# Applying an Advanced Data Model to Graphic Constraint Handling *

*Johan van den Akker*† *Arno Siebes*

CWI
Kruislaan 413
1098 SJ Amsterdam
The Netherlands
*e-mail: {vdakker,arno}@cwi.nl*

**Abstract**

Constraints are studied in a variety of computer science disciplines. Comparative studies of the different perceptions of constraints and approaches to their management can lead to valuable cross-fertilisation of the different research tracks.

This paper sets out such a comparative study. More in particular, it is shown how constraints between graphical objects can be modelled in an advanced model for object databases. Through examples, it is shown that a "divide and conquer" strategy, developed in database research is also a viable approach for constraint management in graphical systems.

## 1 Introduction

In a wide variety of applications, constraints are a natural part of the specification of a system. Consequently, constraints and constraint management are topics in a spectrum of computer science research areas, such as computer graphics and databases. Since the research communities are largely disjoint, the research tracks in constraints develop more or less in isolation.

Comparative studies of the different perceptions of constraints and the approaches to their management can lead to valuable cross-fertilisation of the different research tracks. For example, both in computer graphics and in

*In:*Proceedings of the 5th Eurographics Workshop on Programming Paradigms in Graphics*, Maastricht, The Netherlands, September 2-3, 1995, pp.1-16, ISBN 90-6196-458-X.

†This author is supported by SION, the Foundation for Computer Science Research in the Netherlands through Project no. 612-323-424

databases embedding constraint management in an object-oriented setting causes conflicts with the encapsulation of attributes. Solutions in one area might induce solutions in the other.

In this paper we set out such a comparative study, by showing how a solution for constraint management in (active) object-oriented databases can be used to model constraints between graphical objects. The key-observation of this solution is that a constraint is a relation between objects.

Maintaining integrity constraints in databases has always been an important subject of research. The initial approach was often to check for constraint violation on the commit of a transaction. If the transaction violated the constraint, it was aborted. A more sophisticated approach was made possible by active database management systems (ADBMS) [13]. An ADBMS offers a facility to define production rules. A commonly accepted format for rules in databases is an Event-Condition-Action (ECA) triple. On the event specified the action is executed if the condition is satisfied. Thus, it becomes possible to implement corrective actions for constraint violation beyond transaction abortion. An example is the invocation of a wire rearrangement algorithm if a minimum distance constraint is violated in a circuit design system.

A number of active DBMSs have been built. At first using the relational paradigm [11, 14], more recently systems have been based on the object-oriented paradigm [10, 7]. A description of the use of active rules for the enforcement of constraints can be found in [6].

How rules are integrated with the rest of the database is a subject of discussion. Rules can be integrated with the object database by making them objects as well [8]. This approach emphasises easy manipulation of rules.

Another approach starts from the premise that rules are part of the behaviour of objects. In other words, they should be encapsulated with the object. An object-oriented DBMS that offers such encapsulation is SAMOS [10]. SAMOS, however, is a hybrid model, rules can be part of the class definition, and they can be defined in separate objects. We take this approach to its extremes, *all* behaviour is encapsulated within an object. This makes the objects autonomous.

Object autonomy is the fundamental concept of the data model in this paper. Autonomy has two sides. First, as stated in the previous paragraph, all static and dynamic aspects of an object are defined within the object. Secondly, the object is subjected to its own, local, control only. That is, there is no central control.

The development of a database of autonomous objects is motivated by a number of developments in computing and in business. Developments in parallel computers and networking raise the question of viability of central control in a database system. Examples are load-balancing problems in massively parallel computers or trying to track the nodes connected in a mobile network. It may very well be that the overhead of such control affects system performance in such a way that distribution of all control to components of the system is necessary. In other words, the components are forced to be autonomous.

On the other hand there are situations were central control of an information system is not wanted. This is the case in a lot of inter-organisation information systems. Examples of these are trading systems in financial markets and chain information systems integrating suppliers and customers. In such systems the owners of the parts will not want to give up control over their part of the system. Thus, we need a system that preserves the autonomy of the components of the system.

