Constraint Meta-Object: A New Object Model for Distributed Collaborative Designing

D. Janaki Ram, N. Vivekananda, C. Srinivas Rao, and N. Krishna Mohan

***Abstract*— Object databases have been recognized as providing rich data modeling capabilities for the next-generation applications, especially CAD/CAM, software engineering, and multimedia. The core object model consisting of objects, messages and inheritance, however, is not adequate to tackle all the requirements of these applications. Since collaborative activity is one of the key features of next-generation applications, it is essential that the basic object model be extended to support collaborative activity. We propose an object model for collaborative designing based on objects and constraint meta-objects. Each design object has an associated constraint space consisting of one or more constraint meta-objects. The object/constraint meta-object separation achieves a clear distinction between design object and its associated design environment and provides a framework for supporting collaboration among members of a design team. Collaboration is modeled as dependencies among multiple constraint spaces on different nodes, and collaborative activity as design transactions satisfying local constraints and propagating constraints among interrelated constraint spaces. We propose language extensions to C++ for capturing the object/constraint meta-object paradigm. The extensions are implemented using filters, a special mechanism for capturing messages sent to an object. Our model has been applied to the case of a mechanical CAD problem.**

***Index Terms*—Collaborative designing, computer-aided design (CAD), computer-supported collaborative work (CSCW), constraint meta-object, distributed object databases.**

I. Introduction

NEXT-GENERATION database applications, especially CAD/CAM, multimedia, and software engineering require rich data modeling not offered by conventional relational databases [14]. The core object model consisting of classes, objects, messages, and inheritance addresses some of the needs of these applications, especially complex objects, navigational access, etc. Extensions to the core model have been proposed for version management and schema evolution [14]. However, the proposed extensions to the model do not address the requirements of collaborative activity. Since support for collaborative activity is essential in applications such as design, software engineering, and multimedia, it is necessary to augment the core data model to support collaboration. In this paper, we address the requirement of enriching the object model to support collaborative designing on a network of workstations.

Manuscript received January 15, 1995; revised November 7, 1995 and March 20, 1996.

The authors are with the Distributed and Object Systems Group, Department of Computer Science and Engineering, Indian Institute of Technology, Madras 600036, India (e-mail: djram@iitm.ernet.in).

Publisher Item Identifier S 1083-4427(97)00128-8.

Collaborative work has been extensively studied in building computer supported collaborative work (CSCW) systems [26], [25], especially collaborative writing [4], collaborative document preparation [22], and computer mediated communication [13]. There are relatively few studies in the area of cooperative designing [19] primarily focusing on cooperative newspaper design, cooperative project planning, etc. In this paper, we attempt a generalization of collaborative designing in terms of *constraint satisfaction and constraint propagation*. This is true especially of a number of design problems cooperatively solved by a team of designers. We extend the core object model to support this form of collaborative activity. The application of the model to the mechanical CAD problem has been presented.

The rest of the paper is organized in the following fashion. Section II discusses the requirements of collaborative designing in detail. Section III presents an object model to support collaborative designing. Extensions to object oriented language to support the model are also discussed in this section. We have explained the new constructs by taking a mechanical CAD problem. Section IV discusses the implementation of the model for the case of an automotive design on a network of workstations. The future directions of research and conclusions are given in Section V.

II. Requirements of Collaborative Designing

Collaboration is fundamental to group work. A detailed study of the nature of collaboration occurring in different environments can be found in [26]. Conflicts occurring among team members working on a design problem, software project, or document can be resolved through collaboration. It is essential to understand the nature of collaboration occurring in a particular domain before it can be modeled into a system. The nature of collaboration occurring in designing can be summarized in the following requirements:

A. Resolving Conflicts of Viewpoint and Conflicts of Interest

Many design problems are ill-defined in the beginning. Different team members of a design group can have a conflicting view of the problem. The design problem is defined in increasing detail in progressive steps by identifying the constraints on the solution. Conflicts of viewpoint can be seen as conflicting constraints on the design problem being solved in a collaborative fashion. Conflicts of interest arise when the optimizing functions of group members or different teams working on the problem are different. In other words,

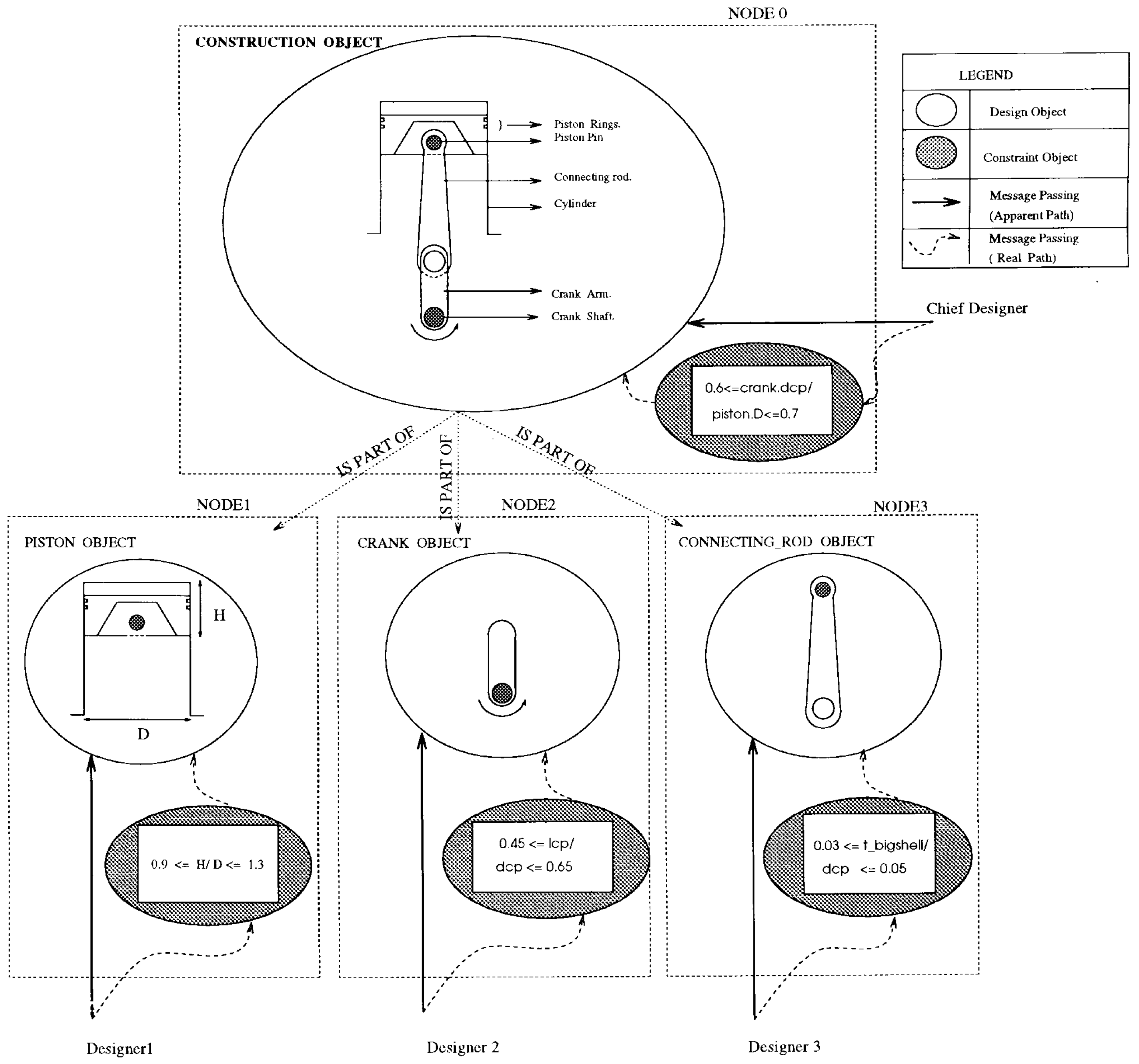


Fig. 1. Constraint meta-object model.

one group or member tries to optimize his solution at the cost of the other group or members. Conflicts of viewpoint occur in the initial stage of the designing often called rough design phase. Conflicts of interest arise between groups with opposing optimizing functions such as between design groups and manufacturing groups.

