V. SUMMARY

In this paper, we have presented a collection of constraint patterns—based on propositional logic—that have been useful for conceptual modeling for database, agent and knowledge-based applications, in addition to object-oriented applications. A set of examples of the patterns with 2, 3 and 4-variable propositions was presented as UML constraints on relationships of different types, including associations, aggregations and generalizations. These *inter-relationship constraints* allow clarification of certain types of ambiguities that can be present in class diagrams. They are common enough that we have found it useful to develop a set of constraints and capture them directly on the class diagram. In so doing, we facilitate greater expressiveness in the modeling and analysis of arbitrary domains without resorting to a separate set of annotations, such as by using OCL, for the type of constraints in question.

The scope of the paper was restricted to a special type of constraint known as *inter-relationship* constraints. It was shown that this is a common type of constraint, and has more patterns in which it manifests itself than has been discussed in the literature thus far [1,2,15,20,21]. Furthermore, many of the patterns are sensitive to the number of relationships across which they are applied, due to changing semantics of the

TABLE I
SUMMARY OF CONSTRAINT PATTERNS

Constraint	UML Notation	Inter-Relationship Notation and Semantics
An instance of *X* is always in both *R1* with *Y* and *R2* with *Z*.	None. An OCL constraint would be written for the specific semantic situation.	*{AND}:* We define this pattern for augmenting the semantics of multiplicity values. Useful in modeling shared subclass semantics.
An instance of *X* is either in *R1* with *Y* or *R2* with *Z*, but not both and not neither.	*{disjoint, complete}:* Only applied in generalization hierarchies, and isn't used for associations or aggregation.	*{XR}:* The Boolean logic function describing the semantics is different depending on the number of relationships involved (where each link is a propositional variable).
An instance of *X* is either in *R1* with *Y* or *R2* with *Z*, or in both *R1* with *Y* and *R2* with Z, but not in neither *R1* with *Y* nor *R2* with *Z*.	*{overlapping, complete}:* Only applied in generalization hierarchies, and isn't used for associations or aggregation.	*{OR}:* The logic describing the semantics is the same, independent of the number of relationships involved. This can be applied across any associations, regardless of type.
An instance of *X* is either in *R1* with *Y* or *R2* with *Z*, or in neither *R1* with *Y* nor *R2* with Z, but not in both.	*{disjoint, incomplete}:* Applied in generalization hierarchies, and isn't used for associations or aggregation.	*{ND}:* The logic describing the semantics is different depending on the number of relationships involved, but can be applied across arbitrary links, not just generalization.
An instance of *X* is either in *R1* with *Y* or *R2* with *Z*, or in neither *R1* with *Y* nor *R2* with Z, or in both.	*{overlapping, incomplete}:* Applied in generalization hierarchies with incomplete sub-classing. It isn't used for associations.	*None:* The multiplicity values are sufficient to cover the desired semantics. From a logical point of view, this is a tautology [3].
An instance of *X* is in *R2* with *Z* **only if** *X* is in *R1* with *Y*.	*{subset}:* invariant with the *dependency* relation between the associations; but only used for associations.	*{IP}:* Logic asserts the following: $x\ a\ y \Rightarrow x\ b\ z$, with well-defined truth values [3]. Can be used with conjunction operator.
An instance of *X* is in *R2* with *Z* **if and only if** *X* is in *R1* with *Y*.	None: An OCL expression is required.	*{EQ}:* $(x\ a\ y \Leftrightarrow x\ b\ z)$ or, alternately, *{EQ}:* $((x\ a\ y \Rightarrow x\ b\ z) \wedge (x\ b\ z \Rightarrow x\ a\ y))$, with well-defined truth values.
An instance of *X* is neither in *R1* with *Y*, nor *R2* with *Z*, nor in neither *R1* with *Y* nor *R2* with *Z*, nor in both.	N/A: This is an invalid formulation of an invariant, as it implies that the relationships have no participation from the source class.	*None:* The multiplicity values would have to be 0..0 on all relationship link targets in order for this assertion to be true. From a logical point of view, this is a contradiction [3].

formal expression of certain constraints as a propositional logic function, dependent on the number of propositional variables in the expression. This was shown to be recognizable as an explicit pattern of 1's in the cells of the K-map.

We acknowledge it is possible to model all of the propositional constraint patterns presented herein using the

OCL, where such examples have been presented for many of the standard UML constraints in [2,21]. However, our motivation in developing this taxonomy is based on finding a set of constraint patterns that: (1) occur frequently in practice; (2) are easy to understand and use; and, (3) are easy for practitioners to recognize. By defining these as additional invariants for UML class diagrams, we can support the richness of propositional logic with this abbreviated notation, and yet have practitioners easily interpret the meaning of the notation (since a large majority of computer science and engineering graduates have had introductory courses in digital logic or discrete mathematics—which is sufficient knowledge for a practitioner to read and verify these constraint patterns).