The onset of active components in database systems is parallelled in the computer graphics community. For example, the programming language PROCOL [5] and the MADE object model for multimedia applications [2], feature active objects. Moreover, large collections of communicating, independent processes are supported by MANIFOLD [3]. This parallel development suggests that it is worthwhile to consider the database solution of constraint management in an active system for computer graphics.

A data model underlying a database system serves to offer a clear conceptual view of the information we bring into an information system. This conceptual view is offered in terms of objects and their relations. The contribution database research can make to the area of computer graphics lies in the prominent role of relations. A constraint between objects is a special kind of relationship between those objects. More in particular, in this paper we show that this localisation of constraints offers a flexible mechanism for constraint management in computer graphics.

Please note that this paper is not an attempt to integrate the two paradigms of object-oriented and constraint programming, such as for example [9]. We simply show how the techniques used in databases can be applied to the field of computer graphics.

The roadmap for this paper is as follows. In the next section we introduce the concepts of the autonomous datamodel, a full description of this model can be found in [1]. In Section 3, it is shown how (computer graphics) constraints are defined in this model. The fourth and final section of this paper contains our conclusions from this exercise.

# 2 The Autonomous Data Model

In the previous section we explained what object autonomy is and what developments promote object autonomy. In this section we describe how autonomous objects can form the foundation of a database system.

## 2.1 A Database of Autonomous Objects

In a database of autonomous objects everything that can be specified a priori of an object is defined on the object itself. In particular, the behaviour of an object is defined by three components, viz., the methods, the (dynamic) constraints and the rules.

In the methods one specifies *what* an object can do. The (dynamic) constraints specify which methods an object is willing to execute in a given context; in general, it are stipulations about the order in which methods have to be executed. The rules specify when an object will execute a given method. Rules state actual actions to be taken in certain situations described in terms of events and object states. Thus, the first two components describe *potential* behaviour, whereas the last component describes *actual* behaviour.

Compared to more traditional object-oriented databases, the rule component of the behaviour specification is new. That is, traditionally only potential behaviour is specified whereas autonomous objects also contain their actual behaviour.

Orthogonal to the active behaviour of objects is the evolution of their relations and capabilities. During its lifecycle an autonomous object develops just like anything in the real world from people to forms. An object is created, acquires and loses relations and consequently gains and loses capabilities. For example, a mouse pointer that is moved over different windows, is related to the window it is currently in. The actions that can be performed by the mouse are dependent on the window it is in.

A further example is given by a multimedia movie. Suppose there are different presentation objects or players for video, audio and text in the system. The movie has relations with those presentation objects currently associated with it. What actions can be performed on the movie is dependent on the presentation objects currently related to the movie. For example, if the movie enters a relation with the video player, it gains methods to have its video component displayed. Once it loses this relation, it loses the capability to display its video component.

These characteristics make a number of requirements on the data model. Firstly, objects are autonomous in trying to initiate and terminate relations. Secondly, the capabilities of an object are dependent on the relations it is currently involved in. Therefore we need a mechanism that, often temporarily, extends an object. In the next sections we will first explain what an individual autonomous object looks like. After that we will describe how relations between autonomous objects are modelled.

## 2.2 An individual autonomous object

To get a more concrete picture of a database of autonomous objects we will first show how an individual autonomous object is defined. An autonomous object has attributes, methods, rules, and constraints. We will often refer to these as the capabilities of an object.

**Attributes** Like any object, an autonomous object is defined by data and behaviour. The data in an object are the attributes defined for it. These can be any basic types, and tuples and sets formed of these. The encapsulation of attributes and methods is the same. An attempt to access an attribute from outside an object is treated as a method call. Thus, access to attributes is controlled through the dynamic constraints of an object. Attributes are declared in the attribute section of an object in a very straightforward way. For example, the attribute section of a line object might look as follows:

```
Attributes
    Position:    (x:integer, y:integer)
    Angle:       integer
    Length:      real
```

