CWI Quarterly Volume 8 (4) 1995, pp. 329 – 354

Unconstrained Constraint Programming

J.E.A. van Hintum
CWI, Department of Interactive Systems
P.O.Box 94079, 1090 GB Amsterdam, the Netherlands
e-mail: hansh@cwi.nl

The expressive power and usability of a constraint system depends on two things; the degree in which the constraint system is suited for the environment it is used in and the ease with which a programmer may adapt the behavior of the constraint system to his own needs. This article will show that the MADE constraint system is very well suited for the environment it is designed for (multimedia applications) and that the programmer is able to adapt the behavior of the constraint system on all its three important tasks: triggering, propagation and satisfaction. A central role in this article is played by the constraint network as this is also the central part of the constraint system. The more a programmer is able to tune the action performed by the constraint system on the constraint network, the more expressive constraints can be defined by that same programmer.

1. Introduction

This paper discusses the constraint system developed for the MADE programming environment. Constraints can be of great practical use in multimedia applications, but only if they are designed correctly for use in the multimedia area. Then, a multimedia constraint system may significantly reduce the amount of work needed to construct a multimedia application; constraints have to be declared and defined once and are maintained automatically in every situation and at all times. This improves readability and maintenance of the application. MADE is a dedicated object oriented, multimedia programming environment using the programming language MC++ (based on the C++ programming language) which provides the programmer with tools and utilities to write multimedia applications easily. For a detailed description of MADE the reader is referred to [1, 2, 13, 16].

The objective of the MADE constraint system is to provide constraints which do not require any adaptations to the objects which are to be constrained. This means, that whenever a programmer wants to constrain a number of objects, the specification of these objects remains untouched. The programmer only has to define the new constraint objects and activate the constraint system. This can be realized by delegation, which allows for the dynamical change of the behavior of a member function without the requirement that the code of the original function has to be adapted or prepared for this[1].

This discussion will be focused on constraint networks as these constraint networks belong (among others) to the most important parts every constraint system. In section 2 the general concepts of constraint networks and satisfaction methods within these networks are presented. Section 3 will relate these general concepts to the requirements for constraint systems that stem from multimedia environments. In section 4, a concrete constraint system for a multimedia environment, the MADE constraint system, is discussed in detail. Subsections will, consequently, deal with the terminology, construction, propagation, triggering, solving and synchronization in and of the constraint network. In section 5 some conclusions are drawn from this text.

2. Constraint Systems in General

Constraints specify dependency relations between 'things'. The nature of these 'things' very much depends on the environment in which the constraints are used. Typical areas in which constraints are used are *user interface control* [5, 23] (with check-buttons, radio-buttons, bars, boxes, etc.), *geometric layout* [24, 25, 26] (with circles, rectangles, lines, points, etc.), *animation* [5] (with timetables, sprites, still images, palettes, etc.) and *media synchronization* [3, 12] (with timetables, media objects, error functions, etc.). However, the precise nature of these 'things' is not relevant for this discussion; in the remainder of this text, we will address these 'things' as objects and assume that each object has at least one property which has a value that can be changed in some way by another object. A dependency relation is assumed between the values of at least two properties. A dependency relation which is maintained automatically by the *constraint system* is called a *constraint.* This maintenance of the constraints is by the constraint system is done using *constraint objects.* Constraint objects are imperative reflections in a programming language of the abstract notion constraint. Most constraint systems have organized the constraint objects in one or more networks (which are called *constraint networks* or *constraint graphs*). Often, the objects whose properties are *constrained* (i.e. the objects from which a property value is involved in a dependency relation) are incorporated in the constraint network as well. Whenever the value of a property which is incorporated in the constraint network is changed, the constraint system has to make sure all constraints still hold. If this is not so, the constraint system has to *satisfy* the constraints. Because of this constraint satisfaction, the values of other properties may change; when their objects are stored in the constraint network, the constraint system may have to satisfy

[1] In the MADE environment all member functions can be delegated and delegated to.

other constraints in which these objects are involved as well. A constraint network is said to be *over-constrained* if not all constraints in the network can be satisfied due to conflicts in the dependency relations. A network is said to be *under-constrained* if the whole network of constraints can be satisfied in more than one way.

Nowadays, several constraint systems exist. These systems differ mostly in the type of constraint network they build and in the methods they use to satisfy the constraints in the constraint network (*satisfaction methods*). In the remainder of this section we will discuss several satisfaction methods (section 2.1) and different kinds of constraint networks (section 2.2).

2.1. Classification of Satisfaction Methods

At present there are six prevailing satisfaction methods [4, 8, 9, 10, 21, 22] (ignoring the satisfaction methods which are used in Artificial Intelligence [14, 15]). In this section each of these methods will be presented.

Propagation of known states (or **Local Propagation**) is a satisfaction method which propagates the results of constraint satisfaction one by one through the nodes of the constraint network. Propagation of known states is a satisfaction method which looks for one-step deductions. The constraint system will look for constraints which can deduce new property values using only the already satisfied constraint objects in the constraint network (initially no constraint object is satisfied and only the changed property value can be used). These constraints will then be satisfied and added to the part of the constraint network that is identified as the part where the dependency relations hold. This process continues until all constraints are known to be satisfied. If, however, not all constraints are satisfied and none of these constraints can be satisfied in one step, this satisfaction method will fail (and another, stronger method should be used).

Propagation of Degrees of Freedom is a method which tries to prune as many constraints as possible from the constraint network. The rationale behind this scheme is the following; when the values of the properties of certain objects in the constraint network only depend on a few other property values and if the former values can be computed once the latter collection of values is known, all former objects (and their constraint objects) can be pruned from the constraint network. The constraint system will now only consider the smaller (pruned) constraint network. This reduced network can be pruned repeatedly until no part of the network can be found to prune. The remaining constraint network then has to be satisfied using another technique after which all the pruned parts can be satisfied (in reverse order).

(**Numerical**) **Relaxation** is a technique which is, usually, used when other techniques (like propagation of known states) fail to solve the problem.

This technique is also often used to solve cycles in the constraint network. A prerequisite for the constraints to be *relaxed* is that they can be approximated by an equation (otherwise, even relaxation may fail to find a solution and enter a never-ending loop). Relaxation starts with guessing an initial value for a property value. Using this initial value, the constraint system satisfies the constraint network. After that, the constraint system estimates the error of the initial guess (using the initial property value and the computed property value) and adjusts it using a linear function. The adjusted value is taken as new initial guess and the constraint system starts satisfying the constraint network again. This process is repeated until the estimated error reaches a level lower than some predefined threshold. Beside the fact that this approach may take a long time, it may find only one solution (where several may exist).

Redundant View is not really a technique; it does not define a way in which information is propagated through a constraint network. A constraint system which supports redundant views can choose among alternatives. Redundant views are different descriptions for the same dependency relation. These redundant views may be helpful in two ways; 1) the constraint system may deduce several property values from one other property value (given the relations C = K - 273, C = 5/9(F - 32) and a value for C, the values of both K and F may be computed) and 2) the constraint system may combine several dependency relations in order to deduce other property values (given the above mentioned relations and a value for K, then the value of F may be deduced from K = (5/9(F - 32) + 273).

