Management of Evolving Constraints in a Computerised Engineering Design Environment

J.S. Goonetillake and G.N. Wikramanayake
University of Colombo School of Computing
{jsg, gnw}@ucsc.cmb.ac.lk

Abstract

Engineering design process has an evolutionary and iterative nature as design artifacts develop through series of changes before the final solution is achieved. Design is performed based on design requirements which are mainly specified as restrictions on the properties of the design artifact, and are considered to be design constraints.

A common problem encountered during the design process is that of constraint evolution, which may involve the identification of new constraints and modification or omission of existing constraints for a number of reasons. These reasons would include changes in customer requirements, changes in the technology, change in the production cost or to improve the performance. As design is an evolutionary process where a solution is sought by trial and error, integrity constraints in engineering design will not remain fixed but will tend to evolve during the design process. This evolving nature of constraints has made the support for automatic integrity validation, non-trivial as it imposes number of key issues that need to be dealt with.

We have been able to successfully address these issues with respect to the engineering design environment and we identify how one could meet the challenges through effective management of these evolving constraints for engineering designs. Thus the original design will continue to exist while supporting the creation of new design versions to meet the changes.

1.0 BACKGROUND

1.1 Version Management

As a result of the iterative and evolutionary (tentative) nature of the engineering design process, design artifacts develop through a series of changes before the final solution is achieved. Each stage of the artifact evolution can be represented through versions and it has been identified that the version management is a crucial feature that should be supported in engineering design [13]. Version management allows derivation of a new design solution from an existing design solution thus saving designer's time. For example consider that V1 is produced as the first solution for a bicycle frame design as depicted in figure 1. Although this would work, the designer may find out under evaluation that this design solution will not provide the stiffness required by the bicycle rider. To produce, a more stiffer road frame a new design solution denoted as version (V2) can be derived from V1 by changing the seat tube size and the material (see figure 1).

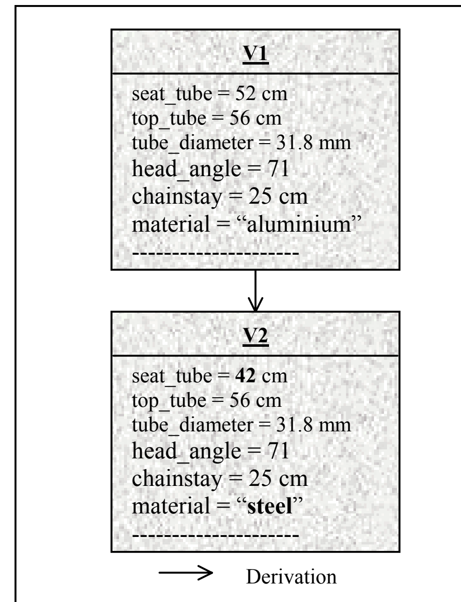

Figure 1: Bicycle Frame Versions

1.2 Engineering Design Constraints

In designing artifacts designers have to make sure that their design solutions, adhere to given sets of design constraints through an integrity validation process. To assist the designer in this task it is necessary to provide the designer with an integrity validation mechanism,

which automatically checks design solutions (represented as object versions), for design data correctness and consistency. There are number of different constraint classification schemes in the literature with respect to the engineering design environment [15, 16, 19, 21]. However, we categorise constraints based on what aspect of the design artifact they are imposed on as depicted in figure 2. In our research we consider only the value-based constraints relevant to physical and structural (i.e. formation features category) constraints that enforce the accuracy of the data values assigned to the corresponding attributes of the object.

1.2.1 Physical Constraints

To this end, we consider the following three value-based constraint classifications:

(i) ***Range Constraints*** (e.g. 35 cm ≤ frame_size ≤ 58 cm) which check the validity of an attribute value specified by the designer by returning true if the value is within the specified range and false otherwise.

(ii) ***Enumeration Constraints*** (e.g. material = {steel, stainless steel, titanium, aluminium}) which may also return a boolean after validation.

(iii) ***Relationship constraints*** are used to establish a physical relationship between two or more attributes that may belong to the same object. The validation of this constraint may either return a derived value or a boolean (e.g. in hub artefact the relationship between its spoke angle and spoke holes is given as *SpokeAngle is round (720 / SpokeHoles)* .

1.2.2 Structural Constraints

Relationship constraints are also used to establish formation features of structural constraints (e.g. the formation features of the bicycle artefact with respect to frame and the wheel which may state that the frame size should be > 40 cm if the diameter of the wheel is > 60 cm).