**Methods** Methods define the actions an object is able to perform. Methods are declared in the method section of an object. Methods can change an object's attributes and can call methods in other objects. An example of the methods of a line object is:

```
Methods
    Rotate(degrees:integer) {
        angle := angle + degrees
    }
    Translate(xtrans:integer, ytrans:integer){
        Position.x := Position.x + xtrans
        Position.y := Position.y + ytrans
    }
```

**Rules** The active behaviour of an object is event-driven. It is defined by event-condition-action (ECA) rules. This means that on a specified event the action is executed if the condition stated is satisfied. All three components are about the object the rule is defined on and the objects that can be reached from it through path expressions. A very simple example is a rule that triggers a redraw each time a line object is translated or rotated.

**Rules**
**On** Rotate
**do** Redraw;
**On** Translate
**do** Redraw;

**Dynamic Constraints** The behaviour of an object can be constraint through the definition of dynamic constraints. These are basic process algebraic expressions [4] with guards. This means that dynamic constraints can enforce sequencing and preconditions of methods. An example of a precondition is that a line in the origin of the coordinate system may not be rotated. An example of sequencing is that a redraw must take place after a rotation or a translation before any other action is taken.

**Dynamic Constraints**
([Position.x!=0 **or** Position.y!=0]Rotate)*
(Rotate;Redraw)*
(Translate;Redraw)*

## 2.3 Relations between Objects

In a database we have entities such as the bank and natural persons that exist independently. These are the ordinary objects in our database. However, these objects do not lead an isolated existence. In this section we will show how we bring the relations between objects in our model. To illustrate the way relations between autonomous objects are modelled, we first give an example from the banking world. In the next section we will look at some examples in computer graphics.

**Example** A bank account is a relation between a person and a bank. A person can open an account with a bank, if he satisfies certain conditions. These conditions vary from presenting a valid identity card to not having a bad credit record. After that the account will be opened, in other words the relation is established. The person now can do a number of things because of this relation. He can, for example, transfer money to other bank accounts. Naturally, the relation can also be terminated. Again a number of conditions need to be satisfied. The bank account will not be closed, if

there is an overdraft on it. Similarly the conditions of the bank account will describe what happens in exceptional circumstances. An example is that the account is transferred to one of the heirs if the account holder dies.

In an autonomous database we proceed in an analogous way in establishing relations. A relation is defined by a protocol. The protocol describes how a relation is initiated, how it can be terminated and what is done in exceptional cases. The protocol for a relation is stored partly in a relation class object and partly in a relation object. The information needed to initiate a relation is stored in the relation class object. The rest of the protocol, needed during the lifecycle of the relation is kept in a relation object.

On an initiation request the relation class object first checks if the objects that request the relation satisfy the conditions in the protocol. If this is successful the actions to actually create the relation are taken. In order to store information about the relation a relation object is created. A relation object is itself an autonomous object. This means that it can in turn engage in relations with other objects. At the same time the objects that engage in the relation are extended with the capabilities to deal with the relation. This is done through the addon mechanism. An addon defines an extension to an object. If an object is extended through an addon, it acquires the capabilities defined in the addon. Since an addon only defines an extension to an autonomous object, there are no instances of an addon. If an object is extended through an addon, the added capabilities cannot be distinguished from the inherent capabilities.

The relation object accepts request for termination of a relation. Again the conditions in the protocol will be checked and if these are satisfied, the relation is terminated. The actions are basically the reverse of those executed on the creation of the relation. This means that the capabilities added to the objects by the addons belonging to the relation are removed and the relation object ceases to exist. However, the protocol might state application-specific actions to be taken. For example, a bank might wish to keep some historical information on closed bank accounts.

Finally, the protocol in the relation objects defines the actions taken in exceptional cases. The most common exception is the situation where one of the partners in the relation ceases to exist. Conceptually the existence of the relation is dependent on the existence of the partners in the relation. Therefore the default protocol will probably be that the relation ceases to exist, if one of the partner objects dies. Again, application specific requirements might lead to another protocol.