Prototyping is a technique that is based on the class hierarchy system. A prototype contains methods and data which are used by several instances of the prototype. By changing a method of a prototype, the behavior of all the instances of the prototype is also immediately changed. Similarly, the data of a prototype can be used to represent common properties. Changing this data, the properties of all the instances are changed. Using one of these methods, it is possible to do constraint programming; the behaviour of the instances can be constrained by changing the methods or the values of the data of the prototype. A difference with the class hierarchy mechanism is, that there is no difference between the generator (i.e. the class) and the generated objects (the instances); a prototype can be generator as well as generated object.

Graph Rewriting and **Term Rewriting** are both techniques which are based on the dynamic alteration of (a copy of) the constraint graph. Using the rewrite technique, the constraint system starts with that part of the constraint graph which has been changed. The constraint system tries to match subgraphs against a set of predefined *rewrite rules.* When the left-hand side of such a rule matches the subgraph, that subgraph is replaced by the right-hand side of the rewrite rule. This process is repeated until no rule matches anymore. Rewrite rules may contain variables: patterns which are filled in during matching and applied during rewriting. The difference between term rewriting and graph rewriting is the fact that graph rewriting can share subgraphs; within a graph several pointers to the same subgraph may exist. When this subgraph is rewritten once, it is rewritten for all its referees. Term rewriting does not know the notion of subterm sharing; if the same subterm occurs more than once in a particular term, this subterm has to be rewritten for every occurrence[2].

Constraint systems may combine several of the above mentioned methods to be able to solve a variety of constraint networks. *Redundant Views* and *Relaxation* even require the presence of another technique because those methods will, generally, not be able to solve arbitrary networks on their own. When a constraint system has several satisfaction methods at its disposal, it may sometimes choose which method to apply and even try another method when the first choice fails.

2.2. Classification of Constraint Networks

Constraint networks used by the different constraint systems can vary in several aspects [6, 9, 10, 11, 21, 22, 27]. Most of these aspects deal with the nature of the dependency relations. In the remaining part of this section, important aspects of constraint networks are discussed.

Dependency relations can be **directional/a-directional**

The distinction between directional networks and a-directional networks is also known as the difference between uni-directional and multi-directional or one-way and multi-way systems. In directional (uni-directional) systems, the dependency relations between objects are directed: the property value of one object influences the property value of another object (but not vice versa). In a-directional (multi-directional) systems, the dependency relation has no specific direction: the property values of the different objects can influence each other; they are mutually dependent.

The fact that a constraint system supports a-directional constraint networks does not automatically imply that directional relations are supported as well; directional and a-directional networks represent two totally different approaches.

Dependency relations **may/may not form a cycle** in the constraint network

Some constraint systems support cyclic structures in the constraint network. Usually, the constraint system provides tools to satisfy the cycle and tools to break the cycle.

[2] Note that Graph Rewrite Systems may produce different results compared with Term Rewrite Systems. Consider the rewrite rules

`arg` $\rightarrow$ `1;`   `arg` $\rightarrow$ `2;`   `farg` $\rightarrow$ `arg + arg`

For `farg` the GRS may produce either the result 2 or 4 whereas the TRS may produce either 2, 3 or 4 as both subterms `arg` are rewritten independently.

Dependency relations **are weighted/have all equal priority**

In non-priority systems (where the dependency relations have equal priority) constraints are satisfied without any distinction between the different constraints. In priority systems, the goal of the constraint system is to satisfy the constraints with the highest priority first. After this is done constraints with a lower priority are satisfied. The priority of a constraint may vary from two to infinity. When a constraint network is over-constrained, the system will try to satisfy at least the most important constraints (those with the highest priority). In case of an under-constrained network, no unique solution exists. Often the *Result of Least Astonishment* is used; property values are chosen such that their value stays as close as possible to the original value.

Dependency relations are **1-1, 1-m** or **m-m**

Some constraint systems allow only dependency relations that connect one object on the one hand and a second object on the other hand (1-1 relations). These systems are sometimes called single-dependency systems. Multi-dependency systems are systems which allow dependency relations between one object on the one hand and several other objects on the other hand (and vice versa) or a number of objects on the one hand and several objects on the other hand (1-m, m-1 and m-m relations). The differentiation between 1-m and m-1 relations can only be made in directional systems.

A network is **static/dynamic**

When the constraints in the network have to be specified in advance (i.e. at compile-time) the system is called a static constraint system. In static constraint systems, constraints may neither be added or deleted nor activated/deactivated at runtime. In dynamic constraint systems, constraints may be added, deleted, activated or deactivated at any moment at runtime. Because of that, the constraint system must adjust the other constraints in the network when one of the above-mentioned operations is performed.

A network is solved **incremental/at once**

An incremental constraint system will solve the constraints one by one; i.e. first one constraint is considered and satisfied before the system looks at any of the other constraints. When a constraint is solved, the conditions of the other constraints are *narrowed.* This means that the number of possible solutions to satisfy the other constraints is reduced (similarly, when a constraint is disregarded to search for other possible solutions, the conditions of the other constraints are *widened*). Other constraint systems may consider all constraints at once and look for a solution which satisfies all the constraints immediately (e.g. a system which tries to find a solution for a set of (in)equations using matrices).

A constraint system uses **one/several constraint networks**

Some constraint systems make a distinction between the network which contains the dependency relations between the different objects and the network which contains the values computed by the constraint system. In that case, the former network contains control information whereas the latter network is used as data/scratch area.

As with the satisfaction methods, constraint systems may combine several of the aforementioned aspects.

3. Constraint in Multimedia

The aspects of constraint systems mentioned above are a summary of available types of constraint systems. In this section the relevance of the different types of systems to multimedia applications is examined. Multimedia applications deal with two notions of information: multimedia data and multimedia information. Multimedia data consist of the raw media chunks (physical entities) whereas multimedia information defines the context in which the multimedia data has to be used (logical entities). Multimedia applications try to define, manipulate and present spatial and temporal information. Multimedia information defines the when, where and how. Multimedia data defines the what [7]. Constraints work on the multimedia information to help the programmer with the definition, manipulation and presentation of multimedia data.

Constraints in multimedia applications have special demands regarding the constraint system. Because multimedia is an area in which many different kinds of objects need to be managed simultaneously, the maintenance of the relations between all those objects can become a complex and difficult task. Consequently, it is more advantageous if multimedia constraints run *parallel* with the ongoing presentation and disturb this presentation as little as possible. This will ensure a presentation of the multimedia data that is as smooth as possible. There is another reason for constraints to be solved in parallel. As different parts of the multimedia presentation may run parallel to each other, the constraint system should disturb this parallelism as little as possible and should therefore solve the constraints per parallel part of the presentation in parallel. Furthermore, as multimedia presentations are often interactive, multimedia constraints should be capable of *interactively changing the status of the constraints and the constraint network.* A further consequence of the interactive nature of multimedia applications is that the constraint system must be *dynamic*, i.e. constraints may be added, deleted, activated and deactivated at runtime. Another important requirement is that dependency relations are best formulated in a declarative way, whereas the usual representation in the programming language is imperative. Therefore, it should be possible to specify multimedia constraints in a declarative way, after which the constraint system translates the declarative dependency relations into an imperative equivalent.