Given that we are interested in identifying patterns in the structure of a class model, the activity of "pattern-finding" is more structured, and therefore easier to do when the constraint specification mechanism consists of annotations made directly to the class diagram itself. Although not discussed at length in this paper, we put forth that incorporating the search for instances of these propositional logic-based constraint patterns during domain analysis can be used as a means to guide human-driven analysis that is more complete and effective than simply relying on the base UML constraint types, regardless of whether OCL notation is used. Exploring this activity is the subject of future work.

REFERENCES

[1] Object Management Group, *OMG Unified Modeling Language Specification Version 1.5,* formal/03-03-01, March 2003, http://www.omg.org.

[2] Warmer, J. B., and A. G. Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley Longman, Inc., 1999.

[3] Stanat, D. F. and D. F. McAllister, *Discrete Mathematics in Computer Science*, Prentice Hall, Inc., 1977.

[4] Davis, J. P., and R. D. Bonnell, "Modeling Semantic Constraints with Logic in the EARL Data Model", in *Proceedings Fifth International Conference on Data Engineering*, IEEE Computer Society Press, 1989, pp. 226-233.

[5] Behm, J., and T. Teorey, "Relative Constraints in ER Data Models", in Elmasri, R., V. Kouramajian, and B. Thalheim (eds.), *Entity-Relationship Approach-93*, Springer-Verlag, Berlin, 1994.

[6] Davis, J. P., and R. D. Bonnell, "A Framework for Constructing Visual Knowledge Specifications in Acquiring Organizational Knowledge", *Knowledge Acquisition*, Vol. 3, 1991, Academic Press, Ltd., 1991, pp. 79-115.

[7] Johnson, N. E., "Mediating Representation in Knowledge Elicitation", in Diaper, D. (ed.), *Knowledge Elicitation: Principles, Techniques, and Applications*, Ellis Horwood, Chichester, UK, 1989, pp. 179-194.

[8] Brucker, A. D., and B. Wolff, "HOL-OCL: Experiences, Consequences and Design Choices", in Jezequel, J. M., H. Hussmann and S. Cook (eds.), *UML 2002, LNCS 2460*, pp. 196-211, Springer-Verlag, Berlin, 2002.

[9] Clark, T., "Typechecking UML Static Models", in France, R., and B. Rumpe (eds.), *UML'99 - The Unified Modeling Language Beyond the Standard · Second International Conference*, Fort Collins, CO, USA, October 28-30, pp. 503-571, 1999.

[10] Spivey, J. M., *The Z Notation: A Reference Manual, 2nd ed.*, Prentice-Hall Publishers, Inc., 1992.

[11] Parnas, D. L., "Mathematical Methods: What We Need and Don't Need", *IEEE Computer*, Vol. 29, No. 4, April 1996, pp. 28-29.

[12] Kent, S., "Constraint Diagrams: Visualizing Invariants in Object-Oriented Models", in *Proceedings OOPSLA-97*, Association for Computing Machinery, pp. 327-341, 1997.

[13] Bottoni, P., M. Koch, F. Parisi-Presicce, and G. Taenzer, "A Visualization of OCL Using Collaborations", in Gogolla, M., and C. Kobryn (eds.), *UML 2001, LCNS 2185*, Springer-Verlag, Berlin, 2001, pp. 257-271.

[14] Felfernig, A., G. Friedrich, D. Jannach, and M. Zanker, "Configuration Knowledge Representation Using UML/OCL", in Jezequel, J. M., H. Hussmann and S. Cook (eds.), *UML 2002, LNCS 2460*, Springer-Verlag, Berlin, 2002, pp. 49-62.

[15] Kent, S., and J. Howse, "Mixing Visual and Textual Constraint Languages", in France, R., and B. Rumpe (eds.), *UML'99 - The Unified Modeling Language Beyond the Standard · Second International Conference*, Fort Collins, CO, USA, October 28-30, pp. 384-398, 1999.

[16] Elmasri, R., and S. B. Navathe, *Fundamentals of Database Systems, 3rd ed.*, Addison-Wesley Longman, Inc., 2000.

[17] Wakerly, J. F., *Digital Design: Principles and Practices, 3rd ed.*, Prentice-Hall Publishers, Inc., 2000.

[18] Cook, S., A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills, "The Amsterdam Manifesto on OCL", in Clark, T., and J. Warmer (Eds.), *Object Modeling with the OCL, LNCS 2263*, pp. 115-149, Springer-Verlag, Berlin, 2002.