### 2.4 Structure of the Object Model

The type structure in the autonomous data model is three layered. Object instances have a class as their type and the classes are typed by one of the three metaclasses. The three levels can be described as follows:

**Instance Level** At this level the instances of objects and relation objects live. These objects are the representation of the real world the system tries to model. This means that lines, squares and circles are at this level.

**Class Level** At the class level each class is represented by a class object. The class object takes care of creation and deletion of instances and keeps track of them during their lifetime. Class objects of relation classes have similar tasks as described in the previous section. Addons offering extensions of objects can also be found at this level.

Each object at the class level keeps track of the object instances it is involved with. This means that a class object or a relation class object knows the objects in their class. An addon keeps track of the objects that it has extended. This way all instances of a class can be accessed through the class object. This means that queries for objects of a certain class should be addressed to the class object of that class.

**Metaclass Level** The metaclasses are also present in the system as objects. The metaclass level is the highest level in the system. The main reason of their presence is to facilitate schema evolution. Through the metaclasses new classes can be added to the database and existing classes can be altered. Analoguous to requests for creating and discarding instances accepted by class objects, metaclass objects accept requests to create and discard classes. In the object model described in this section we see three metaclasses, viz. object classes, relation object classes, and object addons.

A sketch of this structure and what lives at each level is given in Figure 1.

## 3 Implementing Constraints in the Autonomous Data Model

In this section we show how constraints are dealt with in the autonomous data model. A constraint between two graphic objects is a relation between these two objects. We look at a number of different approaches to constraint solving, such as described for example in [12], and show that the autonomous data model can be used for all of them.

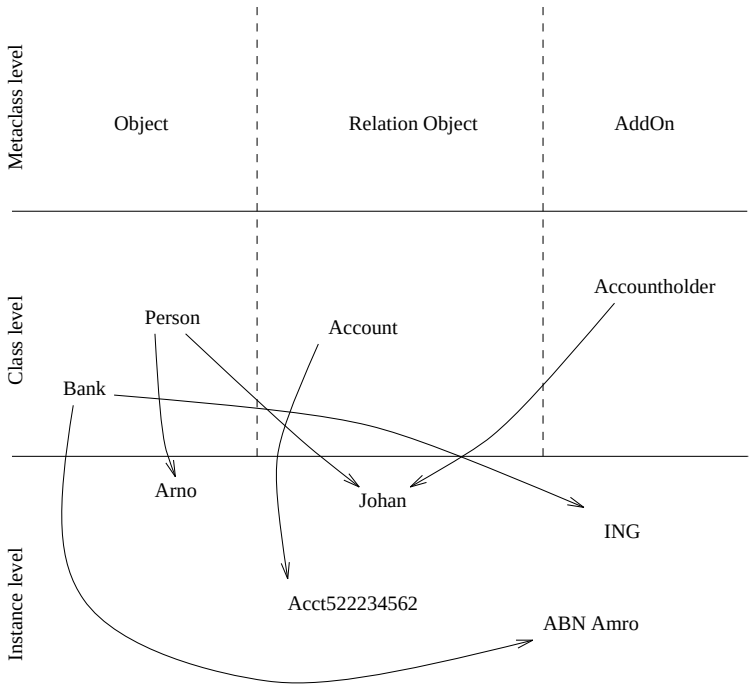


Figure 1: Structure of the object model

In the autonomous data model a constraint is a relation between two objects. When the constraint is created, a relation object is created that contains the actions to be taken to correct constraint violations. The constrained graphical objects are extended with rules and dynamic constraints to deal with the constraint relation. The rules define the reactions of the objects to changes in its state that might cause constraint violations. The dynamic constraints of an object are modified in such a way that it accepts requests to change its state from the relation object.

**Local Propagation** The simplest approach to constraint solving is a local one. This is modelled simply by having a relation for each constraint. Each relation attempts to satisfy its constraint locally. An example is given in Figure 2. The lines in this figure must maintain an angle of 90 degrees relative to each other. The strategy used to enforce the constraint is to change the angle of the other line with an equal amount.

To this end we create a relation between the two lines. Because the way the constraint is enforced is part of the behaviour of the constraint relation, the relation object contains the following rules to take action in case of changes of the angles of the lines.