B. Asynchronous Collaboration

Collaboration among members of a design team occurs in an asynchronous fashion. Each designer works in his own design space consisting of a set of design objects at his convenient pace. A single design transaction may span from several hours to days and hence any collaboration in design environment should support long duration transactions. The need for supporting long duration transactions in design environments was recognized as early as 1985 by Kim *et al.* [10]. However, the mechanisms of check-in and check-out of the design objects from shared space to local space of the designer appears to be an oversimplification of the collaboration needed in practical design environments. A more practical approach is to capture dependencies among the design spaces of different designers and propagate necessary changes from one design space to the other as the design evolves.

C. Design Annotations

Design annotations often convey the designer's mind behind his decisions. Design annotations are critical to the understanding of the design. Substantive meta-level communication occurs among designers through design annotations. Support for recording design annotations into the database may be essential in the collaborative environment. Attribute annotations which record the designer's annotations when attribute values

```
class crankClass {

    class constraint {
        // Constraint class specification can be given here
        .........
    };
public :
    // attribute and method definitions.
    double dcp;
    double lcp;
    ....
    void set_data(int dcp);
    double getLcp();
    ...
};
```

Fig. 2. Constraint meta-class definition.

are changed can be introduced. It is possible for other designers to examine these annotations.

D. Ability to Undo Individual Actions

The need for undoing selectively individual actions when several members work simultaneously on a shared document is recognized in [1]. Such a facility for selectively undoing earlier changes without affecting other designers' changes is an essential ingredient of the design environment when a designer employs hit and trial techniques.

E. Different Roles of Members/Teams/Groups

Members play different roles in design teams. Often the role of a design critic is to critically point out the weak points in a design. The role of a product manager can be seen as one of an arbitrator between groups or members. Any collaborative system should provide role modeling including authorizations for accessing relevant design objects.

F. Support for Weak and Strong Consistencies on Design Objects

It is often important to note that a designer allows some inconsistencies in his design into the process of evolution. Often, as the design evolves in the rough design stage, constraints are dynamically identified. Systems providing strict consistency checks may, in fact, prohibit inconsistent data. This is in contradiction to the basic nature of the design process which probably requires more sophisticated and tunable consistency mechanisms on design objects [20]. As stated earlier, the design process is marked by phase transition from rough design to detailed design. As design evolves in the rough design phase, constraints are dynamically identified. In collaborative design, the designer dynamically identifies his constraint space and also checks for the validity of his constraint space with that of his collaborating team members.

```
class crankClass {

    ...
public :
    ...
};
class crankConstraintClass : constraint crankClass {
    ..
}
main() {
    crankClass crank;
    crankConstraintClass crankConstraint;
    plug crank crankConstraint;
    .
    .
    unplug crank;
}
```

Fig. 3. Dynamic plugging and unplugging of constraint meta-object.

The identification of constraint spaces of a design often marks the end of what is called *rough designing*. In the *detailed design phase*, each designer normally expands his design details to satisfy all the identified constraints. As the constraints themselves have not been identified in the rough design phase, it is not worthwhile to enforce the rules of strong consistency on the design objects. It may be possible to allow design objects to be in inconsistent state while the constraint spaces are evolving. However, once the constraint spaces are frozen in the detailed design state, it should be possible to maintain objects in consistent states. Hence, it may be required as part of collaboration to represent different levels of consistencies for design objects. A similar idea called *lazy consistency* was introduced in [16] in the context of cooperative software development on a network of workstations. It is argued that such an architecture can support a wide range of cooperative processes.

In the next section, we present an extension to the object model to effectively capture collaboration in designing.

III. Object Model for Collaborative Designing

The core object model provides a number of concepts such as object, unique object identifier, class, class hierarchy, message passing, versioning, and schema evolution. The core model allows complex objects and navigational access which are very useful in design databases. Unfortunately, the core model does not provide any support for collaborative work. Often designers work on their own workstations which are connected by a network. The basic nature of work of the designer involves collaborating with other members of the design team. In such design environment, it is often necessary for the model to support collaborative activity among designers working on a network of workstations. In this section, we pro-

```
class < name_1 > : constraint < name_2> {
      int state ;
      <label> :<constraint name > : < semantic transaction>
                [generates | depends ] < constraint label > :
                < procedure for constraint modification > ;
      < label > : < constraint name > :
                 notify_from | notify_to < list of constraint objects > ;
}


class crankConstraintClass : constraint crankClass {
      int state;
      c1 : ((0.45 ≤ lmj/dmj) && (lmj/dmj ≤ 0.6)): NULL;
      c2 : ((0.45 ≤ lcp/dcp) && (lcp/dcp ≤ 0.65)): proc_lcp_dcp(lcp,dcp) ;
      ...
}
```

Fig. 4. Constraint meta-class specification.

```
class pistonConstraintClass : constraint pistonClass {
      ...
      c2 :(( 0.64 ≤ din/dex) && ( din/dex ≤ 0.72 )): NULL :
      generates connectingrodConstraint.(smlshell_pressure ≤ 20 E-6) :
      proc_din_dex (din, dex);
      ...
}
class connectingrodConstraintClass : constraint connectingrodClass {
      c3 : (smlshell_pressure < 20 E -6 ) : NULL :
      depends_on pistonConstraint. ((0.64 ≤ din/dex ) && ( din/dex ≤ 0.76 )) :
      proc_smlpress(pressure);
      ....
}
```

Fig. 5. Generate and depend constraint.

vide extensions to basic object model to support collaboration. The requirements of such a model are:

1) It should provide enough flexibility for individual designers to work on his design space and, at the same time, provide a rich model for cooperation with other designers.
2) The collaborative model should not unduly interfere with the design process such that it slows down the design process itself.

We explain the collaboration model below.

A. Basic Collaboration Model

The collaboration model is based on design spaces and associated constraint spaces distributed across various nodes of the network. Each designer works on his local design space occasionally interacting with design spaces located on other nodes. Each design space consists of a set of design objects and belongs to a single designer represented by a node of the network. Each design object is associated with a constraint space consisting of a set of constraint meta-objects. The constraint meta-object consists of a set of constraints which govern the internal state of the object to which it is connected. They also capture the interdependencies among the design spaces. Thus, any design object's internal state is controlled by the constraints in its constraint meta-objects. We term this *constraint satisfaction on design objects*. When an interdependency among design spaces exists, the design transaction propagates the constraint space from one design space to another to satisfy the interdependency constraint. This we call as *constraint propagation to dependent objects*. In this way, the collaboration is modeled based on *constraint satisfaction and constraint propagation*.

```
class pistonConstraintClass : constraint pistonClass {
        .......
        c3 : D : notify_to crank;
}
class crankConstraintClass : constraint crankClass {
        .......
        c3 : D : notify_from piston;
}
```

Fig. 6. Notify_from and notify_to constraints.

```
class crankConstraintClass : constraint crankClass {
        c1 : ...
}
main() {
        crankClass crank;
        crankConstraintClass crankConstraint;
        plug crank crankConstraint;
        crankConstraint.disable_constraint(c1);
        ...
        crankConstraint.enable_constraint(c1);
}
```

Fig. 7. Modifying constraint space.

The constraint meta-object can be created at the time of the object creation or independently. It is possible to add or delete constraints from the meta-object by sending meta-messages to the constraint meta-object. Design transaction can invoke meta-messages to alter constraint meta-object. A graphical representation of the model is given in Fig. 1. We have taken the example from the collaborative automotive car design. Fig. 12 provides the class hierarchy diagram of a car design. The construction object consists of connecting rod, crankshaft, and piston object (see Fig. 13). Each designer works on one design space consisting of design objects. Each object is associated with a constraint meta-object in the constraint space. The designer invokes methods on the object as shown in Fig. 1. But these messages travel through the constraint meta-object associated with that object and this ensures the validation of the constraints defined at that instance of time on the design object.

