A Multimedia Constraint System
(or: do we have it MADE)

J.E.A. van Hintum and G.J. Reynolds
CWI,
P.O. Box 94079, 1090 GB Amsterdam, the Netherlands,
E-mail: {hansh, reynolds}@cwi.nl

Abstract

The MADE constraint system provides excellent opportunities to introduce constraints in a multimedia application. Multimedia applications are not only a good place to experiment with constraint systems; constraints in a multimedia environment are almost indispensable. Due to the overwhelming amount of data and the number of relations between several parts of this data, multimedia applications almost demand the support of a constraint management system.
The MADE constraint system combines the object oriented programming paradigm, inherited from the mC++ language, the declarative constraint programming paradigm and the special requirements imposed upon the constraint system by the multimedia environment. Among other things, the MADE constraint system provides parallel satisfaction techniques; several constraints may be solved simultaneously and this satisfaction process is performed in parallel with the application.This not only reduces the time needed to solve the constraints, it also allows the multimedia application to proceed with its presentation while (beneath the surface) the constraints are maintained. This not only holds for the parts of the presentation that are not constrained at all, but also for those parts that are. Furthermore, the constraint system is transparent to the multimedia application; no special coding or preparation of the objects in the application is necessary. Constraints can be added later to the application without much work. Besides that, it is also possible to add and remove constraints at runtime; objects may be constrained for only a period of the time the application is running.

CR Subject Classification (1991): D.1.5, H.5.1.
Keywords and Phrases: Constraints, Constraint Programming, Object Oriented Programming, Multimedia Programming.
Note: This research has been carried out in the framework of the ESPRIT III project 6307 (**M**ultimedia **A**pplication **D**evelopment **E**nvironment). It will be presented at the Eurographics ’95 conference.

1 Introduction

This paper discusses the constraint system developed for the MADE programming environment. This discussion will deal with terminology as used in MADE, the several different flavours of constraints in MADE, how to create them, the way in which the constraint system maintains the different constraints and some results regarding the effectiveness of the constraint system (section 2.1 - section 2.3). We begin with a review of constraint systems and the requirements for constraint systems in multimedia applications.

1.1 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* [Borning et al. 86], [Maloney et al. 89] (with check-buttons, radio-buttons, bars, boxes, etc.), *geometric layout* [Nelson 85], [Rankin 91], [Veltkamp et al. 92] (with circles, rectangles, lines, points, etc.), *animation* [Borning et al. 86] (with timetables, sprites, still images, palettes, etc.) and *media synchronization* [Bordegoni 92], [Hardman et al. 92] (with timetables, media objects, error functions, etc.). However, the precise nature of these ‘things’ is not of particular interest to us; 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 one way or another by another object. A dependency relation is assumed to exist between the values of at least two properties. A *constraint* is a dependency relation which is maintained automatically by the constraint system. *Constraint objects* are imperative reflections in a programming language of the abstract notion constraint. Constraints are maintained by the *constraint system* by means of constraint objects. 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 other constraints in which these objects are involved as well.

Nowadays, several constraint systems exist. These systems differ mostly in the type of constraint network they build and the methods they use to satisfy the constraints (*satisfaction methods*). The prevailing satisfaction methods are *propagation of known states*, *propagation of degrees of freedom*, *(numerical) relaxation*, *redundant views*, *prototyping*, and *graph/term rewriting*. For an elaborated discussion on these methods the reader is referred to the literature [Mackworth 77], [Borning 81], [Davis 87], [Leler 88] and [Freeman-Benson 90], [Cournarie et al. 91]. Also the way in which the constraint systems have organized the constraint network may differ. Important issues here are the direction of the links, the presence of cycles, the priority of links, the cardinality of the links, the possibility for constraints to be added to or deleted from the constraint network dynamically and the difference in which the network is solved: incremental or at once. More information about these aspects can be found in [Borning et al. 87], [Freeman-Benson 87], [Freeman-Benson et al. 90] and [Vander Zanden 89].

1.2 Constraints in Multimedia

Although several different constraint systems exist, not all of them are suited to be used in multimedia environments in general and with an object-oriented environment such as MADE in particular. 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; the multimedia information defines the when, where and how and the multimedia data defines the what. Constraints work on the multimedia information to help the programmer with the definition, manipulation and presentation of the multimedia data ([Bulterman 94]).

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 in *parallel* with the ongoing presentation and disturb this presentation as little as possible to assure a presentation of the multimedia data that is as smooth as possible. 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 consequence of the interactive nature of multimedia applications is that the constraint system must be *dynamic*; i.e. constraints can 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 a 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.