**Relation Object** Constraint90

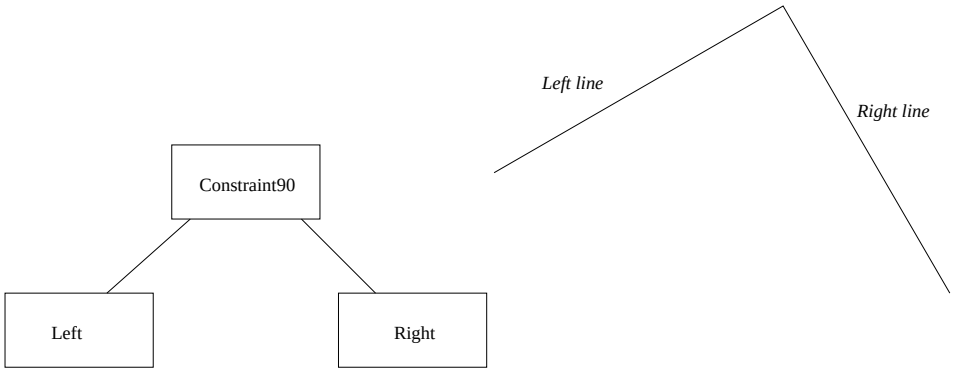


Figure 2: Perpendicular lines

**Attributes**
Left: GraphicObject
Right: GraphicObject
**Rules**
**On** Left.changeAngle(swing)
**if** Left.angle - Right.angle != 90
**Do** Right.changeAngle(swing);
**On** Right.changeAngle(swing)
**if** Left.angle - Right.angle != 90
**Do** Left.changeAngle(swing);
**EndObject**

The rules in this object trigger messages to the constrained objects if the constraints are violated by a change of the angle of the line. Therefore the dynamic constraints of the constrained objects are extended to accept angle setting from the constraint relation object.

**AddOn** ConstrainedByConstraint90
**Attributes**
constrainer : Constraint90
**Dynamic Constraints**
([sender=constrainer]angle)*
**EndAddOn**

If we apply this strategy to a collection of objects, each constraint is represented by a relation. This is illustrated by the example depicted in Figure 3, taken from [15]. Lines $a$ and $b$ must remain upright. Line $c$ has a fixed length. These constraints are all on the objects themselves. Therefore they are encapsulated with the objects in the autonomous data model. Two other requirements further constrain the behaviour of the lines. These are that one end of line $c$ must touch line $a$ and that the other end must touch line $b$.

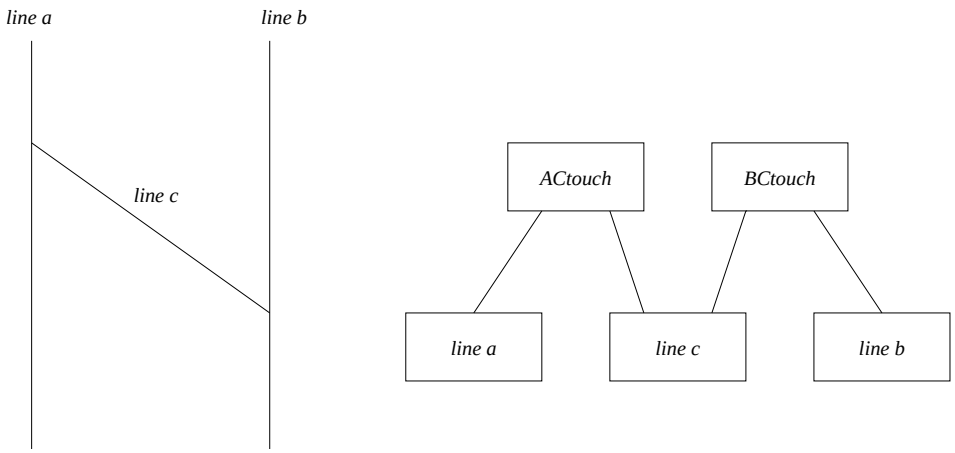


Figure 3: Local propagation with multiple objects

To enforce the constraints there is a relation *ACtouch* between line $a$ and line $c$ and a relation *BCtouch* between line $b$ and line $c$. Suppose that our constraint enforcement strategy is that we allow line $c$ only to be rotated around its midpoint. In that case the implementation of the relation object *ACtouch* will be to instruct line $c$ to rotate if line $a$ moves closer to line $b$. On this rotation *BCtouch* will instruct line $b$ to move closer to line $a$.