From the observations outlined above and from the summaries presented in the previous sections, the following evaluation of the different satisfaction methods for multimedia applications can be made:

- The method known as **Propagation of Known States**, in contrast with most of the other methods, can solve constraints in a parallel environment. Because multimedia applications often use parallel execution schemes, propagation of known states seems a logical choice. Propagation of known states is very well suited to interact with the user of a multimedia application on the different actions to be taken by the constraint system. Changes to objects due to constraint satisfaction occur one by one and the user can easily follow the propagation (and changes of the different objects) within the constraint network.
- **Propagation of Degrees of Freedom** is a method which requires the constraint network not to be altered during constraint satisfaction. When a constraint is satisfied, no other constraints may be added, deleted or (de)activated in the constraint network. This restriction means that this method can hardly be used in a parallel environment. The main application would have to be stopped during constraint satisfaction. In multimedia applications this would mean that first the presentation is stopped, then the constraints are satisfied and finally the presentation is resumed. Such an approach would lead to very jerky presentations.

  Another drawback of this method is that interaction would be more problematic as whole subgraphs are satisfied at once. This would mean that the user cannot get a clear insight in what is going on the the constraint network (even if such an insight is desired).
- The method of **Relaxation** requires its constraints to be numerical. However, constraints in multimedia applications can be applied to all kinds of objects and the demand to force all multimedia constraints to be translated to numerical dependencies would restrict a multimedia constraint system too much. Relaxation tries to change the current situation (in which some constraints are invalidated) smoothly into a situation where all constraints are satisfied (which, hopefully, leads to a smooth adaptation of the presentation). However, relaxation is a time-consuming technique to obtain constraint satisfaction and this would leave the presentation in an invalid situation for too long.
- **Redundant Views** can be used very well in a multimedia constraint system. They may even help the constraint system to find solutions for complex problems. However, it is the programmer who has to provide the multiple views. The constraint system cannot, in general, deduce redundant views itself (this would lie in the scope of AI).
- **Prototyping** can also be used in multimedia constraint maintenance. However, prototypes only provide support for simple constraints. All objects generated from a particular prototype share the same behavior. Its main use is for some kind of equivalence constraint (variables having the same value, functions having the same functionality, etc.).

– **Graph and/or Term Rewriting** may be used by a constraint system in a multimedia environment. However, rewriting, like relaxation, is a very time-consuming technique.

It is not only the particular satisfaction method that may be more or less suitable for multimedia constraint satisfaction, but the different constraint networks may also have their pro's and cons with respect to multimedia constraint satisfaction. The next paragraphs present the minimal requirements for a constraint network in a multimedia environment. From these requirements it is possible to deduce what kind of features a constraint network, which is used in multimedia environments, should support.

– Constraints in multimedia applications should support both, directional and a-directional constraint networks. Directional constraints are necessary to model directed dependency relations, i.e. one property value depends on another property value but not vice versa. These kinds of relations are, for example, necessary to model a grid (the position of the objects depends on the grid size, but the grid size is not dependent on the location of the various objects). A- directional constraints are needed to define interdependent relations, i.e. property values are dependent on each other. An example of a- directional constraints is the multiple view situation, where the same information is presented in more than one way. When this information is changed in an arbitrary view, the other views must be adapted accordingly.
– It must be possible to form cycles of constraints in the constraint network. When a presentation is made, this presentation is constructed using several media objects like sound, video etc. In most cases, the duration of the different objects is related to the duration of another set of objects. Even very simple presentations already have to deal with cyclic dependencies with respect to the duration of the different objects.
– Priority systems are very helpful in constraint systems. Due to the generally large amount of constraints in multimedia systems, there are bound to be constraints which are in conflict with each other. A priority system is a way of resolving these conflicts.
– Dependency relations in multimedia systems should be m-m relations. This means that a set of objects may be connected to several other objects via a constraint object.

  The multiple view is an example of the use of m-m relations. It may be obvious that multiple view objects are 1-m relations: the same information is presented in several ways. However, multiple view require the stronger m-m relations. It is rather common that the information to be presented is made out of information collected from several objects. Therefore, there is a need for m-m relations.
– Dynamic constraint systems are almost vital for multimedia systems. As the multimedia applications are by nature very dynamic, the constraints

imposed on the different parts of the program should be as dynamic as possible. This will restrain expressive power of the constraints as little as possible.

- An incremental constraint system is probably the only real constraint system which produces workable results because of the often complex structures in multimedia applications. Trying to solve the constraint network all at once may be to time-consuming or even be impossible.

4. The Theory of Constraint Satiafaction in Made

In the previous sections the general aspects of contraint systems in multimedia environments were discussed. At this point, the attention will be focused on a concrete system, the Made constraint system.

In this section, first the concepts of the Made constraint system are discussed (4.1), together with the rationale of the different choices made in the design of the constraint system (4.2). After that, the building (4.3), triggering (4.4), propagation (4.5, 4.6), solving (4.7), anti-triggering (4.8) and synchronization (4.9) of and inside the constraint network is presented.

4.1. Concepts in the Made *Constraint System*

Constrainable objects in Made can be added to the constraint graph in two ways: as *dependent objects* or as *independent objects.* Independent objects are objects which may *trigger* a constraint object, i.e. they may indicate to a specific constraint object that the constraint maintained by that object may need to be re-satisfied because changes to the property values of the independent object may have an impact on the property values of other objects. These latter objects are called the *dependent objects.* It is important to note that an independent object of one constraint object can be a dependent object for another constraint object.

Whenever an independent object triggers a constraint object, the changes to the property values are propagated by that constraint object to the dependent object(s). This propagation can be done in two ways, using either *eager constraints* or *lazy constraints.* Eager constraints will propagate changes of independent objects immediately to dependent objects. Eager constraints are especially useful when changes have to have an immediate effect (for instance, when a property needs to be displayed on an output device (like a screen, a speaker, etc.)). Lazy constraints can be used as some kind of buffer to the propagation. They will only propagate changes to the dependent objects if a dependent object requests such an update (if a request is issued but no propagation is buffered, the dependent object will proceed as if the request was never made). Situations where lazy constraints can be useful is where objects have to be aligned to a grid. When the grid-size changes, only newly placed objects use the new grid-size and are aligned to the new grid; already placed objects are unchanged.

The constraint network in MADE is merely a means to propagate changes of the independent objects to the dependent objects. *Propagation* is one of the three main tasks of a constraint system. Another task is the *triggering* of the constraint objects. Because MADE provides an object oriented environment, the objects in MC++ have to obey the OO paradigm; changes to property values have to occur via the object's interface. Thus every time a constrained property value is changed, a member function of an independent object is invoked. This enables the mechanism to trigger the constraint object. The member function of the independent object which changes the property value, and consequently triggers a constraint object, is called a *triggering member function.* Dependent objects *anti-trigger* the constraint object via the *anti-triggering member function.* Anti-triggering a constraint means that a dependent object makes a request to a constraint object to update its property values if necessary.

The actual maintenance of the constraints (i.e. adapting the property values of the dependent objects when the property values of the independent objects have been altered) is implemented by using *delegation.* The triggering member functions of the independent objects and the anti-triggering member functions of the dependent objects are delegated to special member functions of the constraint object (so-called *shadow functions*). When the application invokes the (anti-)triggering member function, the runtime system will invoke the shadow function instead. The purpose of these shadow functions is to enable the constraint system to keep administration of the constraints in the constraint network up to date. There are two types of shadow functions: *independent shadow functions* (shadow functions for triggering member functions) and *dependent shadow functions* (shadow functions for anti-triggering member functions).