When these requirements are taken into consideration for constraint systems in multimedia environments, we conclude that several satisfaction methods cannot be used for our purposes. Propagation of degrees of freedom is unsuitable for use in a parallel multimedia environment; these methods require that the objects in the constraint network are invariant during the process of constraint satisfaction. This implies that the presentation has to be stopped while a constraint is satisfied. Prototyping is another method that is not suited to constraint satisfaction in multimedia applications; prototyping can only be used for simple constraint problems and not for the complex dependency relations which are common in multimedia presentations. Relaxation and graph/term rewriting are in general too time consuming to be realistic alternatives. This leaves only propagation of known states and redundant views.

It is not only the particular satisfaction methods that may be well or ill suited for multimedia constraint satisfaction, but also the different constraint networks may have their pro's and cons with respect to multimedia constraint satisfaction. In the ideal situation, a constraint system should support both directional links (property values depend on other property values but not the other way around) and a-directional links (property values are interdependent; the values of the properties depend on each other), be able to solve cycles in the constraint network, be able to distinguish more important constraints from less important constraint on the basis of their priority and solve the constraints with the highest priority first, allow for constraints which get their input from a set of objects and propagate this information to another set of objects (this in contrast with the situation where the information of only one object is propagated via the constraint object to one other object) and have the ability to dynamically add or delete constraints to or from the constraint network.

Constraints can be of great practical use in multimedia applications, but only if they are well designed for 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 also improves readability and maintenance of the application, which is a very important aspect in object oriented languages.

2 The Theory of Constraint Satisfaction in MADE

The constraint system in MADE is an a-priority, dynamic, incremental system, capable of solving m-m, a-directional, directional and cyclic relations, that 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. To better understand the context in which the MADE constraint system is written, the reader is referred to the literature ([Arbab et al. 93a], [Arbab et al. 93b], [Heeman et al. 93], [Herman et al. 94]). The following sections define some terminology and then discusses how a programmer should use the system and how the constraints, defined by the programmer, are maintained by the constraint system.

2.1 Concepts in the MADE Constraint System

The constraint system is embedded in the MADE environment, and hence can make use of the features mentioned in the previous section. However, before we can show how these features are applied, we need to first discuss the main concepts of the constraint system.

Constrainable objects in MADE can be added to a 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 resatisfied 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 *dependent objects*. The names, dependent and independent, are relative to a specific constraint object; 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 the dependent objects request 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). A situation 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. Because MADE provides an object oriented environment, the objects in mC++ have to obey the OO paradigm; changes to property values have to be made through the object's interface. Thus every time a constrained property value is changed, a member function of an independent object is invoked. This fact is used in the mechanism to trigger the constraint object. The member function of the independent object which changes the property value due to which a constraint object is triggered 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 the constraint object to update its property values if necessary.

The actual maintenance of the constraints, adapting the property values of the dependent objects when the property values of the independent objects have been altered, is implemented 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. One of the actions of the shadow function will be to execute the original (anti-)triggering member function. The purpose of these shadow functions is to enable the constraint system to keep its administration of the constraints in the constraint network up to date. There are two types of shadow functions: *independent shadow functions* (shadow functions for the triggering member function) and *dependent shadow functions* (shadow functions for the anti-triggering member function). The shadow function will activate a *constraint function*. In the constraint function, the programmer specifies the actions which must be performed to maintain the dependency relations. Each constraint object must have at least one constraint function associated with it.

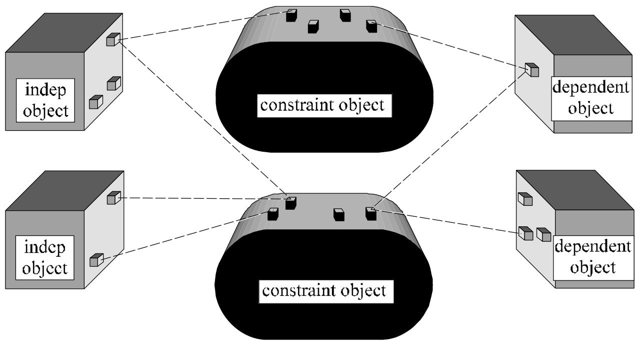

Figure 1: Graphical View of Independent Objects, Constraint Objects and Dependent Objects.

In figure 1, a schematic presentation is given of the relations between independent objects, constraint objects and dependent objects. In this figure, dependent and independent objects are represented by boxes. The little bulges on these boxes represent the member functions of the objects. Some of these bulges are connected with bulges on the cylindric shapes. These cylindric shapes represent the constraint objects and the bulges the shadow functions. The bulges on the boxes which are connected to the shadow functions are the (anti-)triggering member functions.