1.2.3 Hard and Soft Constraints

In the general design context, each constraint category can in turn be divided in to hard and soft constraints [10]. Hard constraints are restrictive and cannot be violated while soft constraints are not restrictive and can be violated if required. For example, the constraint specified on Seat tube (e.g. seat tube >52 cm and <68 cm) of the bicycle frame can be more restrictive and therefore be a hard constraint whilst the constraints specified on the frame material (material = {aluminium, steel}) can be violated and therefore be a soft constraint.

2.0 Problem of Evolving Constraints

A common problem encountered during the design process is that of constraint evolution, which may involve the identification of new constraints and modification or omission of existing constraints for a number of reasons. These reasons would include changes in customer requirements, changes in the technology, change in the production cost or the possible improvements to the performance. Therefore, it is not possible to predict in advance how frequently the constraints will change. As design is an evolutionary process where solution is sought by trial and error, it is necessary to recognise that integrity constraints in engineering applications will not remain fixed but will tend to evolve throughout the design process. We observed that this evolving nature of constraints has made the support for automatic integrity validation, non-trivial as it imposes number of key issues that need to be dealt with. These issues have been identified as given in sections 2.1 to 2.5 [11, 12]:

2.1 Designers as End-users

The designers are not (expert) programmers and may only have a little idea about the inner workings of the application programme and the underlying database system. Hence a system that takes the cognisance that designer is the end user of the system through providing easy creation, modification and deactivation of constraints is required.

2.2 Creating New Design Objects

Changes in constraints may require the creation of new design solutions or new object versions, which satisfy new constraints. The system should be able to validate the consistency of the new version on explicit user request by automatically selecting the currently applicable set of constraints. The violations should be reported to the designer through informative messages enabling him to correct them appropriately. This demands a more flexible environment, which incorporates the dynamic constraint modifications to the system for future constraint validations.

2.3 Constraint Reusability

It is unlikely that all of the constraints pertinent to an artifact will be changed in a modification. Most of the previous constraints imposed on an existing design solution (or a parent object version) are re-applicable to

the new object version and will not require re-definition. This raises the importance of constraint reusability.

2.4 Constraint Evolution History

Even though a new set of constraints is currently applicable, it is necessary to recognise that the consistency of previous versions is still valid against the set of constraints imposed on them at their creation time. Moreover, as design is a trial an error process the designers may want the facility to move back to a previously applicable set of constraints if a particular line of constraint evolution does not work out. These issues illustrate the importance of maintaining a constraint evolution history (to keep track of constraint evolution) particularly against object versions and also of providing the ability to retrieve the set of constraints applicable to each version from the evolution history.

2.5 Existing Object Validation

There can be existing version(s) that may fully or very closely adhere with the new constraints (e.g. in the event of a constraint relaxation). If an existing design version can be successfully validated against the new set of constraints, the major advantage is that it may eliminate the derivation of a new design version. This requires the capability of validating the existing versions and reporting the outcome of the validation to the designer in the event of a constraint change.

Although the main focus of any constraint management mechanism is the provision of automatic integrity validation, it is clear that more facilities will be required when evolving constraints are considered in an engineering design environment. The provision of these facilities cannot be achieved without a well-defined framework.

3.0 CONSTRAINT MANAGEMENT MECHANISMS

Three main mechanisms are used to manage constraints. They are through database schema using the relational or declarative approach, the class methods using object-oriented approach and active rules.

3.1 Static Approach

One of the most widely used approaches for constraint handling is specifying and compiling constraints in the schema definition [6, 8]. This mechanism is known as the *static approach* in [9]. The defined constraints are enforced by the DBMS to ensure the database consistency [8]. This approach has been mainly addressed in relational and deductive databases. The static approach provides *knowledge independence* [4, 10] where knowledge or rules are managed independently from the application program. The benefits of knowledge independence are that there is a substantial reduction in the procedural code necessary to fulfil the application requirements since constraints are managed by the DBMS and not by the applications [4, 5]. However, the static approach, or declaring integrity constraints in a database schema/class in general, has the following shortcomings in relation to evolving constraints:

(i) The designer (i.e. a non-programmer) requires to recompile and rewrite the existing schema to define a new schema to incorporate the constraint changes [2].

(ii) As the constraints are considered as meta data, applicable globally to all the objects of that class, it is not possible to have different objects in a particular class complying with different sets of constraints [14].

(iii) Although this mechanism is suitable for some applications to ensure the consistency of the DB state it is not appropriate for engineering design applications as it prevents constraint modifications unless all the existing objects comply with the new constraints. It further prevents the deactivation of unnecessary constraints [2].

(iv) The constraints declared in the schema are always enforced on the corresponding objects [2]. This approach may not be suitable for applications with soft constraints since the objects must always satisfy all the constraints defined in the schema.