The shadow function perform several actions. One of these actions is the execution of the original (anti-)triggering member function. Furthermore, the shadow function will activate a *constraint function.* The constraint function is part of the third main task of the constraint system: *satisfaction.* In the constraint function, the programmer specifies the actions which must be performed to maintain the dependency relation. Each constraint object must have at least one constraint function associated with it.

4.2. Characterization of the MADE *Constraint System*

The constraint system in MADE is an a-priority dynamic incremental system, using local propagation, that is capable of solving m-m, a-directional, directional and cyclic relations. It combines the object oriented programming paradigm, inherited from the MC++ language, and the declarative constraint programming with multimedia considerations. This approach implies that the constraint system has characteristics from both paradigms and has to make concessions to both paradigms.

The MADE constraint system uses the satisfaction method called *propaga-*

tion of known states. Although this may not be the most powerful satisfaction method, it is the best possible option when the constraint system tries to solve multimedia related constraints between objects which run parallel in different threads of control. It would be impractical to stop all objects for a while to do constraint solving. This would be necessary if any other method than propagation of known states is used.

The constraints in one thread are solved in parallel as much as possible too. To do so, the constraint system creates separate threads for each constraint to solve. Once the constraint is satisfied the thread is destroyed.

The constraint system supports both directional and a-directional dependency relations. Both directional and a-directional dependencies are realized in the following hybrid approach. All dependency relations are directed in the sense that the connection between (in)dependent objects and a constraint object is always directed towards the constraint object. Invocation of the (anti-)triggering member functions always activates the constraint object (due to the delegation mechanism). The a-directional aspect is realized by special constructs which allow a constraint object to check which independent object has last triggered the constraint object. Depending on that result, the appropriate actions can be taken.

It is also possible to define cyclic constraint networks in MADE . Constraints in cycles are handled in a special way. MADE does not use relaxation (because relaxation is essentially limited to numerical problems). Instead, the programmer has to define the actions which have to be performed during the different satisfaction iterations of the cycle. During each iteration, different actions may be taken. As the constraint system does not know when a cycle is solved, it is the responsibility of the programmer to make sure that termination is assured.

The constraint network can be altered dynamically. Constraints may be added and deleted at runtime. The constraint system furthermore also supports m-m dependency relations.

4.3. Building the Constraint Network

The first step for a constraint system is to construct a constraint network on which it can work. An application programmer must provide this information for such a constraint network. For this purpose the programmer must supply the following:

- a programmer must supply the various constraint class definitions which define the various independent and dependent shadow functions which are supported by the instances of this constraint classes.

This information only has to be provided once for a multimedia application. The following information has to be provided for every dependency relation which is to be constructed:

- a new constraint class instance and the name of the constraint function which is going to be the active constraint function for that instance has to be specified.

- for each independent object of this constraint object the triggering member function and the name of the independent shadow function must be given. The name of the independent shadow function is optional. If not provided the constraint system will assume the same name for the triggering member function and the independent shadow function.
- for each dependent object of this constraint object the programmer must specify the anti-triggering member function and the name of the dependent shadow function. The name of both the anti-triggering member function and the dependent shadow function are optional. If, however, a dependent shadow function is specified, the programmer also has to specify the name of the anti-triggering member function. If the name of the anti-triggering member function is specified and no name for the dependent shadow function is provided, the constraint system will assume the same name for the anti-triggering member function and the dependent shadow function.

Using this information, the constraint system is able to determine all the necessary information which is needed to trigger, propagate and solve constraint networks:

Triggering:

- set up the delegation between the triggering member function and the associated independent shadow function (thus, whenever the triggering member function is called, the independent shadow function is invoked instead).
- set up the delegation between the anti-triggering member function and the associated dependent shadow function (thus, whenever the anti-triggering member function is called, the dependent shadow function is invoked instead).

Propagation:

- store which independent and dependent objects are connected to which constraint objects.

Satisfaction:

- store the name of the active constraint function.

In figure 1, these three tasks of the constraint system are visualized. The cylinder- shapes represent the constraint objects. The boxes in these cylinders show the available constraint functions. The checked box represents the constraint function which is active. The active constraint function determines the way in which a constraint object reacts on the triggering of an independent object. The arrows in the diagram represent the dependency relations between independent/dependent objects and constraint objects. This information is used to plan how propagation flows through the constraint network. The single lines between the different member functions show which member function is delegated to which other member function. These delegations are there to be able to trigger the constraint object.

To make the maintenance of the constraint network somewhat more efficient, the total constraint network (from now on referred to as the global network) can be split up in several smaller constraint networks (the sub-networks). Every time a constraint object, independent object or dependent object is added to the global constraint network, the constraint system determines whether the object is already present in one of the existing sub-networks. If so, the new object is placed in that existing sub-network. If not, a new sub-network is created. The reverse, that two sub-networks need to be merged into one sub-network if an object is linked into another sub-network, is also possible. In other words, the global constraint network is divided into disjunct sub-networks. Although this approach creates a little overhead when an object is added to the constraint network, it can reduce the time needed to satisfy (part of) the sub-networks.

After the constraint network is in place, the constraint system is ready to be triggered by an independent object.

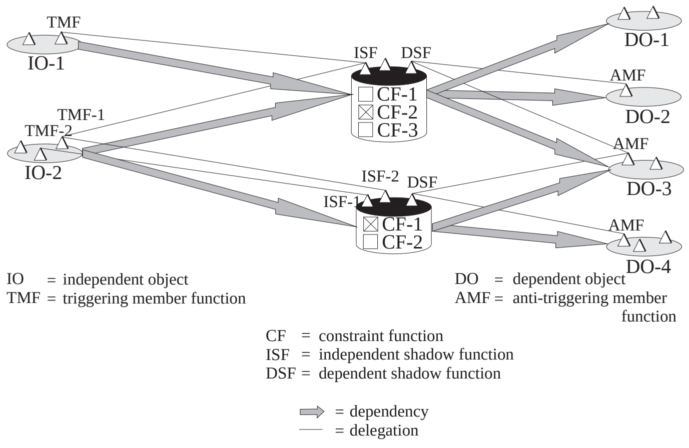

FIGURE 1. A Constraint Network.

4.4. *Triggering the Constraint Object*

Whenever a triggering member function is invoked, not this function itself but the independent shadow function will be executed. The triggering member function is delegated to the independent shadow function. The shadow function will perform a call-back to the original independent triggering member function. The reason for this procedure may be clear: before the constraint system starts to satisfy the constraints, the changes to the independent object should already have taken place.

After the independent member function has returned control to the shadow function, the shadow function now has to inform all the other constraint objects to which the original triggering member function is connected that they are invalidated (and thus realizes the multiple dependency in the constraint network). Invalid constraint objects block the dependent shadow functions. As long as these functions are blocked, the dependent object cannot release any data. This construction is used to guarantee the dependent object does not release any data which will soon be out of date. After the constraint is satisfied, the constraint system will validate the constraint object once again. Only the constraint objects which are directly connected to the independent object are notified. This approach is taken because only a subset of all the objects which might become invalid during satisfaction will actually be invalidated. Invalidating all constraints would block too many objects in the network. It would also, in the case of a parallel environment, destroy most of the parallel execution of the other objects.