2.2 Characterization of the MADE Constraint System

The need for a new constraint system (as opposed to an existing one) was born out of the desire to integrate the utilities and the functionality of the MADE environment with the constraint system. This not only meant that the different media objects like video, audio, graphics, etc. had to be constrainable, it also meant that the constraint objects could be treated like any other object in the MADE environment; it had to be possible to 1) store constraint objects in the database and thus make them persistent, 2) make snapshots of the status of the constraint object, 3) monitor the constraint object and 4) use a constraint from within a scripting language.

So, beside the requirements for a multimedia constraint system, the MADE constraint system also had to consider the above requirements. This section discusses the various choices made for the MADE constraint system.

The MADE constraint system uses the satisfaction method termed propagation of known states. The constraints themselves are solved in parallel as much as possible too. To do so, the constraint system makes use of active objects; one for every constraint that is in the process of being satisfied (i.e. an active object is created just before and destroyed just after the constraint is satisfied; thus, these active objects are dynamically created).

The constraint system supports directional and also a-directional dependency relations. The approach taken is a hybrid one. 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 realised by special constructs which allow a constraint object to check which independent object last triggered the constraint object. Depending on that result, the appropriate actions can be taken.

It is also possible to define cyclic constraint networks. Cyclic constraints 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. The programmer also has 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 also supports m-m dependency relations.

Prototyping is already provided by the MADE programming environment. Therefore, the constraint system does not have to provide special support for this feature. Also, redundant views are not supported. However, these can be programmed by the programmer using the constraint system without much extra effort.

The objective of the MADE constraint system is to provide constraints which do not require any adaptations to the objects which are 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. Delegation allows for the dynamic change of the behaviour of a member function without the requirement that the code of the original member function has to be adapted or prepared for this[1].

2.3 Writing Constraint Classes

In MADE, constraint objects are implemented using classes. Once a constraint class has been defined, instances of constraint objects can be made with just a simple declaration. The main advantage of constraint classes over constraint instances is that classes can be organized in libraries. Using the class-oriented approach, the programmer does not necessarily have to have detailed knowledge of constraint programming; if the desired constraint classes are available, the programmer only has to declare constraint instances. On the other hand, if the programmer has enough knowledge of constraint programming, it is possible to construct specialized constraints tailored for a particular situation.

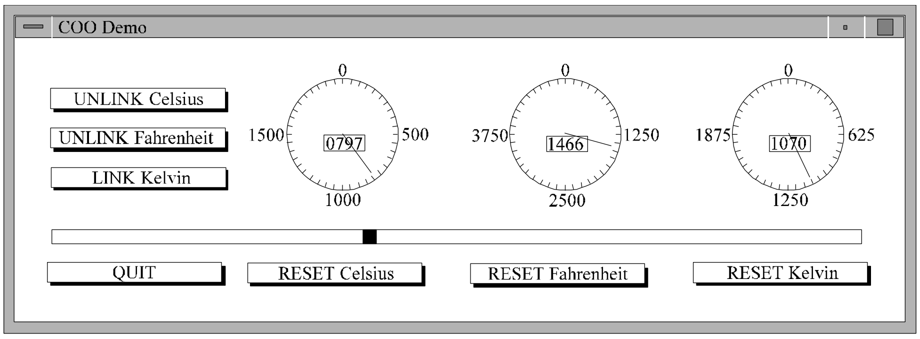

Figure 2: The output of the demo conversion program.

In the remainder of this paper, we will explain the use of the MADE constraint system and constraint objects. As a running example, we will use a simple temperature conversion program which converts Celsius degrees into Fahrenheit and Kelvin degrees and vice versa. This example (see fig 2) shows a type of constraint which will often be used in multimedia applications: a number of different views on the same piece of information, which have to be kept consistent. Conversion can be switched on and off using the link/unlink buttons. Figure 3 shows the constraint network created for this example. The reason for taking this example stems from the observation that the chosen conversion is simple and small and can demonstrate the major features of the constraint system.

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

In this example, it is assumed that a few classes have already been defined (see below). Only their interfaces are shown.

Three unprotected classes are defined (the `unprotected` keyword is to ensure the use of delegation). `Degree` is a class whose sole purpose is to hold the temperature value. The class `Clock` is used to display the temperatures in an attractive way on the screen. The class `Slider` is an input device class, which allows the user to interact with the application. Note that no special preparations are made in the class definitions presented below. This implies that, without any special coding, arbitrary objects may be constrained. This approach is possible due to the presence of delegation.

```
Unprotected Degree {
  public:
    Degree ();
    ~Degree ();
    void setTemp (int i);
    int getTemp ();

  private:
    int _ temperature;



};
```

```
Unprotected Clock {
  public:
    Clock ();
    ~Clock ();
    void setClock (int i);
    int getClock ();
    void drawClock ();
  private:
    int _clockValue;
    int _xCoordinate;
    int _yCoordinate;
    int _zCoordinate;
};
```

```
Unprotected Slider {
  public:
    Slider ();
    ~Slider ();
    void setSlider (int i);
    void setSliderFromScreen ();
    void drawSlider ();
  private:
    int _xCoordinate;
    int _yCoordinate;
    int _zCoordinate;

};
```

2.3.1 **Eager Directed Constraints**

A feature which is very common in multimedia presentations is the output of some data on the screen. In the example, changes to the temperature values maintained in the class `Degree` are to be displayed on the screen. The constraint class `DegreeToClock` is introduced for this purpose. This constraint class shows the general setup needed for a constraint object.