3.2 Object-Oriented Approach

In Object-Oriented databases the normal practise is to code the database schema as a collection of classes using a programming language such as Java or C++. Constraints are coded in corresponding methods of these classes due to encapsulation [17, 22]. Although encapsulation is one of the most important characteristics of the object-oriented approach there are number of drawbacks associated with implementing integrity constraints within methods in class definitions.

(i) It can be seen that the environment is rather rigid and inflexible. It is not possible to modify constraints without affecting the class definition.

(ii) As explained in the static approach, it is totally unrealistic to expect that the designer would change the class definition to incorporate constraint changes.

(iii) Checking constraint evolution is a difficult task as it involves scanning the methods in the corresponding

class definition and furthermore, it is not easy to retrieve the set of constraints applicable to a particular version since constraints are scattered everywhere within class methods [7].

3.3 Active Rules

Constraints represented as active rules avoid some disadvantages of the static approach such as schema/class evolution by separating constraint specification from the schema. This separation allows independent constraint evolution without affecting the schema/class definition [18]. However, this approach has the following shortcomings with respect to the issues outlined in sections 2.1 to 2.5.

(i) The designer is expected to have a sound knowledge of a programming knowledge unless he is supported with a user-friendly interface to modify existing constraints. To our knowledge there is no real commercial product of this kind. It is unrealistic to expect that the designer will write and compile piece of codes relevant to the constraint changes.

(ii) Given a particular version, it is not feasible to access the constraints imposed on it to denote the various consistency states. This limitation occurs for a number of reasons:

  (a) Constraints are scattered in different operations according to events.

  (b) Constraints defined as active rules get triggered for each object based on the event conditions. There is no real association between an object and the constraints applicable on it.

  (c) There are no proper means of maintaining the history of the constraint evolution.

(iii) Active rules are procedural and in general it may be difficult for a layperson to understand them.

Figure 3 summarises how these three mechanisms cater for the issues outlined in section 2.

3.4 Constraint Management Mechanisms in Engineering Design Applications

Some attention has been given in the literature to providing constraint management mechanisms in engineering design applications. These include approaches presented in [2] and [19] that take the evolutionary nature of the design constraints into consideration. In these approaches the flexibility to incorporate constraint evolution is achieved by treating constraints separately from the class definition. To this end Buchmann [2] present constraints as database objects and Ram [19] aggregate constraints into a separate class definition called constraint meta object.

Constraints are associated with objects/instances of the class facilitating different objects of the same artifact to adhere with different constraints. However, a major drawback is that both approaches require the designer to manually associate the applicable constraints with the corresponding design objects. Particularly, the approach proposed by Ram [19] is extremely software oriented which requires the designer to change and compile programs written in C++ to capture new constraints and moreover to attach (plug) and detach (unplug) each constraint to and from the design object. In reality this is not possible when the designer is not a programmer making more hassles to the designer than the benefits. On the other hand, specifying constraints in non-executable form such as normal database objects as in [2] makes the integrity validation a difficult process from the application program's point of view. The same disadvantage is applicable to [19] since the constraints defined in the constraint class in special syntax require to be interpreted by another program. Hence, the data validation itself will become rather cumbersome under both methods. Moreover, these mechanisms do not in fact address all the issues outlined in sections 2.1 to 2.5.

4.0 THE PROPOSED FRAMEWORK

In proposing our framework, we consider that embedding dynamic constraint information in a class definition clearly requires a redefinition of that class, whenever a change occurs. It is also necessary to support the features discussed in sections 2.1 to 2.5. Consequently, we propose our framework based on the concept of the Constraint Version Object (CVO).

Each CVO contains or aggregates a set of integrity constraints that needs to be satisfied by the object version of a particular artifact, at some instant in time. For the bicycle frame artifact for example, the CVO may contain the currently applicable constraints for its frame size, tube diameter, material and seat_tube length. Maintaining CVOs provides the flexibility to capture changed constraints independently from the artifact class definitions.

Constraint evolution is managed by producing new CVOs. Each artifact will have its own set of CVOs depending on how the constraints relevant to that artifact change. However, only one CVO per artifact is active at a time; this is known as the *default CVO*. The default CVO contains the currently active constraint set, and the last CVO created in the system will usually become the default CVO. CVOs govern the validity of design data in object versions. At the time of version creation, each new version is automatically coupled with the corresponding

default CVO, for data validation. The default CVO may change with time as new CVOs are produced to reflect constraint evolution. Consequently, different versions of an artifact may be associated with different CVOs, each representing the default CVO current at the time of data validation.