Our collaboration model is based on a clear separation between design objects and design environment. The design objects represent the objects being worked out by a designer and also called design space. The design environment is represented by the constraint meta-objects which define the constraints that need to be satisfied on the design objects. The separation of design objects from constraint meta-objects has the following advantages:

1) A clear separation of design objects and constraint meta-objects helps in capturing the constraints governing the design process from the design objects themselves.
2) A collaborative model based on constraint satisfaction on objects and constraint propagation to dependent objects can be achieved easily by separating design objects and constraint meta-objects.
3) When design environments change (between companies), it is easy to create new constraint meta-objects representing the changed design environment.
4) Objects can be migrated from one constraint environment to another constraint environment in the process of evolution. This could be the case when objects migrate from designer's environment to manufacturer's environment.
5) As stated earlier, design requires varying degrees of consistency to be captured as part of the model. Since the constraint meta-objects govern the consistency of the design objects, it can be easily obtained by having different consistency models as part of the constraint meta-objects.

Since the model now consists of objects and constraint meta-objects, the following issues need to be considered:

1) How objects and constraint meta-objects are created and how they are related?
2) How are the relationships among constraint meta-objects captured?
3) How are design transactions modeled?

These issues are elaborated in the coming subsections. We have extended the C++ [18] object oriented language to support our model. For explaining the syntax and semantics of our new constructs, we have taken the application of CAD in automotive design (see Fig. 12 for the class diagram of a car design).

B. Object/Meta-Object Creation

Constraint meta-objects can be specified in two ways. They can be defined within the class definition itself. The meta-class is specified in this case as a constraint class within the class. When an object is created, the constraint meta-object is also created simultaneously and the object will be bound to its constraint meta-object at the time of the creation itself. Such a specification can be given as indicated in Fig. 2.

In this specification the coupling between the object and its constraint meta-object can be achieved at compile time. It is also possible to establish the relationship at run time and this is explained below.

The constraint meta-class can be specified separately as:

```
class ⟨name_1⟩: constraint ⟨name_2⟩ {
        // constraint specifications
};
```

In such a definition, name_1 refers to constraint meta-class name and name_2 refers to the class to which this constraint meta-class is being attached. This type of specifying the constraint meta-class is similar to the filter relationship specification discussed in [24]. Specification of a constraint meta-class does not automatically couple it to the design object. The operators *plug* and *unplug* are provided on the constraint meta-object for dynamic binding. The operator *plug* is used to couple a constraint meta-object to an appropriate object. The operator *unplug* is used to break the relationship

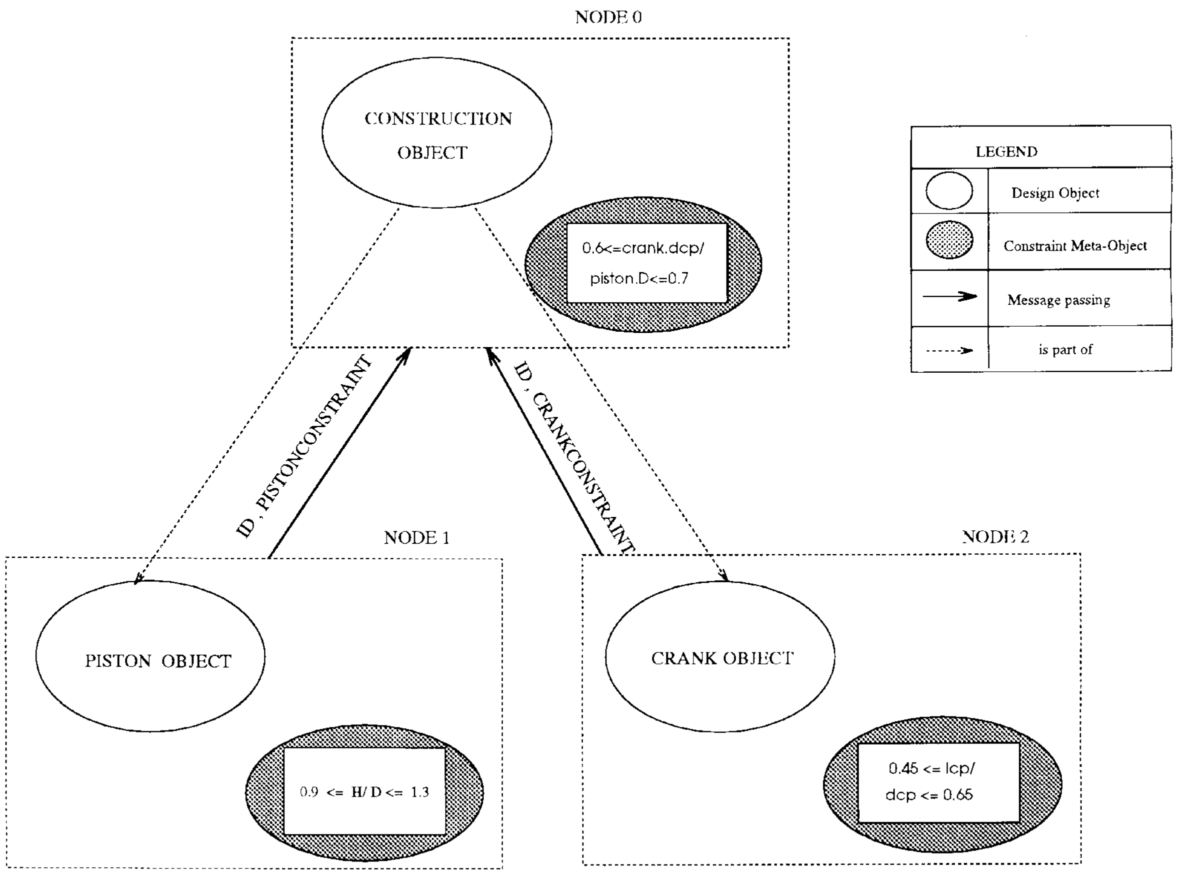


Fig. 8. Global validation scheme.

between design object and constraint meta-object. For example, in Fig. 3, a constraint meta-class *crankConstraintClass* has been defined for the design class *crankClass* outside the definition of the design class. The code binds a constraint meta-object *crankConstraint* to the design object *crank.*

This way of specifying the constraint meta-class outside the class definition provides more flexibility in the specification of constraints for an object. It allows an object to have more than one type of constraint meta-objects.The dynamic plugging and unplugging of constraint meta-object is suitable for modeling different design environments (between companies) for the same design objects.

We provide both these mechanisms for specifying constraint meta-objects.

C. Constraint Meta-Object Specification

Constraint meta-objects govern the internal state of the object. They capture the different types of constraints occurring in the collaborative designing and also the inter-dependencies among design spaces. The set of constraint meta-objects forms the design environment. The design process involves several phases such as rough design phase and detailed design phase. Since the consistency criterion in different phases are different, the meta-objects need to capture the state of design phase. The following paragraphs discuss the way of specifying constraints.

Constraint meta-class definition is similar to the template class definition of Self [8] which consists of two slots. Here, one slot is the *constraint specification* and the other slot is the *semantic transaction* which will be fired when that constraint is violated. These can be seen in traditional programming as exceptions (e.g., ADA) which are invoked when a named condition arises in the program. The constraint specification includes the label to the constraint and specification of the constraint. Each constraint class has a variable *state* which stores the current state of the design phase. In our model, the design phase can be either *rough design phase* or *detailed design phase.*

The syntax of the constraint meta-class specification is shown in Fig. 4. This also shows the constraint meta-class specification of the class *crankConstraintClass.* As shown in the figure, each constraint is associated with a semantic transaction or NULL. For example, when the constraint $(0.45 \leq lcp/dcp \leq 0.65)$ is violated, the procedure proc_lcp_dcp() will be fired to adjust the crank pin length. In case, a NULL procedure is specified, a notification to the designer will be generated.