The code below shows the declaration of an eager, directed constraint class. All constraint classes are `Mutex`, i.e. the execution of the member functions of a particular instance of the class is mutually exclusive (but not the execution of two member functions of two different instances). Furthermore, constraint classes inherit from the superclass `CO`. This superclass provides some basic functionality required by the constraint system.

```
Mutex DegreeToClock : public CO {                                  // constraint class declaration
  public:
    DegreeToClock (Degree* d, Clock* c);
    ~DegreeToClock ();
    void constraintFunction ();                                    // constraint function declaration
    declare_indep_shadow_func (void, setTemp, (int));              // shadow function declaration
  private:
    Degree* _d; Clock* _c;
};

DegreeToClock::DegreeToClock (Degree* d, Clock* c): COEager {      // constructor + initiator
  _d = d; _c = c;
};
DegreeToClock::~DegreeToClock () {};                               // destructor

void DegreeToClock::constraintFunction () {                        // constraint function definition
  int tmp = _d->getTemp (); _c->setClock (tmp);
};

indep_shadow_func_body (void, DegreeToClock, setTemp, (int i), (i));  // shadow function definition
```

The constructor and destructor are provided as usual; only the constructor has to have an initiator. This initiator may be either `COEager` or `COLazy` depending on whether an eager constraint or a lazy constraint is defined (in the example an eager constraint is used since changes to the temperature need to be displayed immediately on the screen).

The programmer also has to provide the constraint function. The only requirement for this function is that the function is declared public and that the signature conforms to `void (CO::*) ()`. There are no restrictions on the name or the body of the constraint function.

Finally, the shadow functions of the constraint object have to be declared and defined. A set of macros are provided to facilitate their use; a pair of macros for independent shadow functions, and a second pair for dependent shadow functions.

° declare_indep_shadow_func (*resultType*, *methodName*, *parameterTypes*)
° indep_shadow_func_body (*resultType*, *constraintObject*, *methodName*, *parameters*, *parameterNames*)
° declare_dep_shadow_func (*resultType*, *methodName*, *parameterTypes*)
° dep_shadow_func_body (*resultType*, *constraintObject*, *methodName*, *parameters*, *parameterNames*)

resultType	the result-type of the (anti-)triggering member function.
methodName	the name of the (anti-)triggering member function.
parameterTypes	a comma-separated parameter list (types only) of the parameters of the (anti-)triggering member function.
constraintObject	the class name of the constraint object the shadow function belongs to.
parameters	a comma-separated parameter list (types and names) of the parameters of the (anti-)triggering member function.
parameterNames	a comma-separated parameter list (names only) of the parameters of the (anti-)triggering member function.

The careful reader may have observed that the names of the shadow functions are fixed for a particular constraint class. However, it is possible to connect an arbitrary (anti-)triggering member function to the name defined in the constraint class. The next section, shows an example of this with some special functions which can be used to handle a-directional constraints.

2.3.2 A-directional Constraints (defining cycles)

The next step in the example program is the real conversion constraint. This constraint class is called `CFK` (**C**elsius, **F**ahrenheit, **K**elvin).

In the code below, several features of the MADE constraint system are used. One of these features, `COEager`, has already been mentioned in the previous section. In this constraint class, shadow function names are used which have no correspondence to an existing triggering member function. In the example, all three `Degree` objects have the member function `setTemp` as triggering member function. However, this would lead to a situation where the constraint function is not able to determine which of the three Degree objects (Celsius, Fahrenheit or Kelvin) has changed. By assigning a different name to each of the shadow functions (`setTempC`, `setTempF` and `setTempK`) and connecting these to the triggering member functions *setTemp*, the constraint function can detect which independent object triggered the constraint object. How this connection is made is shown in a following section.