After giving notification to the other constraint objects, the shadow function triggers the constraint system (which continues in a separate thread) and returns control to the original caller of the triggering member function.

4.5. Propagating changes in Non-Cyclic Constraint Networks

Once the constraint system is triggered, it starts to propagate the changes. Because non-cyclic constraints are considered in this section, it is assumed that there are no a-directional dependency relations in the network (an a-directional dependency relation can be considered to be a cyclic dependency where the cycle is formed by the object connected to the constraint object once as independent object, the constraint object itself and the connected object as dependent object again). Cyclic constraint networks (and also a-directional dependencies) will be discussed in the next section (4.6).

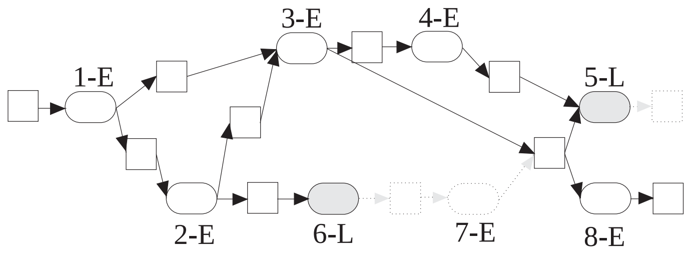

FIGURE 2. An example of a constraint network with only directed constraint objects. The Boxes are independent/dependent objects and the cylinders are constraint objects. The arrows show the direction of propagation. Dotted objects are not reached by propagation.

Propagation of changes in non-cyclic constraint networks is done using a simple rule: constraints are satisfied in such an order that they do not have to be satisfied twice. Consider figure 2, where each constraint object is labeled with a number and a letter. The letter indicates whether the constraint is eager (E) or lazy (L). The number is just for identification. The constraint system is triggered when changes occur in the object on the left hand side. If constraint object 2-E needs to be satisfied due to changes of property values of 1-E, constraint object 2-E will have to be satisfied before the constraint system will consider constraint object 3-E. This is because the execution of the constraint function of 2-E could lead to the invalidation of constraint object 3-E (i.e. constraint object 2-E may cause changes in the property values of the dependent objects of 2-E, which would invalidate constraint object 3-E). However, the relative order in which 4-E and 8-E are satisfied is not laid down.

Lazy constraint objects will stop the propagation of changes. This can be seen in figure 2 at constraint object 5-L. This constraint will buffer the propagation process until one of its dependent objects explicitly requests propagation. Constraint object 6-L also stops propagation. This would mean that constraint object 7-E, although it is connected to constraint object 8-E, will not be satisfied by the constraint system as a result of triggering by the left hand side object.

The constraint system tries to determine the order (as just described) in which constraints are to be solved in advance. The fact that the constraints can be solved in parallel makes it impossible to find an absolute order. Instead, the MADE constraint system determines a relative order. This relative order is always valid and does not suffer from any race-conditions. To determine the relative order the constraint starts by:

- finding all those constraint objects in the network which are eager, and for which the triggering object is an independent object.
- recursively finding all the connected eager constraint objects. A constraint object is considered to be connected if it has an independent object which is also a dependent object to a constraint object found earlier.
- determining for each constraint found so far, how often an independent object is also a dependent object of one of the other constraints that have already been found.

These steps result in a table, which, in the case of figure 2, would look like:

constraint object	count	path
1-E	0	
2-E	1	via 1-E
3-E	2	via 1-E, via 2-E
4-E	1	via 3-E
8-E	1	via 3-E

All the found constraint objects are put in an *execution-set*. Furthermore, a so-called *blocked-set* is filled with the constraint objects found after step 2. Each constraint object is put in the blocked-set as often as its corresponding count-number (in case of figure 2, this means that blocked-set will include 2-E, 3-E, 3-E, 4-E, 8-E). The execution-set and blocked-set can be determined in advance. In the MADE constraint system these sets are computed when they are needed. Then they are stored for later re-use. When the constraint network is dynamically changed, these execution-set and blocked- set may become invalid and are destroyed (to be computed again when needed).

Starting with those initial sets, the constraint system starts solving the constraint network. During solving only those constraints whose constraint objects are in the execution-set and not in the blocked-set may be satisfied. The constraints of the constraint objects that are stored in the execution-set and not in the blocked-set can be satisfied in parallel.

Once a constraint is satisfied, that constraint object is removed from the execution-set. Those constraint objects are also removed from the blocked-set if one of their independent objects is a dependent object of the constraint object that has just been removed from the execution-set. The constraint system is finished when the execution-set is empty.

A possible execution of the constraint system could be:

execution-set	blocked-set		satisfy
{ 1-E,2-E,3-E,4-E,8-E }	{ 2-E,3-E,4-E,8-E }	⇒	{ 1-E }
{ 2-E,3-E,4-E,8-E }	{ 3-E,4-E,8-E }	⇒	{ 2-E }
{ 3-E,4-E,8-E }	{ 4-E,8-E }	⇒	{ 3-E }
{ 4-E,8-E }	{ }	⇒	{ 4-E,8-E }
{ }			

If, during the satisfaction of the sub-network, a property value of an independent object (which is also a dependent object of one of the constraints objects in the execution-set) is altered, the change of this property value is immediately processed in the current satisfaction process. This approach has the advantage that the number of times (part of) a sub-network is satisfied, with very little time inbetween, is reduced as much as possible. Of course, when an object is removed from the execution-set and a property value of that object is altered after its removal, the sub-network will have to be satisfied twice in a row.

4.6. Propagating changes in Cyclic Constraint Networks

Propagation in cycles is different from propagation in a network without any cycles. Cycles can be introduced to the constraint network in two ways:

- a cycle is formed when two objects have an a-directional constraint between them.
- a cycle is formed by a sequence of (a-)directional links so that a dependent object of a former link is an independent object of the next link. Additionally, a dependent object of the last link has to be an independent object of the first link.

Cycles are detected automatically by the constraint system.

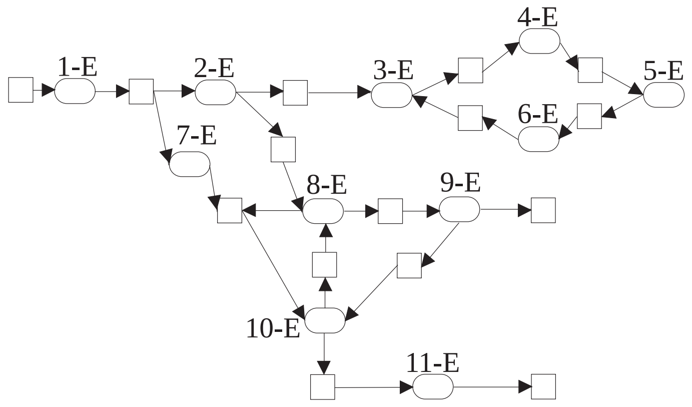

FIGURE 3. An example of a constraint network which contains cycles.