A naming scheme is needed to identify CVOs uniquely within the design environment. As each design artifact has its own set of CVOs, the naming scheme that is adopted here takes the form <*objectnameCVO_number*>, which represents a unique id for each CVO. The number is assigned sequentially, indicating that the CVO with the highest id number is created last for a given design artifact. The initial framework is depicted in figure 4a. As shown in the diagram the object versions (*artifactversion1, artifactversion2* and *artifactversion3*) are instances of the same artifact class but have been validated under different CVOs (*artifactCVO_1* and *artifactCVO_2*). Any number of object versions can be associated (or validated) with one CVO. It is important that each object version keeps track of the CVO relevant to its data validation as it enables tracking of the corresponding CVO for each object version later on. Since a CVO aggregates a set of constraints, retrieval of the constraints imposed on each object version is easily achieved through its corresponding CVO. In defining constraints within CVOs, each constraint should be given a name in addition to the constraint specification, to uniquely identify that constraint within the CVOs belong to the same artifact. A CVO may contain any combination of active constraints, of any category, e.g. range, enumeration and relationship constraints, which are in turn, can be either hard or soft.

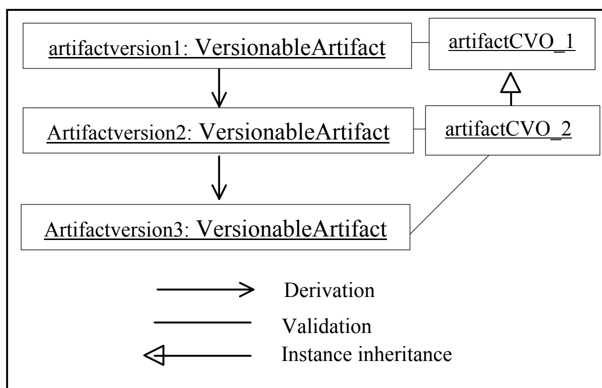

Figure 4a: Proposed Framework

Creating New CVOs

The necessity to create a new CVO may arise for the following reasons.

(i) A modification of existing constraints. For example the tube diameter of the bicycle frame artifact may change from (25mm ≤ tube_diameter ≤ 28 mm) to (29mm ≤ tube_diameter ≤ 31 mm).

(ii) Introduction of new constraints, e.g. the head angle of a bicycle frame which was not restricted before can be constrained to a particular set of values.

(iii) Omission of previously used constraints. For example moving on to free size bicycle frames means the frame size constraint will no longer be applicable.

(iv) Any combination of (i)-(iii).

A new child CVO will be defined with:

(i) refined constraints for the constraints that changed in the parent CVO.

(ii) new constraints that did not exist in the parent CVO.

(iii) a mechanism to indicate the omission of inactive constraints from the parent CVO.

(iv) any combination of (i), (ii) & (iii).

Consequently, a new CVO contains only the changes to its parent CVO constraint set, and the means of referring to its parent for the unchanged constraints. Thus, when invoking constraints for data validation from the child CVO, there should be means in the child CVO to:

- delegate constraints to the parent CVO if not declared within the child CVO.
- override constraints in the parent CVO when they are redefined in the child CVO.
- omit constraints defined in the parent CVO when they are no longer be applicable.

Since CVOs are objects, we consider that *instance inheritance* [23] with *overriding* and *differential* mechanisms is the means to best provide these features in CVOs. If a constraint is defined in a child CVO and the same constraint is defined in the parent CVO, then through the overriding mechanism the constraint of the parent CVO is not part of the child CVO. The differential mechanism allows the designer to be selective about what is inherited from the parent object and thus provides facilities to specify what should not be inherited from the parent object. Classical *type inheritance* differs from instance inheritance and does not cater for these requirements. For example in type inheritance there are problems with omitting and overriding constraints from the parent object in the child object [1]. It may further cause class evolution [14]. Within CVOs each constraint is defined as a rule where its head denotes the constraint name, and its body defines the constraint specification. A

child CVO only contains the refined constraints and inherits the unchanged ones from its parent as depicted in figure 4a.

For example, the frame version (frameVersion1) is produced within the set of constraints frameCVO_1 (figure 4b). However, to improve the comfort and to reduce the weight of the bicycle a new set of constraints (denoted by frameCVO_2) for seat tube, top tube, chainstay and tube diameter is agreed upon whilst keeping the constraints for head angle and the material unchanged. Due to inheritance the new CVO will only contain the constraints relevant to the changed constraints. A new version frameVersion2 is derived from frameVersion1 by changing the values accordingly to satisfy the new constraints. The designers may explore more versions under the current set of constraints. For example another version frameVersion3 can be derived from frameVersion2 within the constraints defined in frameCVO_2 either to improve the flexibility of the bicycle or to satisfy the rider's requirements.