Another point of interest in the code is the special functions used in the constraint function. The MADE constraint system provides some basic functionality which supports the programmer in maintaining cycles in the constraint network. These functions take as arguments the names of the independent shadow functions (note that these functions are not defined for the dependent shadow functions):

° MBool **is_defined** (*independentShadowFunction*)
 a shadow function is said to be '*defined*' when the corresponding triggering function has triggered the constraint object and has not been '*undefined*' since. `MBool` is a predefined type in mC++: `MBool = {MFALSE, MTRUE}`
° void **undefine** (*independentShadowFunction*)
 undefines the shadow function *independentShadowFunction*.
° MBool **last_updated** (*independentShadowFunction*)
 allows the programmer to check whether the independent triggering function which triggered the

constraint object is the one which is delegated to *independentShadowFunction*.

- int **cmp_update** (*independentShadowFunction1*, *independentShadowFuntion2*)

  returns information about the history in which the constraint object was triggered by the various independent triggering member functions. If the independent triggering function delegated to *independentShadowFunction1* has triggered the constraint object more recently than the independent triggering member function delegated to *independentShadowFunction2*, or when *independent-ShadowFunction1* is defined and *independentShadowFunction2* is not, the result equals 1. If both, *independentShadowFunction1* and *independentShadowFunction2*, are undefined, the result is 0. Otherwise, the result equals -1.
- void **update** (*independentShadowFunction*)

  this function reorders the information of the triggering history; it makes the constraint system treat the triggering member function which is delegated to *independentShadowFunction* as the triggering member function which has triggered the constraint object last. Thus `update (F); last_updated (F);` always returns `MTRUE` and `update(F); last_updated(G);` always returns `MFALSE`.

```
Mutex CFK : public CO {
  public:
    CFK (Degree* d1, Degree* d2, Degree* d3);
    ~CFK ();
    void constraintFunction ();
    declare_indep_shadow_func (void, setTempC, (int));
    declare_indep_shadow_func (void, setTempF, (int));
    declare_indep_shadow_func (void, setTempK, (int));
  private:
    Degree *_d1, *_d2, *_d3;
};
CFK::CFK (Degree *d1, Degree *d2, Degree* d3) : COEager { _d1 = d1; _d2 = d2; _d3 =d3; };
CFK::~CFK () {};

void CFK::constraintFunction () {
  Cycle (
    CycleDo (1) {
      if (last_updated (setTempC)) {
        int tmp = _d1->getTemp (); _d3->setTemp (tmp + 273); _d2->setTemp (((9 * tmp) / 5) + 32);
      }
      else if (last_updated (setTempK)) {
        int tmp = _d3->getTemp (); _d1->setTemp (tmp - 273); _d2->setTemp (((9 * (tmp - 273)) / 5) + 32);
      }
      else if (last_updated (setTempF)) {
        int tmp = _d2->getTemp (); _d1->setTemp (((tmp - 32) * 5) / 9); _d3->setTemp (((tmp - 32) * 5) / 9 + 273);
      }
    };
    CycleBreak;
 );
};

indep_shadow_func_body (void, CFK, setTempC, (int i), (i));
indep_shadow_func_body (void, CFK, setTempF, (int i), (i));
indep_shadow_func_body (void, CFK, setTempK, (int i), (i));
```

Another aspect in the constraint function which supports the maintenance of cycles in the constraint network are the macros `Cycle`, `CycleDo` and `CycleBreak`. These constructs allow the programmer to specify which actions the constraint object has to perform in the different iterations of the cycle. This implies that the programmer also has to specify under which conditions a constraint cycle is believed to be solved and under which conditions a cycle need to be broken. The general use of these constructs is as follows:

```
Cycle (
    CycleDo (count-1) { ... } [ ; ]
    CycleDo (count-2) { ... } [ ; ]
        ...
    CycleDo (count-3) { ... } [ ; ]
    [ CycleBreak [ ; ] ]
);
```

The `CycleDo` commands have to be sorted by the value of *count-?*; in other words *count-1* < *count-2* < ... < *count-n*. The `CycleDo (count-n)` construct will execute mC++ statements in the associated body when the constraint system is iterating the cycle for at least *count-n* times, but not more than the count of the next `CycleDo` command: *count-n* ≤ iteration < *count-n+1*. The `CycleBreak` command allows the programmer to instruct the MADE constraint system to break the cycle (i.e. the cycle is considered solved and the constraint system will proceed to solve the remaining part of the constraint network).

In the example, a cycle is formed with the `CFK` constraint object and the three `Degree` objects connected to the `CFK` object; whenever a `Degree` object triggers the `CFK` constraint object, the other two `Degree` objects are updated (via the `setTemp(...)` member function). These two objects, in their turn, will trigger the constraint object `CFK` again. As the values of the `Degree` objects are already correct after the first iteration, a `CycleDo (1)` statement is specified, which is only executed for the first iteration. In the second iteration, no CycleDo statement is available that could be executed. In this case, the `CycleBreak` statement is executed, which causes the constraint system to consider the constraint `CFK` satisfied and the `Degree` objects to have their correct values.