[19] Demuth, B., H. Hussmann, and S. Loecher, "OCL as a Specification Language for Business Rules in Database Applications", in Gogolla, M., and C. Kobryn (eds.), *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools, LNCS 2185*, Springer-Verlag, Berlin, 2001.

[20] Rumbaugh, J., I. Jacobson, and G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley Longman, Inc., 1999.

[21] Gogolla, M., and M. Richters, "Expressing UML Class Diagrams Properties with OCL", in Clark, T., and J. Warmer (Eds.), *Object Modeling with the OCL, LNCS 2263*, pp. 85-114, Springer-Verlag, Berlin, 2002.

[22] Singh, M. P., "Conceptual Modeling for Multiagent Systems: Applying Interaction-Oriented Programming", in Chen, P. P. (ed.), *Conceptual Modeling, LCNS 1565*, Springer-Verlag, Berlin, 1999, pp. 195-210.

[23] Bauer, B., "UML Class Diagrams Revisited in the Context of Agent-Based Systems", in Wooldridge, M. J., G. Weiss, and P. Cianciarini (eds.), *Agent-Oriented Software Engineering - AOSE 2001, LNCS 2222*, Springer-Verlag, Berlin, 2002, pp. 101-118.

[24] Israel, D. J., and R. J. Brachman, "Some Remarks on the Semantics of Representation Languages", in Brodie, M. L., J. Mylopoulos, and J. W. Schmidt (eds.), *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Springer-Verlag, Berlin, 1984, pp. 119-146.

[25] Russomano, D. J., R. D. Bonnell, "A Pedagogical Approach to Database Design via Karnaugh Maps", *IEEE Transactions on Education*, Vol. 42, No. 4, Nov. 1999, pp. 261-270.

[26] Elmasri, R., J. Weeldreyer, and A. Hevner, "The Category Concept: An Extension to the Entity-Relationship Model", *Data and Knowledge Engineering*, Vol.1, No. 1, Elsevier Science Publishers, B.V., 1985, pp. 75-116.

[27] Peatman, J., *The Design of Digital Systems*, McGraw-Hill Publishers, Inc., 1972.

[28] Schroeder, M., and Saltzer, J., "A Hardware Architecture for Implementing Protection Rings", *Communications of the ACM*, Vol. 15, No. 3, March 1972, pp. 157-170.

[29] Coplien, J. O., *Multi-Paradigm Design for C++*, Addison-Wesley Longman, Inc., 1998.

[30] Eliens, A., Principles of Object-Oriented Software Development, 2nd Ed., Addison-Wesley Pearson, 2002.

[31] Manna, Z., and R. Waldinger, *The Deductive Foundations of Computer Programming*, Addison-Wesley Publishing, Inc., 1993.

[32] Sterling, L., and E. Shapiro, *The Art of Prolog*, MIT Press, 1986.

James P. Davis (M'80) is Associate Professor, Department of Computer Science and Engineering, University of South Carolina. He received his B.S. and M.S. degrees in Electrical Engineering, in 1980 and 1981, and Ph.D. in Computer Engineering in 1989, all from South Carolina. His research interests are in the areas of architecture and analysis methods for hardware and software systems design, and the design of reconfigurable computing architectures using FPGAs and custom VLSI logic.

During his 25 years in industry, he has developed hardware and software systems, as well as managed research and development of such systems, in a number of industry segments for a number of companies, including NCR and

Mitsubishi Electric. He has held a number of senior management positions at technology start-ups, including Knowledge Based Silicon, creating high-level VLSI CAD software, and HealthMagic, responsible for one of the first longitudinal, secure, consumer-oriented, electronic health records for the Internet.

Dr. Davis is a member of Tau Beta Pi and Eta Kappa Nu engineering honor societies, and IEEE, ACM, AAAI, and AMIA organizations.

Ronald D. Bonnell (M'80, SM'81) is Professor, Department of Computer Science and Engineering, at the University of South Carolina. He received the B.S. and M.S. degrees in mechanical engineering from the University of Kentucky, Lexington, in 1957 and 1961, respectively. He completed further graduate studies in dynamics and control at the University of Tennessee, Knoxville, from 1961 to 1965.

He held an NSF Traineeship in computers and control from Georgia Institute of Technology, Atlanta, from 1965 to 1967. He was an Engineer with IBM from 1957 to 1961, a member of the faculty of the University of Tennessee from 1961 to 1965, and a member of the faculty of the University of Kentucky from 1967 to 1970. He has served as Chairman of the Electrical and Computer Engineering Department and as Director of the Center for Machine Intelligence at the University of South Carolina.

Professor Bonnell is a registered Professional Engineer and a member of IEEE, ACM and AAAI.