5.0 ADVANTAGES TO THE DESIGNER

The Object-Oriented paradigm (OOP) and thus the OODBMS (based on OOP) have been recognised as the most suitable technology for engineering data modelling, version management and storage [3].

We have chosen to implement our model in Java2. Objectivity has been chosen as the database system to add persistence to design object and versioning data since it is an object-oriented database with rich OO features. Prolog is well suited for defining CVOs since the integrity constraints can be handled in a straight forward manner as user defined predicates in Prolog. SICStus Prolog has the advantage that it provides SICStus objects with named collection of predicates with a facility for inheritance. Furthermore, the package called jasper provides a bi-directional interface between Java and SICStus [20].

5.1 Automatic Integrity Validation

CVOs are used to check the consistency or adherence of a particular object version to a given set of design requirements, before making them persistent in the database. Accordingly in our framework, when a new object version is derived, data validation is invoked by explicit user request and is performed only on the new object version. On issue of the data validation command, the system is responsible for communicating with the corresponding default CVO and invoking the relevant constraints for validation, without the designer's involvement. In the event of a violation, the designer is provided with an informative messaging system which specifies details such as the violated constraints and the reasons for violations.

In our opinion, in an engineering design environment designers should be given the facility to correct violations, in order to maintain the autonomy of participating disciplines, rather than automating the violation correction process. As a result, in our framework, the designer repairs the violation and the informative message system helps the designer to take the necessary actions to correct it. In relation to hard and soft constraints

- violation of soft integrity constraints are merely flagged to warn the designer, while still allowing him to proceed.
- violation of hard integrity constraints should warn the designer and at the same time prevent him from proceeding any further without correcting the violation.

A new version is made persistent in the database, after successful validation of version data. Forming a new version after the successful completion of the validation process ensures the design integrity of object versions stored in the database. In our framework, two main situations are considered in which an object version data is validated. The first situation is at the time of a new version derivation. The second situation is the validation of existing versions against a newly created CVO.

When a new CVO is created which reflects a new set of constraints, the existing object versions are checked against the design constraints within the newly created default CVO. Unlike the previous case, in this situation the system itself invokes the integrity validation process on creation of a new CVO. On receipt of the designer's consent to proceed with the validation, it will carry out the validation task for the existing versions. A successful validation is feasible, for example if the child CVO contains constraints with relaxed or intersected domains. The designer will be notified of the outcome of the validation. In this way the designer finds out whether any existing versions comply with the new constraints. If there are no successful validations from the existing set of design versions, the system is capable of proposing to the designer those versions which are closest to successful validation.

5.2 Easy Capturing of Constraints

The designer should be able to capture constraints in a form-filling manner without requiring him to write or compile any program code. To this end, we consider providing the designer with a graphical interface that

enables him to enter the constraint specifications and other required information into the system, e.g. new CVO name, parent CVO name, in a form-filling manner (see figure 5). From this information, the creation of a new executable CVO for a particular design artifact is made transparent to the designer. Syntactical errors that may occur through typing mistakes are avoided, by using option buttons and lists in the graphical interface, and thus enabling the designer to select the values/parameters relevant to the constraint specifications. Text fields are used to enter information such as the new CVO name and the values/parameters for range and enumeration constraint types. One constraint specification is entered at a time. On completion of each constraint specification, a button is pressed to carry out this conversion. This is transparent to the designer. Once all the constraints relevant to a CVO are specified, the designer only has to press a button to indicate CVO completion, upon which the executable form of the whole CVO is automatically written into the system.

Figure 5: Capturing of constraint specifications

5.3 Retrieval of Design Constraints

Retrieval of design constraints is an important aspect from the designer's point of view. This becomes a trouble-free task with CVOs, since each CVO is defined separately from the schema, and aggregates a set of constraints. To this end, the designer is easily able to retrieve constraints in the default CVO as the system maintains a record of it. Retrieval of any other CVO is possible as constraint evolution history is managed through CVOs and is performed by specifying the CVO name. As the system associates and keeps track of the CVO pertinent to each version, the CVO of any required version can be retrieved by specifying the version name. Upon selection of a particular version name, the CVO name associated with it is obtained to display the CVO details. However, constraints are defined in executable form as rules in CVOs and consequently, in displaying constraints they should be converted into a form close to natural language. The implementation of this feature can be performed using a parser, which reads and converts rules in each CVO to natural language.