In figure 3 a structure is presented containing two cycles ([3-E, 4-E, 5-E, 6-E] and [8-E, 9-E, 10-E]). The triggering independent object is once again found on the left hand side. The satisfaction of cycles is postponed as long as possible. If a constraint object can be found which does not lie on a cycle and does not have a dependent object which is an independent object of one of the cyclic constraints, that constraint object is satisfied first. If no such a constraint object is found, the cycle is satisfied. This implies that, in the example, the constraint objects 1-E, 2-E and 7-E are satisfied first (where the relative order between 2-E and 7-E is not laid down). Next, either the cycle [3-E, 4-E, 5-E, 6-E] or [8-E, 9-E, 10-E] will be satisfied. Again, the relative order in which this is done is not laid down. When, for instance, the cycle [8-E, 9-E, 10-E] is satisfied, only these three constraint objects are considered. Propagation of changes through the constraint network is resumed to other objects only after the cycle is satisfied. In the example (figure 3) the cycle [8-E, 9-E, 10- E] is satisfied first, consequently the changes are propagated to constraint object 11-E which is then satisfied. Finally the cycle [3-E, 4-E, 5-E, 6-E] will be satisfied.

Similar to the previous section, the constraint system starts by creating an execution-set and a blocked-set using the information from the constraint network itself:

constraint object	count	path
1-E	0	
2-E	1	via 1-E
3-E	2	via 2-E, via 6-E
4-E	1	via 3-E
5-E	1	via 4-E
6-E	1	via 5-E
7-E	1	via 1-E
8-E	2	via 2-E, via 10-E
9-E	1	via 8-E
10-E	3	via 7-E, via 8-E, via 9-E
11-E	1	via 10-E

The satisfaction process, in the case of cycles, is somewhat more complicated. In addition to what was administrated in the case of directed constraints, information also needs to be stored about which cycles are reached. Accounts need to be kept of every time propagation reaches a cyclic constraint object. This constraint object is added to the *cycle-set*.

execution-set	blocked-set		satisfy	cycle-set
{ 1-E,2-E,3-E,4-E, 5-E,6-E,7-E,8-E, 9-E,10-E,11-E }	{ 2-E,3-E,3-E,4-E, 5-E,6-E,7-E,8-E, 8-E,9-E,10-E,10-E, 10-E,11-E }	⇒	{ 1-E }	{ }
{ 2-E,3-E,4-E,5-E, 6-E,7-E,8-E,9-E, 10-E,11-E }	{ 3-E,3-E,4-E,5-E, 6-E,8-E,8-E,9-E, 10-E,10-E,10-E,11-E }	⇒	{ 2-E, 7-E }	{ 3-E, 8-E, 10-E }
{ 3-E,4-E,5-E,6-E, 8-E,9-E,10-E,11-E }	{ 3-E,4-E,5-E,6-E, 8-E,9-E,10-E,10-E, 11-E }	⇒	{ }	{ 3-E, 8-E, 10-E }

At this point, no constraint can be satisfied as all constraint objects in the execution-set are also contained in the blocked-set. In this situation all paths along which propagation has occurred have either reached the end of that path, or a cycle in the network. Because the execution-set is not empty, the constraint network is not totally solved. Thus there are still cycles in the network which now need to be solved by the constraint system. The constraint system will arbitrarily choose a cycle to solve by randomly choosing a constraint object from the cycle-set.

This chosen constraint object will be considered to be the root of the cycle for the duration of the solving of the cycle. All references to this root must be removed from the blocked-set. When the constraint system solves a cycle, it may only solve those constraint objects which are in the execution-set and not in the blocked-set. An aditional requirement is that those constraint objects must lie on a cycle.

When a constraint object is solved, this object is removed from the execution-set. That same constraint object is then once more added to the blocked-set as often as it has independent objects. The aforementioned cycle-set is also maintained during the solving of a cycle. The task that every cyclic constraint object which has been solved is added to a new *current-cycle* set is new.

The next table describes the situation when constraint object 8-E is chosen from the cycle-set:

execution-set	blocked-set		satisfy	cycle-set	current cycle
{ 3-E,4-E,5-E, 6-E,8-E,9-E, 10-E,11-E }	{ 3-E,4-E,5-E, 6-E,9-E,10-E, 10-E,11-E,8-E, 8-E }	⇒	{ 8-E }	{ 3-E, 10-E, 9-E }	{ 8-E }
{ 3-E,4-E,5-E, 6-E,9-E,10-E, 11-E }	{ 3-E,4-E,5-E, 6-E,10-E,11-E, 8-E,8-E,9-E }	⇒	{ 9-E }	{ 3-E, 10-E, 9-E }	{ 8-E, 9-E }
{ 3-E,4-E,5-E, 6-E,10-E,11-E }	{ 3-E,4-E,5-E, 6-E,11-E,8-E, 8-E,9-E,10-E, 10-E }	⇒	{ 10-E }	{ 3-E, 10-E, 9-E, 8-E }	{ 8-E, 9-E, 10-E }

(continued on next page)

(continued from previous page)

execution-set	blocked-set		satisfy	cycle-set	current cycle
{ 3-E,4-E,5-E, 6-E,11-E, }	{ 3-E,4-E,5-E, 6-E,11-E,8-E, 8-E,9-E,10-E, 10-E }	⇒	{ }	{ 3-E, 10-E, 9-E, 8-E }	{ 8-E, 9-E, 10-E }

After this process has been completed the cycle has been traversed once (this can be deduced from the fact that all constraint objects in the execution-set are also contained in the blocked-set and the current-cycle is not empty). If some of the constraints in the cycle are invalidated during the solving of this cycle, the cycle has to be traversed once again. In that case, the following actions are taken:

– remove the root of the cycle from blocked-set
– remove current-cycle from cycle-set
– add current-cycle to execution-set
– make current-cycle empty
– start solving again (note that a situation is created which is similar to that at the beginning of the table shown above, so at this point, the solving of the constraint network recommences at the top of this table again)

If no updates are made, the following actions are now performed:

– remove current-cycle from cycle-set
– remove current-cycle from blocked-set
– remove those constraint objects from blocked-set for which holds that one of its independent objects is a dependent object of one of the constraint objects in current-cycle.
– make current-cycle empty

execution-set	blocked-set		satisfy	cycle-set	current cycle
{ 3-E,4-E,5-E, 6-E,11-E }	{ 3-E,4-E,5-E, 6-E }	⇒	{ }	{ 3-E }	{ }

At this point, we can solve a non-cyclic constraint (11-E):

execution-set	blocked-set		satisfy	cycle-set	current cycle
{ 3-E,4-E,5-E, 6-E,11-E }	{ 3-E,4-E,5-E, 6-E }	⇒	{ 11-E }	{ 3-E }	{ }
{ 3-E,4-E,5-E, 6-E }	{ 3-E,4-E,5-E, 6-E }	⇒	{ }	{ 3-E }	{ }

Again, the constraint system has reached a situation where it can only solve a cycle. The root of the current cycle to solve is 3-E:

execution-set	blocked-set		satisfy	cycle-set	current cycle
{ 3-E,4-E,5-E, 6-E }	{ 4-E,5-E,6-E, 3-E,3-E }	⇒	{ 3-E }	{ 4-E }	{ 3-E }
{ 4-E,5-E,6-E }	{ 5-E,6-E,3-E, 3-E,4-E }	⇒	{ 4-E }	{ 4-E, 5-E }	{ 3-E, 4-E }
{ 5-E,6-E }	{ 6-E,3-E,3-E, 4-E,5-E }	⇒	{ 5-E }	{ 4-E, 5-E, 6-E }	{ 3-E, 4-E, 5-E }
{ 6-E }	{ 3-E,3-E,4-E, 5-E,6-E }	⇒	{ 6-E }	{ 4-E, 5-E, 6-E, 3-E }	{ 3-E, 4-E, 5-E, 6-E }
{ }	{ 3-E,4-E,5-E, 6-E }	⇒	{ }	{ 4-E, 5-E, 6-E, 3-E }	{ 3-E, 4-E, 5-E, 6-E }

In the current situation, the second cycle is traversed once. Again, if a constraint object in the cycle is updated, the proper adjustements are made to the apropriate sets and execution recommences from the top of the table. If no updates have occurred, the remaining part of the network is solved:

execution-set	blocked-set		satisfy	cycle-set	current cycle
{ }	{ }	⇒	{ }	{ }	{ }

As at this point the execution-set is empty, the whole constraint network is solved.