An alternative strategy is that we wish to keep line $b$ fixed and allow line $c$ to be translated. In that case we have a different implementation of the relation objects *ACtouch* and *BCtouch*. The reaction of *ACtouch* remains the same, but instead of instructing line $b$ to move, *BCtouch* will instruct line $c$ to translate in such a way that its endpoint remains on line $b$. This scenario implies another action to move line $a$ to the right again to keep touching line $c$.

**Coordination between constraints** In the previous two examples we used local constraint satisfaction. In some situations it might be desirable to coordinate the solution of a number of constraints. The notion of a relation is again central in the coordination of constraints. If we wish to coordinate the solution of a set of constraints in some way, we do so by creating a relation between those constraints.

This scheme works for all kinds of coordination. The coordinating activity is defined by the protocol of the coordinating relation. It may vary from solving the constraints simultaneously to giving one constraint priority over another. Prioritising constraints may be useful if there are multiple constraints between a pair of objects. An example is given in Figure 4.

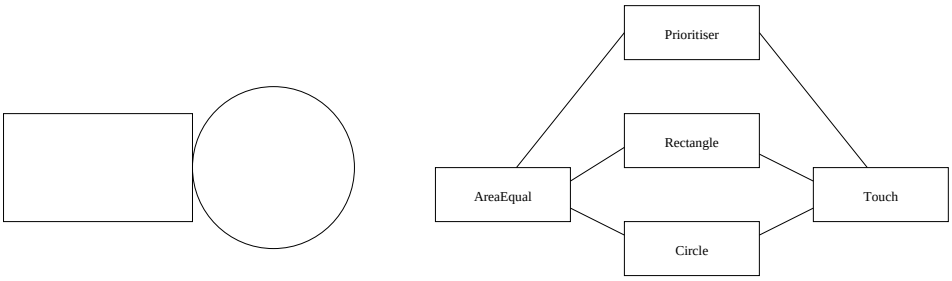


Figure 4: Coordinating two constraints

Here we have two constraints between a rectangle and a circle. The first constraint is that the areas of both figures must be equal. Secondly, both figures are required to touch. Suppose now the rectangle grows and consequently violates both constraints. Without coordination there are two possible scenarios to achieve renewed satisfaction of the constraints. The first possibility is to enlarge the area of the circle and then move it in order to have the figures touch again. The second possibility first moves the circle, then grows it and again moves it in order to restore the touching constraint.

The result is the same in both cases, but for reasons of efficiency we might prefer the first solution over the second. To achieve this we must establish a relation between the two constraints. This relation gives one constraint priority over another. To achieve this the rules and dynamic constraints of the area and touch constraints are extended to inform the prioritiser and accept synchronising messages.

**AddOn** OrderedConstraint
**Attributes**
guvnor : Prioritiser
**Dynamic Constraints**
(checkConstraint;PrioritiserOK;newSolution)*
**Rules**
**On** newSolution
**do** AskPrioritiser
**EndAddOn**

The prioritiser gives priority to the area constraint, if both constraints are violated at the same time. If both constraints ask the constraint prioritiser to go ahead, only one, the area constraint, will receive permission to do so.

**Relation Object** ConstraintPrioritiser
**Attributes**
first : Constraint

second : Constraint
**Rules**
**On** first.AskPrioritiser & second.AskPrioritiser
**do** first.PrioritiserOK;
**EndObject**

Another situation where regulation of local propagation can be useful is in the situation depicted in Figure 5. Here a node indicates a graphical object. An edge between two nodes denotes the presence of a constraint between the two graphical objects represented by the nodes.