5.4 Moving back to Previous Stage of Constraint Evolution

The designer performs this through changing the default CVO, if one constraint evolution path does not work out. The decision as to which CVO should be made the default should come from the group/project leader. The group/project leader should communicate this decision to the corresponding team members. The default object version is dependent on the default CVO. Therefore, when the current default CVO is changed, an object version belonging to the new default CVO should be set as the default. Consequently, the operation *changing the default CVO* is associated with the *set default version* operation. To support the designer in setting this new default object/configuration version, the system, through the version manager, selects and presents the designer with the list of version numbers that are eligible to be the default.

6.0 CONCLUSION

In this paper we presented a framework to manage evolving design constraints in a computerised engineering design environment. We base our framework on the concept of CVOs which provide a flexible environment in managing evolving constraints. Our approach is novel since it both maintains the consistency of versions whilst catering for the issues that emerge as a result of constraint evolution within an engineering design environment. We believe that our proposed model is an important first step towards addressing the challenges of evolving constraint management and we consider that it can be adopted for any application with evolving constraints.

7.0 ACKNOWLEDGEMENTS

The work presented here is based on the research done by the first author at University of Wales Institute, Cardiff (UWIC), UK under the supervision of T.W. Carnduff and W.A. Gray. The research was funded by UWIC and Asian Development Bank.

8.0 REFERENCES

1) Bassiliades, N. and Vlahavas, I., 1994, Modelling Constraints with Exceptions in Object-Oriented Databases, In proceedings of the 13th International Conference on the Entity-Relationship Approach (ER'94), Loucopoulos P. (eds.), Manchester, U.K, 189 – 204.

2) Buchmann, A.P., Carrera, R.S. and Vazquez-Galindo, M.A., 1992,Handling Constraints and their Exceptions: An Attached Constraint Handler for Object-Oriented CAD Databases, On Object-Oriented Database Systems, Dittrich K.R., Dayal U. and Buchmann P. (eds.), Springer-Verlag, 65-83.

3) Carnduff, T.W., 1993, Supporting Engineering Design with Object-Oriented Databases, PhD thesis, Department of Computer Science, University of Wales Cardiff, UK.

4) Ceri, S. and Fraternali, P., 1997, Database Applications with Objects and Rules, Addison-Wesley.

5) Ceri, S. and Ramarishnan, R., 1996, Rules in Database Systems, ACM Computing Surveys, vol. 28 (1), 109-111.

6) Date, C.J., 1995, An Introduction to the Database Systems, Addison-Wesley.

7) Diaz, O., Paton, N. and Gray, P., 1991, Rule Management in Object Oriented Databases, A Uniform Approach, 17th National Conference on Very Large Databases (VLDB 91), Barcelona, 317-326.

8) Elmasri, R. and Navathe, S., 1994, Fundamentals of Database Systems, Second Edition, Addison-Wesley.

9) Finin, T.W, Nicholas, C.K and Yesha, Y., 1992, Consistency Checking in Object Oriented Databases: Behavioral Approach, Information and Knowledge Management, First International Conference (CIKM'92), Baltimore, Maryland, USA, 53-68.

10) Friesen, O, Gauthier-Villars, G., Lefebvre, A. and Vieille, L., 1994, Applications of Deductive Object-Oriented Databases Using DEL, Applications of Logic Databases, Ramakrishnan R. (eds.), Kluwer Academic Publishers, 1-22.

11) Goonetillake, J.S., Carnduff, T.W. and Gray, W.A, 2001, Integrity Validation for Object Versions in a Co-operative Design Environment, In Proceedings of the 6th International Conference on Computer Supported Cooperative Work in Design (CSCWD'01), Shen, W., Lin, Z., Barthes, J. and Kamel, M. (eds.), Ontario, IEEE, 89-94.

12) Goonetillake, J.S., Carnduff, T.W. and Gray, W.A., 2002, An Integrity Constraint Management Framework in Engineering Design, In the International Journal of Computers in Industry, vol. 48(1), Elsevier Science.

13) Katz, R.H., 1990, Towards a Unifying Framework for Version Modeling in Engineering Databases, ACM Computing Surveys, vol. 22 (4), 376-408.

14) Katz, R.H. and Chang, E., 1992,Inheritance Issues in Computer-Aided Design Databases, On Object-Oriented Database Systems, Dittrich, K.R., Dayal, U., Buchmann, A.P. (eds.), Springer-Verlag, 45-52.

15) Lin, J., Fox, M.S. and Bilgic, T., 1996, A Requirement Ontology for Engineering Design, In Proceedings of Advances in Concurrent Engineering (CE'96), Sobolewski, M., Fox, M. (eds.), Toronto, 343-351.

16) Lin, J., Fox, M.S. and Bilgic, T., 1996, A Requirement Ontology for Engineering Design, Concurrent Engineering Research and Applications, vol. 4(3), September, 279-291.