The constraint meta-classes are also inherited in the same way as normal classes. If objects belonging to the derived class are created, the object constraint space includes the base class as well as the derived class. In the case of part_of_relationship, the constraints in the root class can be specified in terms of the variables of the container classes.

Constraints can be of different types. The coming sections explain in detail different kinds of constraints and the syntax for specifying such constraints.

1) Private and Public Constraints: Private constraints are those constraints, which belong entirely to the object's local

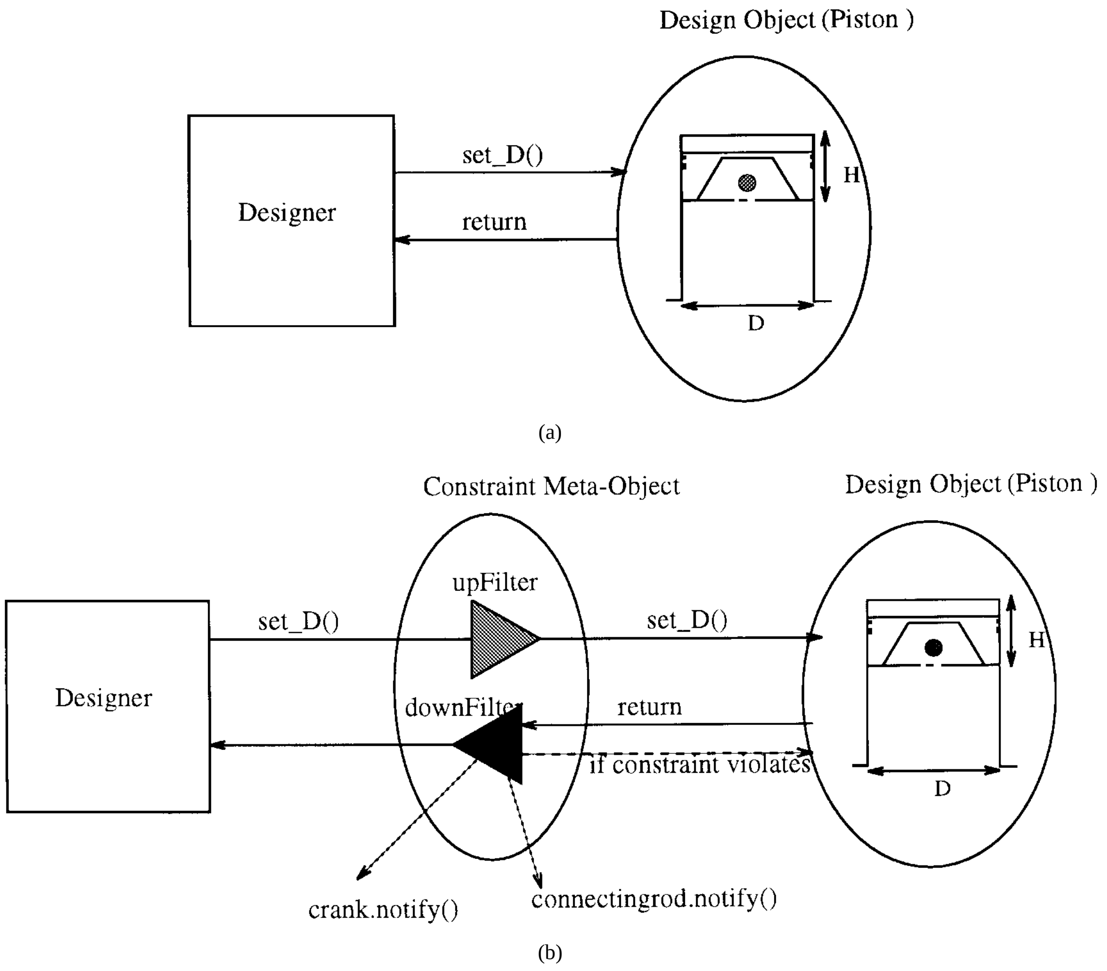


Fig. 9. Constraint meta-object implementation.

space. These constraints have validity only in that object's address space. Public constraints are those constraints whose validity spans multiple objects. For example in the case of crankClass, the constraint (*crank-pin length/crank-pin diameter* $<$ .065) is a private constraint as its validity depends only on the local state of the object. However in the case of construction class, the constraint (*crank-pin diameter/piston diameter* < 0.7) depends on both piston object and crank object. Thus, this constraint's validity extends to multiple objects.

2) Generate and Depend Constraints: Very often in engineering design, one constraint generates other constraints. For example in the case of material selection, not only does it generate stress constraints but also manufacturing constraints. In each constraint meta-class, it is possible to identify the *generate* and *depend* relationships among constraints. After the constraint specification, the generate relation captures the constraints generated by this constraint. It also identifies the procedure which can be used to subsequently generate the modified constraints, if the original constraint is altered. In case the procedure for modification is omitted, a notification to the affected objects will be generated. The reverse of *generation relation,* i.e., *depends relation,* is captured in the constraint meta-class of the generated constraints. If the generated constraints for any reason need to be modified, the classes of the dependent constraints are notified. As an example, consider the pistonConstraintClass of Fig. 5. The values of din/dex in the pistonConstraintClass generates the constraint on smlshell_pressure in connectingrodConstraintClass. Any modification of the din/dex constraint affects the constraint on smlshell_pressure and procedure proc_din_dex (din,dex) can be used in deriving the new constraint. The connectingrodConstraintClass captures the *depends* relation of the generated constraint (smlshell_pressure $<$ 20 E-6).

3) Notify_from and Notify_to: In collaborative designing, unless an earlier design decision is made, later decisions cannot be made. For example, most of the crank dimensions depend on the piston diameter. In other words, unless the piston designer has decided on the piston diameter, the crank designer will not be able to come up with many of his dimensions. Such precedence constraints in design are captured by *notify_to* and *notify_from* constraints. For example, in the case of pistonConstraintClass of Fig. 6, the piston diameter is added in the constraint meta-class as a *notify_to* constraint. The classes to be notified when the value of the piston diameter is changed is given in the *notify_to* constraint. Similarly, a *notify_from* constraint is added in the crankConstraintClass so that whenever the value changes, it is notified to the designer.

Notify constraints help in explicitly capturing the precedence order in constraints and also posting the designers