An example of multiple links has also been used in the `CFK` constraint object. In the constraint function of that constraint object, a change of one `Degree` object will be propagated to the two other `Degree` objects. The other possible way to have multiple links (i.e. two different triggering member functions are connected to the same independent shadow function) will be shown in the next section.

Most features of the MADE constraint system have now been shown. The definition of the constraint class `SliderToDegree` is omitted as it does not contain any new features.

2.3.3 Instantiating the Constraint Objects

Once the constraint classes are defined, the programmer may create the actual constraint objects. The code to declare the necessary constraint objects for the example program consists of the statements shown below.

- This code is, beside the class definition of the constraint objects, the only code which has to be written to be able to use the constraints. Thus, instead of adapting old, existing code, we only have to add some new code. This implies that old code can also be subject to constraints without the need to change it. The only requirement is that the member function of the objects which need to be constrained can be delegated. In practice this means that these objects have to be MADE classes (and not just an ordinary C++ class).
- Independent objects are registered with the function `registerIndependentObject`. This function may take three or four arguments. The fourth argument is used to specify a nonstandard shadow function name. In case of constraint object `Cnstr2`, the member functions `setTemp` are connected to the shadow functions `setTempC`, `setTempF` and `setTempK`. Thus the fourth argument enables the programmer to define arbitrary names for the shadow functions in the constraint classes. If the name of the shadow function is equal to the name of the independent triggering member function, the fourth argument may be omitted.
- Dependent objects are registered using the function `registerDependentObject`. This function may take two, three or four arguments. When only two arguments are supplied, no anti-triggering member function exists. This can be used for eager constraints (like in our example), where propagation is done immediately and dependent objects do not have to request for an update via an anti-triggering member function. The third argument is, just as with `registerIndependentObject`, the name of the anti-triggering member function and the fourth argument the nonstandard shadow function name.
- The function `registerConstraintObject` is used to specify which constraint function has to be used by the constraint system when a particular constraint object is to be satisfied. A constraint object may choose (if desired) from different constraint functions, although only one constraint function is used at a time. Another constraint function is selected by another invocation of `registerConstraintObject`.

- The member function `void celsius::setTemp(int)` is delegated to both `Cnstr2` and `Cnstr3`. This is an example where one triggering member function is connected to two different shadow functions.

```
startConstraintPackage ();                                                        // setup constraint system

SliderToDegree* Cnstr1 = new SliderToDegree (mySlider, celsius);                  // make a SliderToDegree constraint object
registerDependentObject (celsius, Cnstr1);                                        // celsius is a dependent object
registerIndependentObject (mySlider, Cnstr1, "void setSlider(int)");              // void mySlider::setSlider is a triggering member function
registerIndependentObject (mySlider, Cnstr1, "void setSliderFromScreen(int)");    // void mySlider::setSliderFromScreen is a triggering mem function
registerConstraintObject (Cnstr1, (cft) SliderToDegree::constraintFunction);      // define constraint function to use

CFK* Cnstr2 = new CFK (celsius, fahrenheit, kelvin);                              // make a CFK constraint object
registerDependentObject (celsius, Cnstr2);                                        // celsius, fahrenheit and kelvin are dependent objects
registerDependentObject (fahrenheit, Cnstr2);
registerDependentObject (kelvin, Cnstr2);
registerIndependentObject (celsius, Cnstr2, "void setTemp(int)", "void setTempC(int)");        // delegate celsius::setTemp to setTempC
registerIndependentObject (fahrenheit, Cnstr2, "void setTemp(int), "void setTempF(int)");      // delegate fahrenheit::setTemp to setTempF
registerIndependentObject (kelvin, Cnstr2, "void setTemp(int)", "void setTempK(int)");         // delegate kelvin::setTemp to setTempK
registerConstraintObject (Cnstr2, (cft) CFK::constraintFunction);                 // define constraint function to use

DegreeToOutput* Cnstr3 = new DegreeToOutput (celsius, cClock);                    // make a DegreeToOutput constraint object
registerDependentObject (cClock, Cnstr3);
registerIndependentObject (celsius, Cnstr3, "void setTemp(int)");
registerConstraintObject (Cnstr3, (cft) DegreeToOutput::constraintFunction);

DegreeToOutput* Cnstr4 = new DegreeToOutput (fahrenheit, fClock);                 // make another DegreeToOutput constraint object
registerDependentObject (fClock, Cnstr4);
registerIndependentObject (fahrenheit, Cnstr4, "void setTemp(int)");
registerConstraintObject (Cnstr4, (cft) DegreeToOutput::constraintFunction);

DegreeToOutput* Cnstr5 = new DegreeToOutput (kelvin, kClock);                     // and a third DegreeToOutput constraint object
registerDependentObject (kClock, Cnstr5);
registerIndependentObject (kelvin, Cnstr5, "void setTemp(int)");
registerConstraintObject (Cnstr5, (cft) DegreeToOutput::constraintFunction);

   ...                                                                            // constraints are active

closeConstraintPackage ();                                                        // delete the constraint system

delete Cnstr1; delete Cnstr2; delete Cnstr3; delete Cnstr4; delete Cnstr5;        // delete the constraint objects
```