17) Meyer, B., 2000, Object-Oriented Software Construction, Prentice Hall International Series in Computer Science.

18) Pfieifer, A. and Wulf, V., 1997, Negotiating Conflicts in Active Databases, In Proceedings of the Concurrent Engineering: Research and Applications Conference (CE97), Ganesan, S. and Prasad, B. (eds.), 443-450.

19) Ram, D.J., Vivekananda, N., Rao, C.S. and Mohan, N.K., 1997, Constraint Meta-Object: A New Object Model for Distributed Collaborative Designing, IEEE Transactions on Systems, Man and Cybernetics, March, vol. 27(2), 208-220.

20) SICStus Prolog User's Manual, 2000, Release 3.8.4, Swedish Institute of Computer Science,Sweden.

21) Twari, S. and Franklin, H., 1994, Automated Configuration Management in Concurrent Engineering Projects, Concurrent Engineering Research and Applications, vol. 2(3), 149-161.

22) Urban, S., Karadimce, P and Nannapaneni, R., 1992, The Implementation and Evaluation of Integrity Maintenance Rules in an Object-Oriented Database, Proceedings of 8th International Conference on Data Engineering, 565-572.

23) Wilkes, W., 1988, Instance Inheritance Mechanism for OODBS, Object Oriented Database Systems (OODBS 88), 274-279.