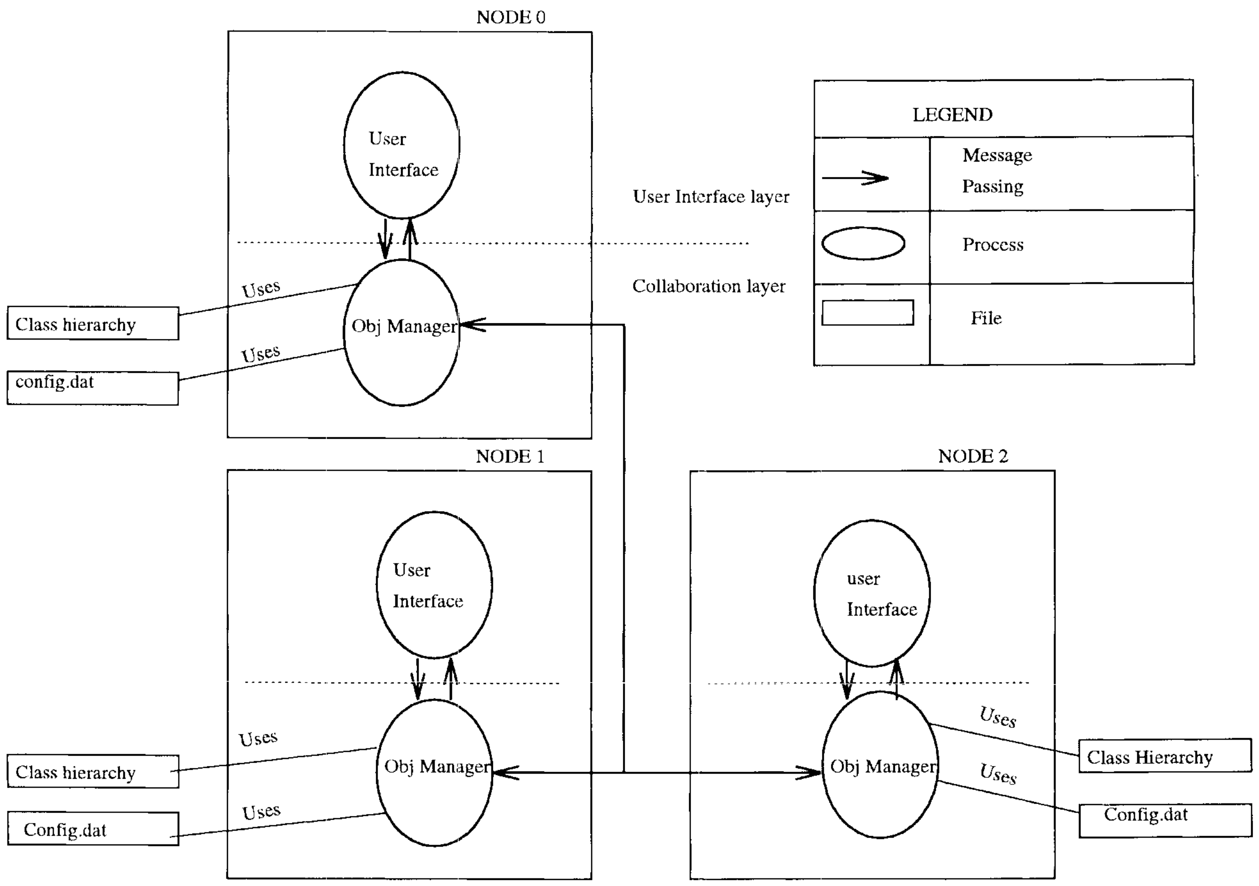


Fig. 10. Distributed collaborative design environment.

with changes in the values in other design spaces having bearing on their component design.

D. Transaction Model

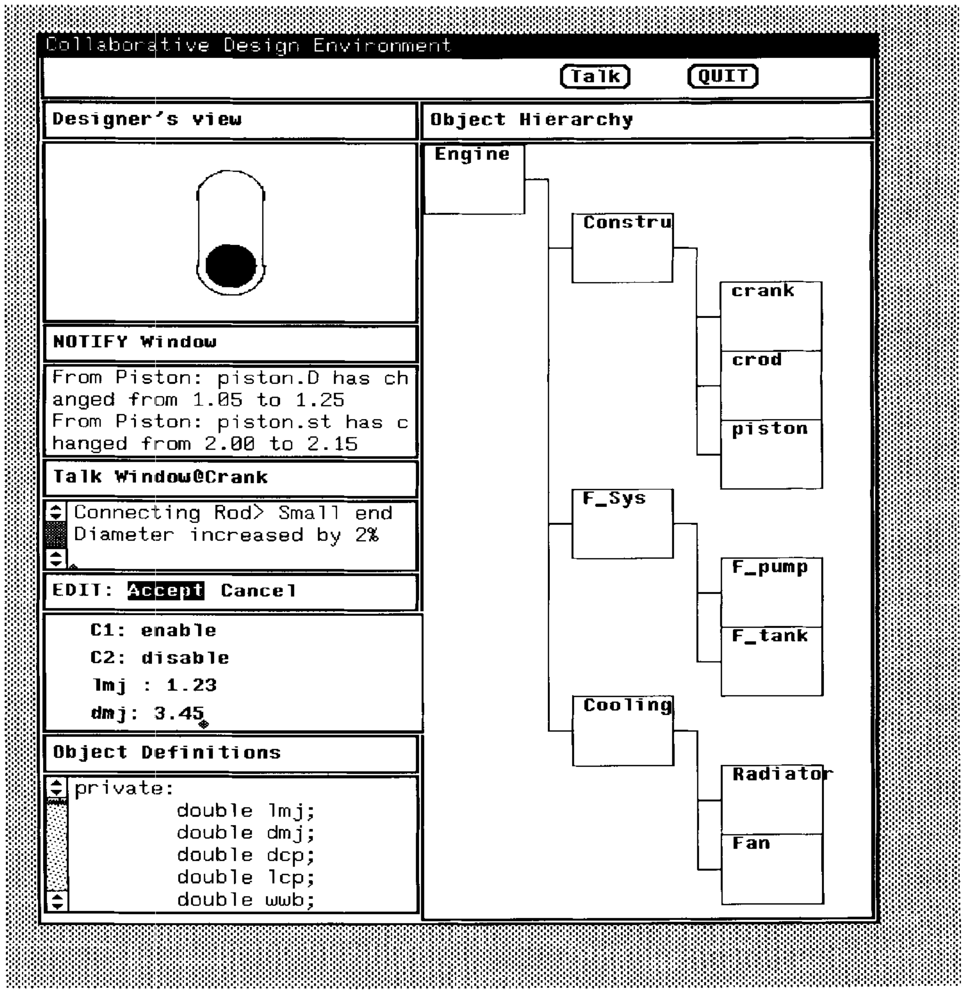
In the case of distributed object data bases, nested transactions [12] provide a natural way of modeling transactions, as messages to objects are treated as transactions, and they can further send messages to other objects. However, our transaction model differs to a greater degree in terms of the consistency imposed on the database. It has long been realized that traditional concurrency control techniques do not suit both object databases as well as advanced database applications. The notion of serializability is a very strict condition for design databases where designers often keep evolving their designs over a period of time and inconsistencies in evolving designs have to be tolerated. In this context, we propose our transaction model based on constraint spaces.

The transactions operating on a object are validated against the constraint space of the object. If a transaction does not violate the constraint space of an object, it is said to be validated. If the transaction violates a specific constraint on the object, a named method associated with the constraint called *semantic transaction* is invoked and the transaction is said to have finished successfully after the method is executed.

There are two distinct phases in the execution of the system. An evolving phase of design which we call *rough designing*. In this phase the constraint spaces are allowed to be modified by the transactions. In the later phase of design, which we call *detailed design*, the constraint spaces are not allowed to be modified. At this point, the methods associated with the constraints are not executed if constraints are violated but they are simply terminated. Thus it follows real world activity. Designers cannot make all the decisions in the beginning itself and as they make decisions they will generate new constraints which affect later design decisions. Thus, in the first phase it is identification of constraints on each design space and the second phase consists of actual detailing of the design space. The constraint space is not allowed to be modified in the second phase. The following section discusses the constructs for modifying the constraint space.

1) Modifying Constraint Space: As discussed in earlier sections, in rough design phase the designers are allowed to add, delete and modify the constraints. We have provided two special messages called meta-messages which can be sent to the constraint meta-objects for masking and unmasking the constraints. These messages take labels of the constraints as its parameter and mask or unmask corresponding constraints. As in Fig. 7, the message *disable_constraint* is sent to the constraint meta-object crankConstraint to mask the constraint with label *c1*. Similarly, the *enable_constraint* message is sent to the meta-object crankConstraint to unmask the constraint with label *c1*. However in detail design phase, the designers are not allowed to modify the constraint space.

2) Validation of Constraint Spaces: In the case of private constraints, the validation of object's state is done with respect to its local constraint space. However, in the case of public



```
From Piston: piston.D has ch
anged from 1.05 to 1.25
From Piston: piston.st has c
hanged from 2.00 to 2.15
```

```
Connecting Rod> Small end
Diameter increased by 2%
```

```
C1: enable
C2: disable
lmj : 1.23
dmj: 3.45
```

```
private:
        double lmj;
        double dmj;
        double dcp;
        double lcp;
        double wwb;
```

Fig. 11. Screen layout of collaborative design environment.

constraints, the validation needs to be done against other constraint spaces. We follow general mechanism of validation which is described in the next paragraph.