The final constraint network, which is constructed for the example program is shown in figure 3. The convention which is adapted in this figure corresponds with that used in figure 1:

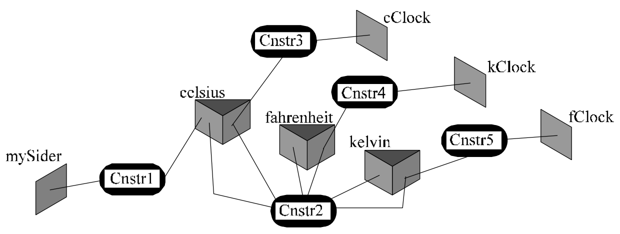

Figure 3: The Constraint Network for the Conversion Program.

3 Results and Conclusions

In this paper, the theory and ideas behind the MADE constraint system have been discussed. An important notion within this system is delegation. The whole scheme of triggering the constraint objects (and thus the constraint system) is based upon delegation. Without this mechanism, constraints could not be incorporated into ordinary applications in the transparent way as they can be done now. Due to the presence of the notions of message passing and delegation, the MADE constraint system allows for easy declaration and maintenance of new and existing constraints in (partially) already developed applications.

The MADE constraint system has been developed for use in multimedia applications. An important aspect of multimedia applications is parallelism (either true parallelism or interleaved). The MADE constraint system realizes parallel constraint satisfaction. The advantages of this approach become apparent when the measured performance of the conversion example is examined.

These performance measurements were considered in relation to an explicitly programmed solution that does not make use of the constraint satisfier, and which executes as a single active object. We observed that, besides the fact that the constrained system is considerably slower than the reference solution (something which is to be expected as the constraint system has to maintain the complete administration of constraints, (in)dependent objects and constraint networks), the standard deviation (the mean difference between the different measurements) is constant and negligible for the reference solution whereas the standard deviation for the constrained program is more noticeable. The reason for this lies mainly in the scheduler of the MADE runtime code which does not always make the same assignment for the different processes, and thus the parallel running satisfaction processes may sometimes take a little bit more time to complete a task.

However, when the same measurements were taken where the result of the conversion is shown on the screen by the three graphical clocks, the situation is different. In this case, the constrained program performs considerably better compared to the previous test; sometimes the constrained program performs even better than the reference program. This result is mainly to do with the fact that the tests were run under X-windows. In this situation the reference program had to wait for X-windows to complete the screen actions, whereas, the constrained program can still satisfy part of the constraint network. This result can also be concluded from the standard deviations: the standard deviation of the reference program increases considerably (compared with the previous test); these fluctuations are caused by interactions with the windowing system and network traffic. The constrained program is less burdened by the windowing system, as requests to this system are made by the parallely running satisfaction processes. These satisfaction processes do not have to be finished for the main program to continue. However, if the number of requests to the windowing system become large, eventually the constrained program is hindered somewhat; the satisfaction processes (and thus the constraint objects) are delayed long enough by the windowing system that the main program also has to wait.

In general, we conclude that the MADE constraint system provides a constraint system that is well suited for use in object-oriented programming environments and that is relatively efficient for multimedia purposes. The constraint system meets the requirements imposed by the MADE environment in that it is integrated with the other object features of the environment and actually exploits them in some cases. It also satisfies our other main objective of providing a powerful constraint system that at the same time has little programming impact on an applications existing code. The parallel constraint satisfaction approach has allowed a system to be developed that works well with multimedia applications and the particular constraint satisfaction method implies only nominal organizational overhead.

One problem with this system is the performance affect described above which can lead to inaccurate reactions to certain events. However, changes in multimedia presentations should not be made instantaneously anyway. Presentations should adapt in a smooth fashion in order to prevent them from jolting every time a deviation from the ideal situation is detected.

4 Acknowledgements

We would like to thank F. Arbab and R.C. Veltkamp for their valuable ideas and remarks. Special thanks goes to F.C. Heeman for he was always willing to explain the details of C++ and mC++. This research was carried out within the framework of the ESPRIT III project 6307 (MADE). We would also like to thank our many co-workers in this project for their valuable comments on the implementation.