If, for instance constraint object 6 is lazy (6-L instead of 6-E), the former cycle [3-E, 4-E, 5-E, 6-L] would no longer be a cycle as constraint object 6-L does not propagate changes to constraint object 3-E. The order in which constraint objects are satisfied will be different from the order in the example presented above. First the constraint objects 1-E, 2-E, 3-E, 4-E, 5-E, 7-E, are satisfied (not necessarily in this order). Next the cycle [8-E, 9-E, 10-E] is satisfied. Finally, after the only cycle in the network is solved, 11-E will be satisfied.

4.7. Solving the Constraint Network

In the previous section, the propagation of the changes through the constraint network was discussed. However, propagation alone will not solve the constraint network. During the process of propagation, constraint objects 'solve' the constraint relation, i.e. when propagation reaches a constraint object, it may (if necessary) execute its constraint function and see to it that the constraint relation holds again. In MADE , special constructs are introduced which assist a programmer when writing constraint functions. There is one group of three macros which are used for managing cycles in the network and there is another group of macros which can be used to determine the order in which independent objects triggered the constraint object.

This section starts with the first group of macros which define tools to manage cycles in a constraint network: *Cycle*, *CycleDo* and *CycleBreak*. These macros may only be used in the constraint function and they are used as follows:

```
Cycle (
      CycleDo ( N1 )  Statements1 ;
      CycleDo ( N2 )  Statements2 ;
      ...
      CycleBreak;
);
```

The macro *Cycle* in a constraint function provides a context in which the macros CycleDo and CycleBreak can be used. The *CycleBreak* macro allows a programmer to instruct the constraint system to stop executing the constraint function of a particular constraint object during the remaining iterations of the cycle. As a cycle is considered solved when updates no longer occur during the traversal of a cycle, the programmer can break the cycle by issuing a CycleBreak command to the constraint functions of all the constraint objects in the cycle.

The *CycleDo* command allows the programmer to specify different actions which should be executed in different iterations of the cycle. For this purpose an iteration counter is associated with each CycleDo macro. The statements of a particular CycleDo command are executed only if the cycle has iterated at least the number of times as the iteration counter of the previous CycleDo macro indicates (or 1 if no such macro exist). However, the number must not exceed that of the iteration counter of itself. Thus $\mathit{Statements}_1$ are executed in iteration 1 to N_1 and $\mathit{Statements}_2$ are executed in iteration N_1+1 to N_2.

The other group of macros (*isDefined*, *undefine*, *lastUpdated*, *cmpUpdate* and *update*) is used to determine and manage the order in which independent objects have triggered the constraint object. When a constraint object has more than one independent object it can be very useful to know which independent object triggered the constraint object, as the constraint function may have to perform different actions in different cases. Using the macros *lastUpdated (shadowFunction)* and *cmpUpdate (*$\mathit{shadowFunction}_1$*,* $\mathit{shadowFunction}_2$*)*, the constraint function can check the relative order in which shadow functions were triggered. The macro *update (shadowFunction)* allows the constraint function to re-order the ordering. It will shuffle *shadowFunction* in such a way that it is marked as the last triggered shadow function. The macro *isDefined (shadowFunction)* allows the constraint function to test whether a particular shadow function is triggered at all. The last macro, *undefine (shadowFunction)*, can be used to reset the information about the triggering history of a shadow function.

4.8. Anti-triggering the Constraint Object

The anti-triggering mechanism is very similar to the triggering mechanism. Whenever an anti-triggering member function is invoked, not the function itself but the dependent shadow function will be executed. The anti-triggering member function is delegated to the dependent shadow function. If the constraint object is invalidated the shadow function will first execute the constraint function. Then it will perform a call-back to the original anti-triggering member function.

Note that, in case of an eager constraint object, the constraint object will never be invalidated when a dependent shadow function is executed. Note also, that, when the constraint object is a lazy constraint object and no propagation is buffered, the constraint object will not be invalidated. In this case no special actions will be performed by the constraint system. The constraint system will satisfy the constraint only if the constraint object is a lazy constraint object and if propagation is buffered in the constraint object.

4.9. Synchronization in the Constraint Network

In the MADE constraint system it is very easy to synchronize the constraint actions within the constraint network itself. This synchronization results from the propagation-scheme as defined in the MADE constraint system. Thus, the MADE constraint system cannot only be used to implement constraints, it can also be used to synchronize certain actions in the application.

By *synchronizing* constraint objects in the constraint network is meant that the execution of the constraint function of two different constraint objects can be made sequential (i.e. one actions occurs before another action in time).

In general, it is not possible to say in which order two constraint objects are going to be satisfied. Figure 4 shows an example. Constraint object A is always satisfied before constraint object B and C. However, the relative order between constraint object B and C is not known (they can be solved in parallel, C before B or B before C). If it is desired that, for instance, constraint object B should execute its constraint function before constraint object C, the programmer can define a directed dependency relation between constraint object B and object X. The constraint system must then be notified that object X has to be a dependent object to constraint object B. As object X is an independent object to constraint object C, constraint object B will always be satisfied before constraint object C.

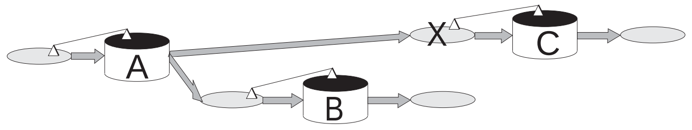

FIGURE 4a. The evaluation order of constraint objects B and C is unknown.

Although both independent and dependent relations can be used for synchronization, there is one important reason why a dependent relation is used (as opposed to an independent relation). When a dependent relation is defined, no anti-triggering member function or dependent shadow function have to be specified (and thus no extra overhead is involved). If an independent relation would be specified, at least a triggering member function and the constraint object to which the synchronization link was made should be specified.

Furthermore this constraint object should have a special independent shadow function (see figure 4b and 4c).

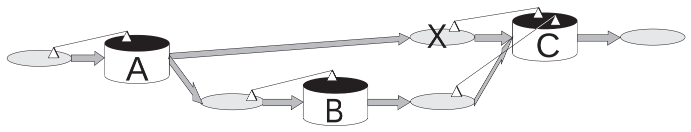

FIGURE 4b. Constraint object B executes its constraint function before constraint object C. There is an overhead in terms of triggering member functions and independent shadow functions.