Each method piggy-backs the constraint space of the object which invokes the method. The method not associated with any object piggy-backs a null constraint space which is automatically validated against any other constraint space. The piggy-backed constraint space of a method is validated against the constraint space of the object on which the method tries to operate. If there are violations, the method invokes the semantic transaction associated with the constraint on the piggy-backed constraint space. This semantic transaction operates on the original object. If the method is allowed to operate on the object and subsequently violates the constraint space, the method associated with the constraint space is invoked. If the method generates one more method, the new method now inherits the constraint space of the current object as well. Thus, it piggy-backs both its earlier constraint space as well as the current constraint space. We achieve using this property, both constraint satisfaction as well as constraint propagation.

At the end of every design phase, a validation phase is started. In the case of rough design phase, the validation phase is a dummy nested transaction which carries with itself the constraint spaces from one design space to the other design space and does a global validation of the constraint spaces. As shown in Fig. 8, when the designer of *construction object* initiates the validation, the constraint spaces of its child objects *piston* and *crank* move to the node of their parent object *construction*. The node of the *construction* object does the validation of the inter-dependent constraints (e.g., the constraint $0.6 \leq$ crank.dcp/piston.D ≤ 0.7) against the local constraints of *piston* (e.g. $0.9 \leq$ H/D ≤ 1.3) as well as *crank* (e.g. $0.45 \leq$ lcp/dcp ≤ 0.65). Once the global validation is over, the constraint spaces of each designer are frozen. Hence, the designer cannot change his constraint space. This in effect commits each designer firmly to the choices he made in the rough design phase.

IV. Implementation of the Model

The distributed object model for collaborative designing has been implemented on a network of SUN workstations. We have taken a mechanical CAD problem in automotive car design to demonstrate the working of our model. The detailed car design represented in terms of class diagram is shown in Fig. 12. As can be seen, it is not possible for a single designer to design the whole car within a reasonable time.

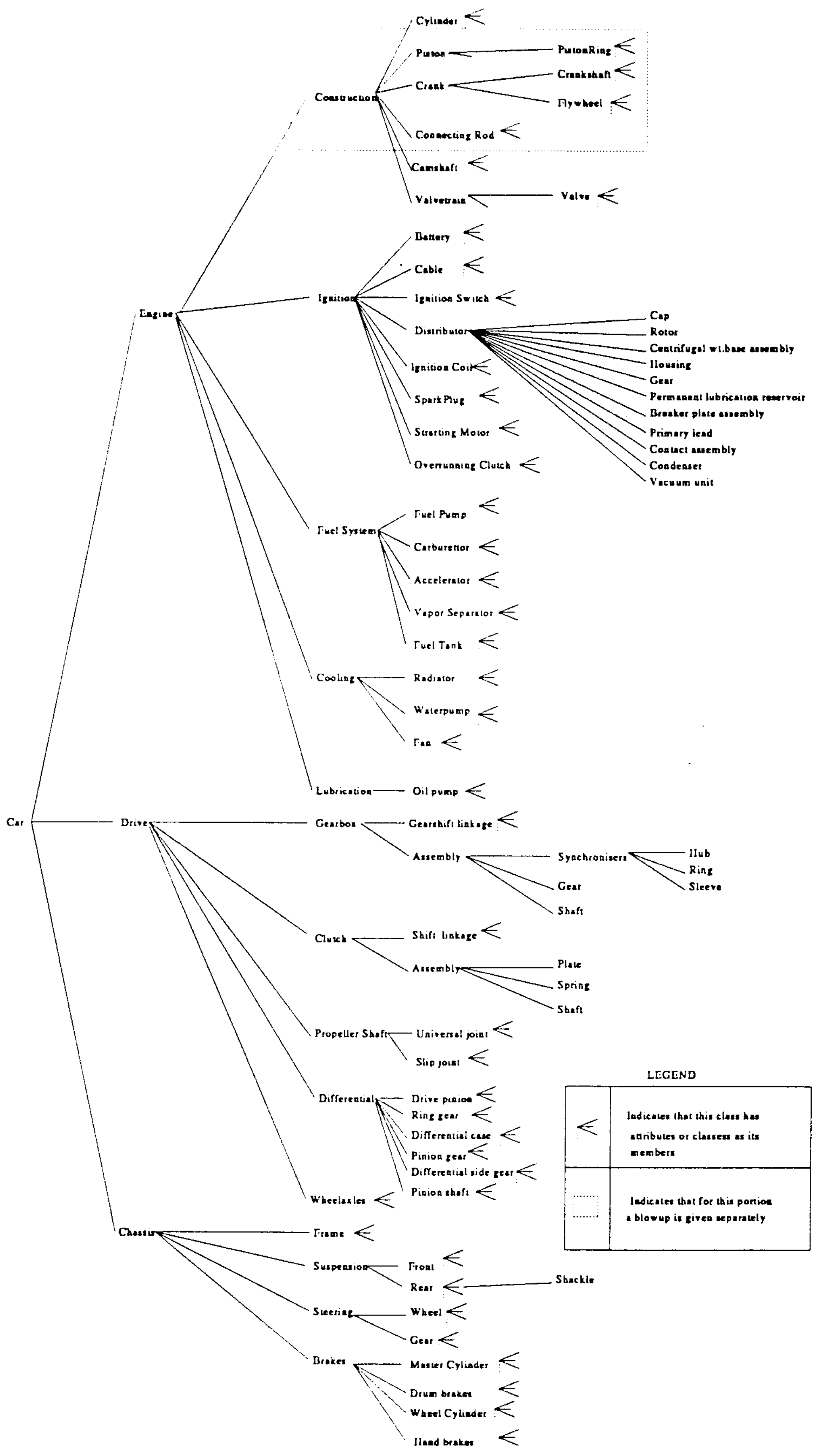


Fig. 12. Class diagram of a car design.