5 References

[Arbab et al. 93a] Arbab F., Hagen P.J.W. ten, Haindl M., Heeman F.C., Herman I., Reynolds G.J., Siebes A., *Specification of the MADE Object Model*, tech.rep. T/OM, CWI, 1993.

[Arbab et al. 93b] Arbab F., Herman I., Reynolds G.J., An Object Model for Multimedia Programming, in proceedings EuroGraphics'93, *Computer Graphics Forum*, **12:3**, pp. 101-113, The Eurographic Association, 1993

[Bordegoni 92] Bordegoni M., *Multimedia in Views*, tech.rep. CS-9263, CWI, 1992.

[Borning 81] Borning A., The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory, in *ACM Transactions on Programming Languages and Systems*, **3**, pp. 353-387, ACM, 1981.

[Borning et al. 86] Borning A., Duisberg R., Constraint-Based Tools for Building User Interface, in *ACM Transaction on Graphics*, **5**, pp. 345-374, ACM, 1986.

[Borning et al. 87] Borning A., Duisberg R., Freeman-Benson B., Kramer A., Wolf M., Constraint Hierarchies, in *proceedings of OOPSLA'87*, pp 48-60, ACM, 1987.

[Bulterman 94] Bulterman D.C.A., Managing the Adaptive Processing of Distributed Multimedia Information, in *CWI Quarterly*, **7:1**, pp. 3-25, CWI, 1994.

[Cournarie et al. 91] Cournarie E., Beaudouin-Lafon M., Alien: a prototype-based constraint system, in *Second Eurographics Workshop on Object Oriented Graphics*, pp 93-114, Springer Verlag, 1991.

[Davis 87] Davis E., Constraint Propagation with Interval Labels, in *Artificial Intelligence*, **32**, pp 281-331, Elsevier Science Publishers BV, 1987.

[Freeman-Benson 90] Freeman-Benson B.N., Kaleidoscope: Mixing Objects, Constraints and Imperative Programming, in *proceedings of ECOOP-OOPSLA'90*, pp. 77-88, ACM, 1990.

[Freeman-Benson et al.90] Freeman-Benson B.N., Maloney J., Borning A., An Incremental Constraint Solver, in *Communications of the ACM*, **33**, pp. 54-63, ACM, 1990.

[Hardman et al. 92] Hardman L., Bulterman D.C.A., Rossum G. van, *The Amsterdam Hypermedia Model, extending hypertext to support real multimedia*, tech.rep. CWI, 1992.

[Heeman et al. 93] Heeman F.C., Herman I., Reynolds G.J., Ruiter M.M. de, *Implementation Specification of the MADE mC++ language*, tech.rep. T/OM-S.1, CWI, 1993.

[Herman et al. 94] Herman I., Reynolds G.J., Davy J., MADE: A Multimedia Application Development Environment, in *proceedings of the IEEE International Conference on Multimedia Computing and Systems*, IEEE CS Press, 1994.

[Hintum et al. 93] Hintum J.E.A. van, Reynolds G.J., *Constraints Objects - initial specification*, tech.rep. T/COO/S.0, CWI, 1993.

[Hintum 94a] Hintum J.E.A. van, *Implementation of the Constraint Objects*, tech.rep. T/COO/P.1, CWI, 1994.

[Hintum 94b] Hintum J.E.A. van, *System Implementation of the Constraint Objects*, tech.rep. T/COO/P.2, CWI, 1994.

[Leler 88] Leler Wm., *Constraint Programming Languages, their specification and generation*, Addison Wesley Publishing Company, Reading Massachusetts, 1988.

[Mackworth 77] Mackworth A.K., Consistency in Networks of Relations, in *Artificial Intelligence*, **8**, pp. 99-118, North-Holland Publishing Company, 1977.

[Maloney et al. 89] Maloney J.M., Borning A., Freeman-Benson B.N., Constraint Technology for User-Interface Construction in ThingLab II, in *proceedings of OOPSLA'89*, pp. 381-388, ACM, 1989.

[Nelson 85] Nelson G., Juno, a Constraint Based Graphics System, in *SIGGRAPH*, **19**, pp. 235-243, ACM, 1985.

[Rankin 91] Rankin J.R., A Graphics Object Oriented Constraint Solver, in *Second Eurographics Workshop on Object Oriented Graphics'91*, pp. 69-91, Springer Verlag, 1991.

[Veltkamp et al. 92] Veltkamp R.C., Arbab F., Geometric Constraint Satisfaction with Quantum Labels, in *Computer Graphics and Mathematics*, pp. 211-228, Springer Verlag, 1992.

[Vander Zanden 89] Vander Zanden B.T., Constraint Programming - A New Model for Specifying Graphical Applications, in *proceedings of CHI'89*, pp. 325-330, ACM, 1989.