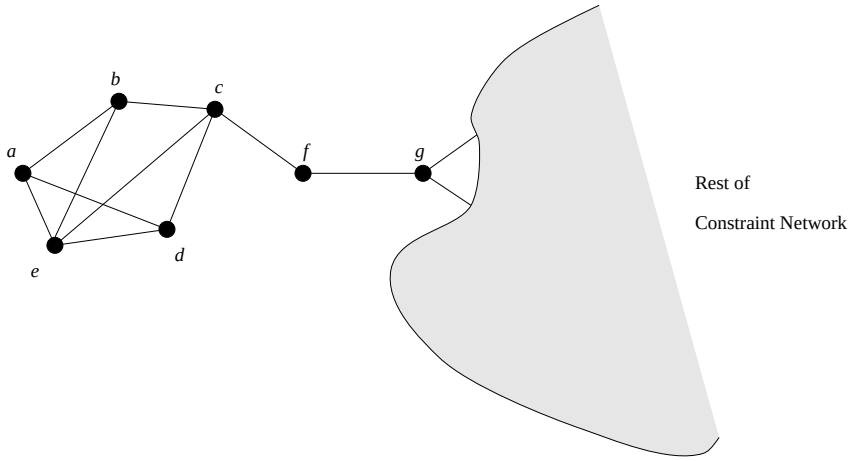


Figure 5: Blocking propagation from part of a constraint network.

In this situation a change in one of the objects $a$ to $e$ that violates a constraint, triggers local propagation of new values. Because the subgraph is highly connected, it may take some time before a stable state satisfying all constraints is found. It may be desirable to prevent any attempts at satisfying constraints outside this subgraph before it has reached a stable state. In other words, constraint $c$-$f$ can only be solved after $b$-$c$, $e$-$c$ and $d$-$c$ have been solved. This is a metaconstraint, so it is implemented by establishing a relation between these four constraints.

**Global Constraint Solving** The autonomous data model does not force a constraint solving strategy on the programmer. We can use it equally well to implement a global constraint solving strategy. In this strategy, we have one central constraint enforcer, that checks for constraint violations and generates a new state for the graphical objects in the system.

With the view of constraints as relations limiting the behaviour of autonomous objects in mind, we look at what a global constraint solver does

relative to the constraints. If one of the constraints is violated, the constraint solver takes the current state of the system and generates a new state that does satisfy the constraints. A constraint in this situation is a relation between the constrained objects and the global constraint solver. The relation checks for violation of its constraint and, if necessary, passes data on to the constraint solver.

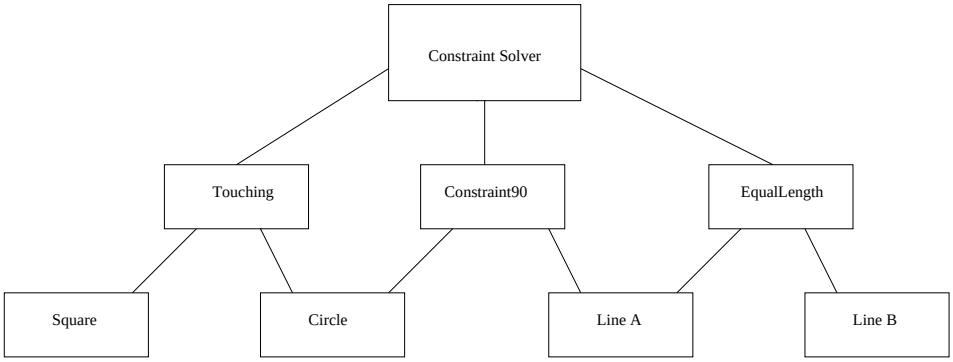


Figure 6: Global constraint solving with autonomous objects

# 4 Concluding Remarks

As we have shown in the previous section, autonomous objects offer a very flexible framework for implementation of constraint solving strategies. The unavoidable violation of attribute encapsulation is regulated by the constraint relations. The definition of a relation in a protocol gives us an exact description of the workings of a relation.