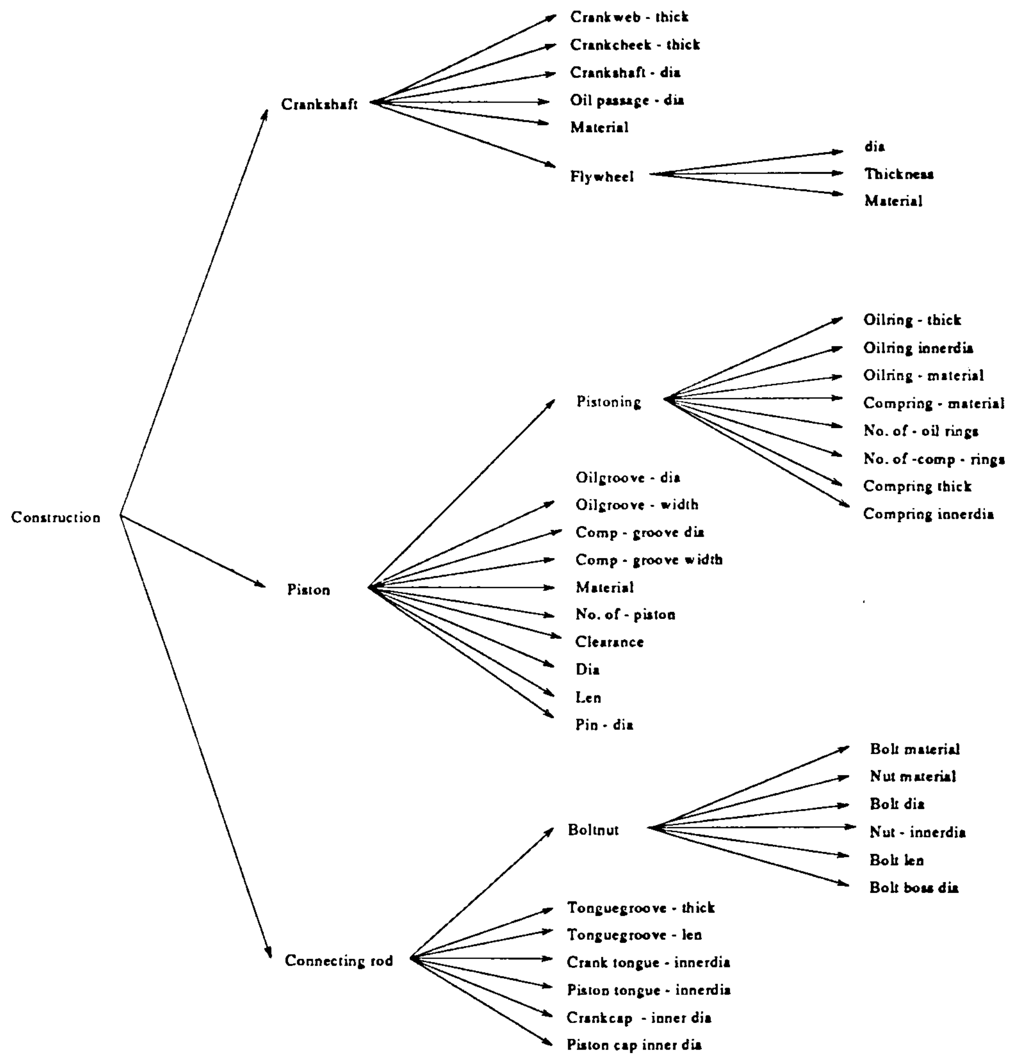


Fig. 13. Class diagram of construction object.

We take a specific portion of the class diagram to explain our model. This portion is shown in Fig. 13. It consists of a root class named constructionClass and it contains pistonClass, crankClass, and connectingrodClass. The pistonClass, the crankClass, and the connectingrodClass are distributed on three nodes of the network. These three components are being designed by three designers cooperatively. We have taken the specific information relating to piston, crank, and connecting rod design from [30] and [23].

This section deals with the implementation details of our model. First subsection deals with the translator which we have developed to translate the C++ code with constraint meta-object constructs to the plain C++ code. The second subsection discusses the implementation of the distributed design environment.

A. User Level Implementation of New Constructs

A translator has been built to convert the C++ code with the proposed language extensions to plain C++ code. It has been developed using the concept of *filter-objects*. The complete details of *filter-objects* can be found in [24]. The *filter object* model provides a mechanism for capturing messages sent to an object without the sender being aware of the filtering action. Since constraint meta-objects capture the messages sent to the design object for validation purposes, we built the constraint meta-object model based on the *filter-object* model. Fig. 9 depicts the implementation of the constraint meta-object using *filter-object* model. The traditional object model in which messages are directly delivered is shown in Fig. 9(a). As opposed to this model, the *filter-object* model provides a transparent interception of the messages sent to an object by the filter object, in this case the constraint meta-object. The constraint meta-object does some housekeeping operations before the message is passed to the actual object. It also captures the return messages from the object before it is delivered to the sender.

The design application programs are written using C++ language with the extensions provided in Section III. This code is translated by a translator into plain C++ code.

The main working steps of the translator are given below:

1) For each design object's class, the translator identifies specification of all the constraint meta-classes. These will be made as the derived classes of a common class,

namely constraintClass. Since the design object can have at most one constraint meta-object attached to it, the instance of the constraint meta-object is made as an instance of the constraintClass when a meta-object is plugged to an object. The constraintClass is made *a part of* of the design class.

2) Since the constraint meta-classes need to access the attributes of the design object, the constraint meta-classes are made as *friend* classes of the design object.
3) In the translated code, each constraint meta-class will have two methods namely *upFilter* and *downFilter* [see Fig. 9(b)]. Every method invocation on the design object is made to pass through the *upFilter* function. This keeps a copy of the object in temporary location. This is required to recover the object from the new state if the constraints are not satisfied. The return message from the design object is made to go through the *downFilter* method. All constraint specifications are translated as statements in the *downFilter* method.
4) The *downFilter* achieves the constraint satisfaction on the objects. In the case of objects which involve child objects, the *downFilter* gets the constraint spaces from its child objects for global validation of the constraints (see Fig. 8).
5) The implementation of meta-messages *enable_constraint* and *disable_constraint* are achieved by setting and clearing the mask bit associated with each constraint.

B. Implementation of Distributed Environment

Since each designer is working on his own workstation, the design objects are distributed across various nodes of the network. The distributed design environment allows the designer on one workstation to access objects located on other workstations. Transactions such as validate and notify also use the distributed design environment to achieve *constraint propagation and satisfaction* among design spaces distributed over the network. The distributed object environment as shown in Fig. 10 consists of *object manager module* and *user interface module* located on each node of the network participating in the collaborative design.

1) Object Manager: Object manager runs in background on every node of the system. It provides persistence of the design objects and also ensures the consistency of objects by *constraint satisfaction and constraint propagation* mechanism. Each node maintains a configuration file, namely *config.dat*, that stores the distribution of design objects in the class diagram and also the access permissions for various objects. An example config.dat for the class diagram of Fig. 13 has been given in Table I. Each class is associated with a password which can be used to explicitly access objects of the class. The password is required while modifying the objects individually. It is also possible to set *read password* to prevent objects from unauthorized reading.

TABLE I
CONFIGURATION FILE (config.dat) FOR CLASS DIAGRAM

class	node	user	password
construction	node1	construction	passcon
crank	node1	crank	passcrank
connectingrod	node2	connectingrod	passrod
piston	node2	piston	passpiston

The object manager can receive transactions from object managers residing on other nodes or from the user interface module. User interface module generates the *design transactions* on the objects. Before these transactions can make the modification on the object, they are validated against the constraints specified in the constraint meta-object which is plugged to that object. In certain situations, object manager generates transactions such as *notify* or *generate* which is sent to other object managers. The communication among object managers is achieved by RPC mechanism [29].

2) User Interface: Each designer is provided with an interface through which he can manipulate design objects. This has been implemented using *suncore* language (see Fig. 11 for the screen layout of the collaborative design environment). A *Object Hierarchy window* displays the hierarchical structure of the class diagram. The designer can choose an object and work on that object. Along with this window it provides other windows such as *Designer's view window*, *NOTIFY window*, *talk window*, and *EDIT window*. The *EDIT window* facilitates the designer to modify the values of the objects and the constraints (only in rough design phase). These actions are translated as design transactions and they are passed to the object manager. The *Designer's view* displays corresponding object. *NOTIFY window* notifies the designer when a *notify* message from other design objects arrives. *Talk window* facilitates the designers to engage in discussions. This assumes significance in the context of collaborative activity and helps in resolving conflicts of view and conflicts of interest. Fig. 11 shows the screendump of the crank designer. The designer has invoked crank object from the class hierarchy of the automotive car design. The *Object Definitions window* is showing object definitions of the crank object. The *NOTIFY window* shows the two notify messages from the piston object regarding the changes the piston designer has made to the object. *Talk window* shows the discussion the crank designer had with the designer of connectingrod.

V. CONCLUSION AND FUTURE WORK

A general model for collaborative designing based on *constraint meta-object model* has been proposed. The constraint meta-object model achieves *constraint satisfaction and propagation* among design spaces distributed across various nodes of the network. The model has been applied to the case of a mechanical CAD problem. A prototype system has been developed and its working has been presented.

We are currently working on extensions to the model in which same design objects can exist in different contexts such as in design environment and manufacturing environment. In such cases the object needs to display different behavior in different contexts. It is possible to attach dynamically mutable abstractions to persistent objects to allow these to exist in different contexts. We call such objects *interface polymorphic (IP) objects* [5]. IP objects coupled with constraint meta-object model can provide collaboration between design teams and manufacturing teams.

ACKNOWLEDGMENT

The authors would like to thank the anonymous referees whose comments have greatly helped them in improving the quality of this paper. They would also like to thank all the members of Distributed and Object Systems Group, especially R. K. Joshi, for providing valuable suggestions.

REFERENCES

[1] A. Prakash and M. J. Knister, "Undoing actions in collaborative work," in *Proc. ACM 1992 Conf. Computer-Supported Cooperative Work (CSCW 92)*, pp. 273–280.