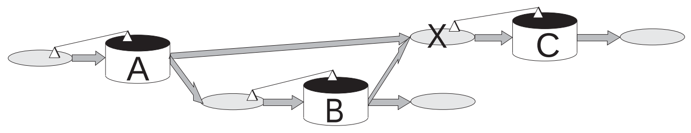

FIGURE 4c. Constraint object B executes its constraint function before constraint object C. There is no overhead.

5. CONCLUSIONS

The MADE constraint system uses a constraint network which is very well suited for multimedia applications. It provides parallel satisfaction of constraints, supports cycles in the constraint network, is capable of handling directional as well as a-directional dependency relations and can adapt the constraint network dynamically. Another important feature is, that objects that are to be constrained do not have to be prepared for constraint satisfaction. Arbitrary objects can be subject to constraint management by using of delegation.

The flexibility of the MADE constraint system is partially due to the fact that the system is able to combine the three basic tasks of a constraint system (*triggering*, *propagation* and *satisfaction*) without losing the ability to address each of these tasks separately. As the programmer has the ability to adapt each of these three tasks to its own needs, it is possible to write whatever constraint is desired. Because the MADE constraint system allows this to be done at a high level of abstraction, the programmer does not need to know all the minute details of the constraint system. Only the basic concepts of propagation, triggering and solving have to be understood.

In other constraint systems, most of the underlying mechanisms of the constraint system are hidden from the programmer. This means that the programmer only has to know how to define constraints and does not have to know how propagation, triggering and solving are realized in that system. When the constraint system is implemented as some kind of black box, it is very hard to tune the constraint system to the specific needs of the programmer (if that is possible at all). Therefore it is questionable if such a constraint system will provide the programmer with the right equipment. In the latter case, it can very well be the programmer who is constrained by the constraint system.

ACKNOWLEDGEMENTS
This research was carried out in the framework of the ESPRIT III project 6307 (MADE). Thanks goes to the many co-workers in this project for their valuable comments on the implementation.

REFERENCES

1. ARBAB F., HAGEN P.J.W. TEN, HAINDL M., HEEMAN F.C., HERMAN I., REYNOLDS G.J., SIEBES A. (1993). *Specification of the* MADE *Object Model*, ESPRIT Project 6307 (MADE) deliverable T/OM, CWI.
2. ARBAB F., HERMAN I., REYNOLDS G.J. (1993). An Object Model for Multimedia Programming, *Computer Graphics Forum*, **12** (3), pp. 101–113, The Eurographic Association.
3. BORDEGONI M. (1992). *Multimedia in Views*, report CS-9263, CWI.
4. BORNING A. (1981). The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory, in *ACM Transactions on Programming Languages and Systems*, **3**, pp. 353–387, ACM.
5. BORNING A., DUISBERG R. (1986). Constraint-Based Tools for Building User Interfaces, in *ACM Transaction on Graphics*, **5**, pp. 345–374, ACM.
6. BORNING A., DUISBERG R., FREEMAN-BENSON B., KRAMER A., WOLF M. (1987). Constraint Hierarchies, *Proceedings of OOPSLA'87*, pp 48-60, ACM.
7. BULTERMAN D.C.A. (1994). Managing the Adaptive Processing of Distributed Multimedia Information, in *CWI Quarterly*, **7** (1), pp. 3–25, CWI.
8. COURNARIE E., BEAUDOUIN-LAFON M. (1991). Alien: A Prototype-Based Constraint System, in *Second Eurographics Workshop on Object Oriented Graphics*, pp 93–114, Springer Verlag.
9. DAVIS E. (1987). Constraint Propagation with Interval Labels, in *Artificial Intelligence*, **32**, pp 281-331, Elsevier Science Publishers BV.
10. FREEMAN-BENSON B.N. (1990). Kaleidoscope: Mixing Objects, Constraints and Imperative Programming, *Proceedings of ECOOP- OOPSLA'90*, pp. 77–88, ACM.
11. FREEMAN-BENSON B.N., MALONEY J., BORNING A. (1990). An Incremental Constraint Solver, in *Communications of the ACM*, **33**, pp. 54–63, ACM.
12. HARDMAN L., BULTERMAN D.C.A., ROSSUM G. VAN (1992). *The Amsterdam Hypermedia Model, extending hypertext to support real multimedia*, CWI.
13. HEEMAN F.C., HERMAN I., REYNOLDS G.J., RUITER M.M. DE (1993). *Implementation Specification of the* MADE M*C++ language*, ESPRIT project 6307 (MADE) deliverable T/OM-S.1, CWI.
14. HENTENRYCK P. VAN (1991). Constraint Logic Programming, in *The Knowledge Engineering Review*, **6** (3), pp. 151–194.
15. HENTENRYCK P. VAN, SIMONIS H. (1991). DINCBAS M., *Constraint Satisfaction using Constraint Logic Programming*, tech.rep. CS91-62, Brown

University.

16. HERMAN I., REYNOLDS G.J., DAVY J. (1994). MADE : A Multimedia Application Development Environment, in *proceedings of the IEEE International Conference on Multimedia Computing and Systems*, IEEE CS Press.
17. HINTUM J.E.A. VAN, REYNOLDS G.J. (1993). *Constraints Objects - initial specification*, ESPRIT project 6307 (MADE) deliverabl T/COO/S.0, CWI.
18. HINTUM J.E.A. VAN (1994). *Implementation of the Constraint Objects*, ESPRIT project 6307 (MADE) deliverable T/COO/P.1, CWI.
19. HINTUM J.E.A. VAN (1994). *System Implementation of the Constraint Objects*, ESPRIT project 6307 (MADE) deliverable T/COO/P.2, CWI.
20. HINTUM J.E.A. VAN, REYNOLDS G.J. (1995). A Multimedia Constraint System (or: do we have it MADE), in proceedings EuroGraphics'95, *Computer Graphics Forum*, pp. 135–148, The Eurographic Association.
21. LELER WM. (1988). *Constraint Programming Languages, their specification and generation*, Addison Wesley Publishing Company, Reading Massachusetts.
22. MACKWORTH A.K. (1977). Consistency in Networks of Relations, in *Artificial Intelligence*, **8**, pp. 99–118, North-Holland Publishing Company.
23. MALONEY J.M., BORNING A., FREEMAN-BENSON B.N. (1989). Constraint Technology for User-Interface Construction in ThingLab II, in *proceedings of OOPSLA '89*, pp. 381-388, ACM.
24. NELSON G. (1985). Juno, a Constraint Based Graphics System, in *SIGGRAPH*, **19**, pp. 235–243, ACM.
25. RANKIN J.R. (1991). A Graphics Object Oriented Constraint Solver, in *Second Eurographics Workshop on Object Oriented Graphics*, pp. 69–91, Springer Verlag.
26. VELTKAMP R.C., ARBAB F. (1992). Geometric Constraint Satisfaction with Quantum Labels, in *Computer Graphics and Mathematics*, pp. 211–228, Springer Verlag.
27. VANDER ZANDEN B.T. (1989). Constraint Programming - A New Model for Specifying Graphical Applications, in *proceedings of CHI'89*, pp. 325–330, ACM.