The autonomous data model simplifies the way we deal with constraints in a system. By offering a view of constraints as relations between objects we emphasise the local character of a constraint. Consequently, problems are solved at the place where they occur. This approach promotes a "divide and conquer" strategy to solving problems in constraint satisfaction, which in general favours simpler solutions.

Constraint maintenance is mostly declarative in the autonomous data model. Detection of possible violations, checking for them and corrective action can all be defined in the rules. However, because the autonomous model offers a superset of functionality of a conventional object-oriented system, it allows other ways to implement constraint enforcement.

Because of the addon mechanism, the implementation of the graphical object is separate from any functionality needed for the constraint mechanism. We can implement, for example, a pentagon without needing to worry about the

possible constraints on the pentagon. Therefore, new types of constraints can be added to the system at any time. It also means that each object in the system only carries the functionality it currently needs.

In addition a data model can serve in an implementation as a uniform representation. This way we can avoid using custom-made representations for certain constraint solving strategies. This facilitates the combination of multiple strategies and the interoperability of software components.

The contribution of database research to computer graphics research in general, and the area of constraint programming in particular, can be found in data models. A data model can help in getting a good conceptual understanding of a problem through a clear description of the data involved. We have shown in this paper that the view of constraints between graphical objects as relations between autonomous objects leads to simple, mostly local, solutions to problems encountered in dealing with constraints.

## References


[1] J.F.P. van den Akker and A.P.J.M. Siebes. A data model for autonomous objects. Technical Report CS-R9539, CWI, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1995. Available by ftp (ftp://ftp.cwi.nl/pub/CWIreports/AA/CS-R9539.ps.Z).

[2] F. Arbab, I. Herman, and G.J. Reynolds. An object model for multimedia programming. Technical Report CS-R9327, CWI Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1993.

[3] F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency, Practice and Experience*, 5(1):23-70, 1993.

[4] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1990.

[5] Jan van den Bos and Chris Laffra. PROCOL: A concurrent object-language with protocols, delegation and persistence. *Acta Informatica*, 28:511-538, September 1991.

[6] Stefano Ceri and Jennifer Widom. Deriving production rules for constraint maintenance. In D. MacLeod, R. Sacks-Davis, and H. Schek, editors, *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 566-577, 1990.

[7] U. Dayal et al. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record*, 17(1):51–70, March 1988.

[8] Umeshwar Dayal, Alejandro P. Buchmann, and Dennis R. McCarthy. Rules are objects too: A knowledge model for an active, object-oriented database system. In *Advances in Object-Oriented Database Systems*, pages 129–143, Berlin, Germany, September 1988. 2nd International Workshop on Object-Oriented Database Systems, Springer.

[9] Bjorn N. Freeman-Benson and Alan Borning. Integrating constraints with an object-oriented language. In O. Lehrmann Madsen, editor, *Proceedings of the 1992 European Conference on Object-Oriented Programming (ECOOP'92)*, LNCS 615, pages 268-286, Berlin, Germany, 1992. Springer.

[10] Stella Gatziu, Andreas Geppert, and Klaus R. Dittrich. Integrating active concepts into an object-oriented database system. In Paris Kanellakis and Joachim W. Schmidt, editors, *The Third International Workshop on Database Programming Languages: Bulk Types and Persistent Data*, pages 399-415, San Mateo,. CA, USA, August 1991. Morgan Kaufmann.

[11] Eric N. Hanson. An initial report on the design of Ariel: A dbms with an integrated production rule system. *SIGMOD Record*, 18(3):12-19, September 1989.

[12] Wm Leler. *Constraint Programming Languages: their specification and generation.* Addison-Wesley, Reading, MA, USA, 1988.

[13] Michael Stonebraker. Triggers and inference in database systems. In M.L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems*, chapter 22, pages 297-314. Springer, 1986.

[14] J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of the 1990 ACM SIGMOD Conference on the Management of Data*, pages 259-270, 1990.

[15] D. Zeleznik et al. An object-oriented framework for the integration of interactive animation techniques. *ACM Computer Graphics*, 25(4):105-112, 1991.