[2] B. Meyer, *Eiffel: The Language*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

[3] B. Reeves and F. Shipman, "Supporting communication between designers with artifact-centered evolving information spaces," in *Proc. ACM 1992 Conf. Computer-Supported Cooperative Work (CSCW 92)*, pp. 394–401.

[4] C. M. Neuwirth, R. Chandhok, D. S. Kaufer, P. Erion, J. Morris, and D. Miller, "Flexible diff-ing in collaborative writing systems," in *Proc. ACM 1992 Conf. Computer-Supported Cooperative Work (CSCW 92)*, pp. 147–154.

[5] D. J. Ram, N. Vivekananda, and R. K. Joshi, *Interface Polymorphism: Difference Faces of an Object*, Dept. Comput. Sci. and Eng., Indian Institute of Technol., Madras, India, Tech. Rep IITM-CSE-DOS-95-12, Aug. 1995.

[6] D. L. Spooner, M. A. Milicia, and D. B. Faatz, "Modeling mechanical CAD data with data abstraction and object-oriented techniques," in *IEEE Int. Conf. Data Engineering*, 1986, pp. 416–424.

[7] D. Sriram, R. Logcher, A. Wong, and S. Ahmed, "An object-oriented framework for collaborative engineering design," in *Proc. Workshop on Computer-Aided Cooperative Product Development: Lecture Notes in Computer Science*, vol. 492. Berlin, Germany: Springer-Verlag, 1989, pp. 51–92.

[8] D. Ungar, R. B. Smith, C. Chambers, and U. Holzle, "Object, message, and performance: How they coexist in self," *IEEE Comput.*, vol. 25, no. 10, pp. 53–64, 1992

[9] E. Egger and I. Wagner, "Time-management: A case for CSCW," in *Proc. ACM 1992 Conf. Computer-Supported Cooperative Work (CSCW 92)*, pp. 249–256.

[10] F. Bancilhon, W. Kim, and H. F. Korth, "A model of CAD transactions," Dept. Comput. Sci., Univ. of Texas, Austin, Tech. Rep. TR-85-06, 1985.

[11] H. F. Korth and G. D. Speegle, "Long-duration transactions in software design projects," in *IEEE 6th Int. Conf. Data Engineering*, 1990, pp. 568–574.

[12] J. E. B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*. Cambridge, MA: MIT Press, 1985.

[13] J. Galegher, "Computer mediated communication and collaborative writing: Media influence and adaptation to communication constraints," in *Proc. ACM 1992 Conf. Computer-Supported Cooperative Work (CSCW 92)*, pp. 155–162.

[14] J. V. Joseph, S. M. Thatte, C. W. Thompson, and D. L. Wells, "Object-oriented databases: Design and implementation," *Proc. IEEE*, vol. 79, no. 1, pp. 42–64, 1991.

[15] K. Gronbaek, M. Kyng, and P. Mogensen, "CSCW challenges in large-scale technical projects—a case study," in *Proc. ACM 1992 Conf. Computer-Supported Cooperative Work (CSCW 92)*, pp. 338–345.

[16] K. Narayanaswamy and N. Goldman, "Lazy consistency: A basis for cooperative software development," in *Proc. ACM 1992 Conf. Computer-Supported Cooperative Work (CSCW 92)*, pp. 257–264.

[17] L. Shu, "Groupware experiences in three-dimensional computer aided design," in *Proc. ACM 1992 Conf. Computer-Supported Cooperative Work (CSCW 92)*, pp. 179–186.

[18] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*. Reading, MA: Addison-Wesley, 1990.

[19] M. Kyng, "Designing for cooperation: Cooperating for design," *Commun. ACM*, vol. 34, no. 12, pp. 65–73, 1991.

[20] G. T Nguyen and D. Rieu, *Database Issues in Object Oriented Design: Technology of Object Oriented Languages and Systems TOOLS 4*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

[21] N. S. Barghouti and G. E. Kaiser, "Concurrency control in advanced database applications," *ACM Comput. Surv.*, vol. 23, no. 3, pp. 269–317, 1991.

[22] P. Luff, C. Health, and D. Greatbatch, "Tasks-in-interaction: Paper and screen based documentation in collaborative activity," in *Proc. ACM 1992 Conf. Computer-Supported Cooperative Work (CSCW 92)*, pp. 163–170.

[23] P. Lukin, G. Gasparyants, and V. Rodionov, *Automobile Chassis Design and Calculations*. Moscow, Russia: Mir, 1989.

[24] R. K. Joshi, N. Vivekananda, and D. J. Ram, "Message filters for object oriented systems," Dept. Comput. Sci. and Eng., Indian Institute of Technol., Madras, India, Tech. Rep. IITM-CSE-DOS-95-08, July 1995 (to appear in *Software Practice and Experience*).

[25] S. Chakravarthy, K. Karlapalem, S. B. Navathe, and A. Tanaka, "Database supported cooperative problem solving," *Int. J. Intell. Cooperative Inform. Syst.*, vol. 2, no. 3, pp. 249–287, 1993

[26] S. M. Easterbrook, E. E. Beck, J. S. Goodlet, L. Plowman, M. Sharples, and C. C. Wood, *A Survey of Empirical Studies of Conflict, CSCW: Cooperation or Conflict?*. Berlin, Germany: Springer-Verlag, 1993, pp. 1–68.

[27] T. C. Ralow, J. Gu, and E. J. Neuhold, "Serializability in object-oriented database systems," in *IEEE 6th Int. Conf. Data Engineering*, 1990, pp. 112–120.

[28] V. Arkhangelsky, M. Khovakh, Y. Stepanov, V. Trusov, M. Vikhert, and A. Voinov, *Motor Vehicle Engines*. Moscow, Russia: Mir, 1976.

[29] W. R. Stevens, *Unix Networking Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1993.

[30] W. H. Crouse, *Automotive Mechanics*. New York: McGraw-Hill, 1981.

D. Janaki Ram was born in 1962 in Andhra Pradesh, India. He received the B.Tech degree in mechanical engineering from the College of Engineering, Kakinada, India, in 1983, and the M.Tech and Ph.D. degrees in computer science from the Indian Institute of Technology (I.I.T), Delhi, in 1985 and 1989, respectively.

He was with the National Informatics Center, New Delhi, and subsequently moved to the I.I.T., Madras, in 1989. He is currently an Assistant Professor in the Department of Computer Science and Engineering, I.I.T., Madras, and heads the Distributed and Object Systems Group. His research interests include distributed and heterogeneous computing, distributed databases, object technology, software engineering, and CAD/CAM. He has published several international papers in these areas. He regularly teaches courses in the above areas at postgraduate and undergraduate levels and has guided several B.Tech, M.Tech, and M.S. projects in the above areas. He has conducted several national level workshops on object oriented programming, object oriented design and analysis, and client-server computing, and introduced the course on object oriented software development at I.I.T., Madras. He is a consultant to several industries and offers consultancy services, mainly in the area of object orinted design and analysis.

N. Vivekananda received the B.S. degree in computer engineering from Karnataka Regional Engineering College, Suratkal, India, in 1993. He is presently pursuing the M.S. degree in computer science and engineering at the Indian Institute of Technology, Madras.

He was previously a Research and Design Software Engineer with Wipro Infotech Ltd., Bangalore, India. His areas of interest include object technology, distributed collaboration, and computer networks.

C. Srinivas Rao received the M.Sc degree in computer science from Andhra University, Visakhapatnam, India, in 1988, and the M.Tech degree in computer science from the Indian Institute of Technology, Madras, in 1996.

He has been a Scientist at the Centre for Aeronautical Systems Studies and Analyzes (CASSA), Defence Research and Development Organization, Bangalore, since 1989. His areas of interest are object oriented databases and computer networks.

N. Krishna Mohan received the Master of Technology degree in computer science and engineering from the Indian Institute of Technology, Madras,

He is a Senior I.T. Engineer with CMC Ltd., Bombay, India. His current areas of interest include object-oriented systems, distributed systems, relational database systems, computer networking, operating systems and OLTP.