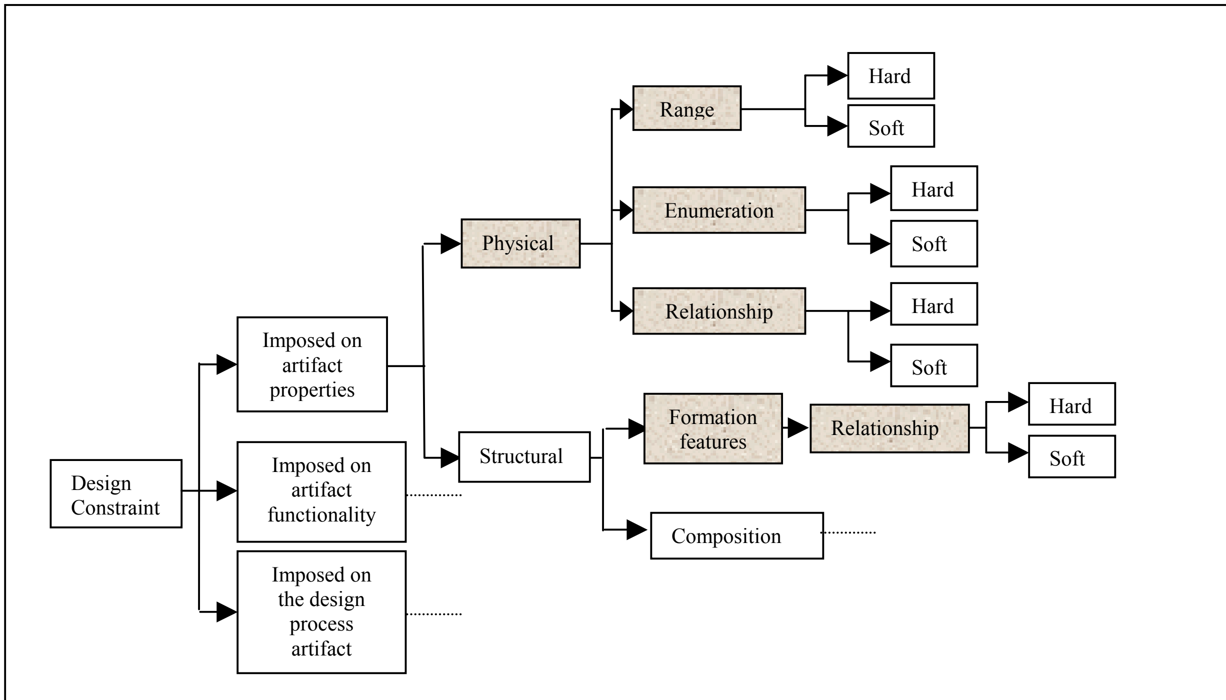

Figure 2: Constraint Categories

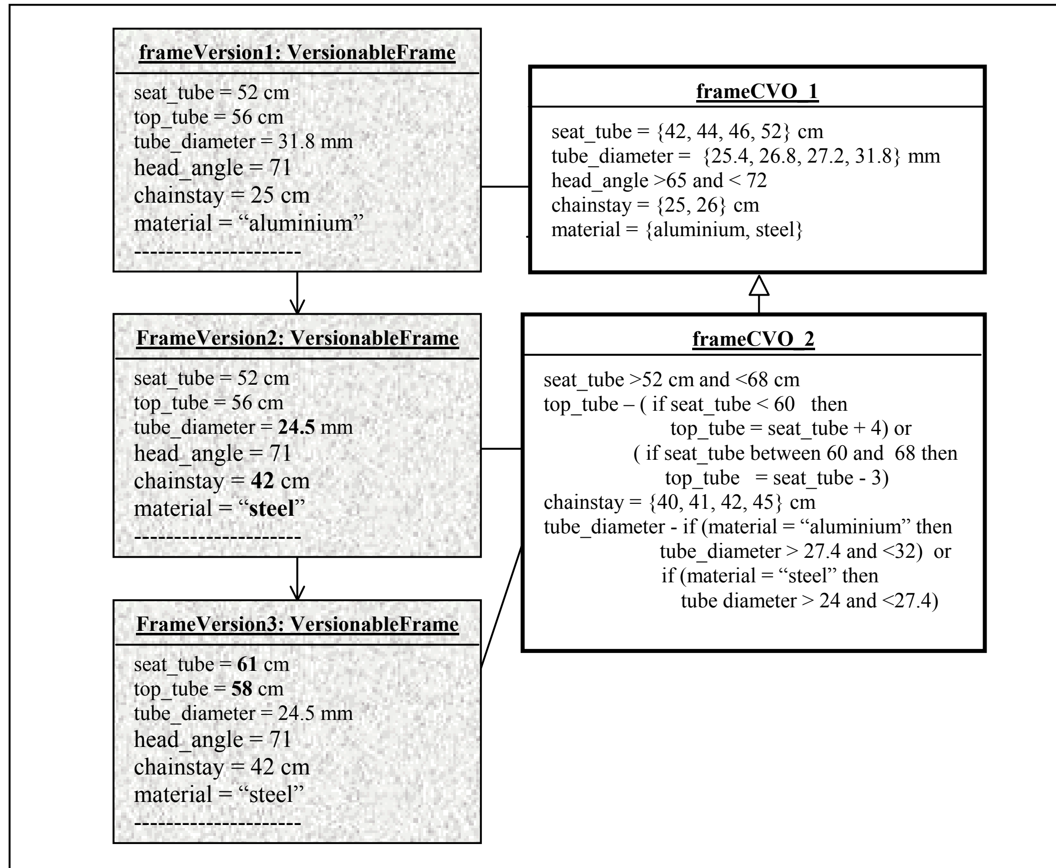

Figure 4b: Illustration of the Proposed Framework using Frame Versions and the Frame CVOs

Issues	Static Approach	Object-Oriented Approach	Active Rules
Constraints impose on	Globally on class level	Globally on class level	Globally on class level
Flexibility in manipulating evolutionary constraints	Not flexible	Not flexible	Quite flexible as they are maintained separately from the schema.
Designer's ability to define/modify constraints	Difficult since it requires programming knowledge and compilation of the program.	Difficult since it requires programming knowledge and compilation of the program.	Difficult since it requires programming knowledge and compilation of the program.
Possibility for re-use of unchanged constraints while preventing their re-definition	Yes, if the existing schema is modified with new constraints. No, if a new schema is defined with new constraints.	Yes, if the existing class is modified with new constraints. No, if a new class is defined with new constraints.	Yes, as constraint modifications can be performed independently from the schema.
Triggering validation on explicit user request	No, as validation is normally invoked immediately and automatically by the DBMS.	Depends on how the application program is designed.	No, as constraints get triggered based on the event.
Enforcing the currently active set of constraints	Automatically enforced by the DBMS.	Can be enforced automatically through the application program.	Automatic, as the constraints get triggered based on the event.
Messages to report violations enabling the designers to correct them	Validation is associated with a transaction and the transaction is aborted in a violation cancelling all the work done so far.	Can be provided through the application program.	The rule may have an action part to repair violations without designer's involvement.
Validation of existing objects in constraint modifications	Automatically takes place but inappropriate for CAD since it prevents modifications if previous objects in the DB do not adhere with the new constraints.	Not available	Not available
Managing constraint evolution histories	Histories can only be kept through producing new class definitions.	Histories can only be kept through producing new class definitions.	Histories can be kept through producing new class definitions.
Ability to retrieve constraints applicable to a particular object	Difficult as they are embedded in the schema.	Difficult as the constraints are scattered in the class definition.	Difficult as constraints are scattered in different operations.
Moving back to a previous constraint set	Possible through using the class/schema version with the required constraints.	Possible through using the class version with the required constraints.	Possible through using the class version with the required constraints.
Association between constraint evolution and object evolution	Not applicable.	Not applicable.	Not applicable.

Figure 3: Comparison between the widely used constraint management approaches