Chapter 6

Design and Implementation

The previous chapter presented a formal specification of the theory behind the Constraint Graphs system. This chapter describes the design and implementation of one particular program that fulfills that specification. This chapter also extends the description to include a user interface, which is a separate layer within the system as per **Requirement 1** (see Figure 16 in Section 6.1). Due to space limitations and its subservient focus, the user interface design is only superficially described in Section 6.2 (and is not formally specified at all); but an earlier version of the user interface is documented elsewhere (Kremer 1993). The actual C++ implementation from the specification in chapter 5 is described in Section 6.3. The implementation is a collection of related C++ classes, each corresponding to a component of the specification. While the description of the class libraries gives the reader a very good idea of the software design and structure, it does not shed much light on how the tool is used in a practical sense. Section 6.4 describes how the software can be used to implement a visual language using a type lattice, arc termination constraints, and ad-hoc constraints, as well as how to use the ontological levels to improve the user interface. Finally, Section 6.5 is a catch-all which describes some of the interesting technical details in the implementation.

6.1 Independence of User Interface and Constraint Graph

In the overall structure of the design, there is an abstract Constraint Graph (described in chapter 5) which implements all of the requirements of Section 4.3. While this abstract graph presents a very specific software interface, it implements no user interface at all.

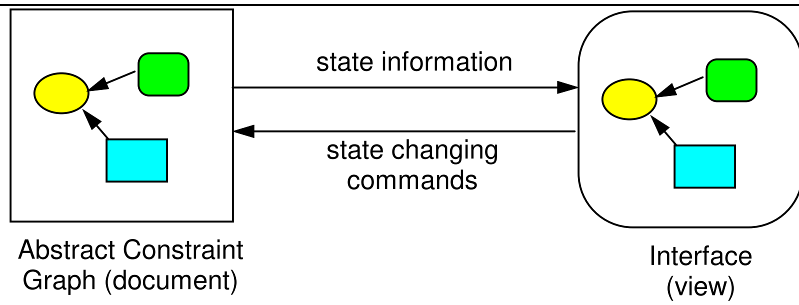


Figure 16: Separate abstract graphs and interfaces communicate state information and state-changes commands.

There is a separate and independent user interface (see Figure 16) which is solely responsible for

1. drawing a version of the abstract graph on the screen,
2. collecting input from the user,
3. translating the input into operations on the abstract graph, and
4. updating the on-screen version of the graph to reflect any state changes resulting from graph operations.

The design closely tracks the Model-View-Controller pattern described in (Buschmann et al. 1996) and used in the Smalltalk class library; however, the View and Controller are bundled into a single role (often called the Document-View variant), which is not uncommon (Gamma et al. 1995). This pattern has several advantages, for example:

- Many different interface packages may be layered on top of the abstract constraint graph.
- Several different sub-views (or "filters") of the abstract constraint graph may be implemented on a single abstract graph (see Figure 17).

The constraint graph is responsible for all information about relationships between *components* (the collective name for nodes and arcs), including the type lattices. It is also responsible for maintaining arbitrary information about all components in the form of attribute/value pairs associated with each component. Attribute/value pairs are inherited though the type lattice. The constraint graph is *not* responsible for maintaining physical information about components such as physical size and location; it is only responsible for relationships. However, a view may store some physical information (color, for example) in the constraint graph's components' attribute/value pairs list if this information is intended to be global between views and is intended to be inherited by subtypes. Size and location are attributes that should be neither global between views nor inherited by subtypes: The same component in more than one view may not necessarily occupy the same coordinates. Furthermore, inheritance of location may lead to the absurd situation where every object occupies the same default location, since every component is a subtype of T (top).

Views, or interfaces, are responsible for physical information such as location and size of node. As already mentioned, views may store some physical information in the constraint

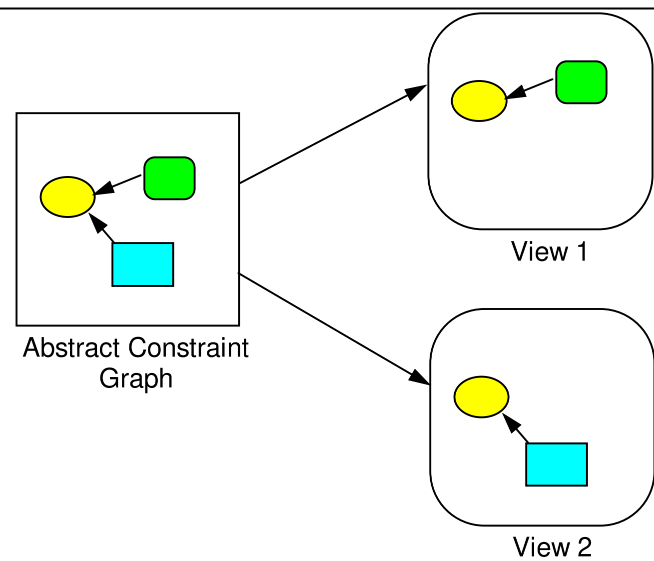


Figure 17: A single abstract graph with multiple sub-views in the interface.

graph's components' attribute/value pairs, provided the information is intended to be view-global and inherited through the type lattices.

The independence of the abstract constraint graph and interface is reflected in the storage format as described in Section 6.5.1.

6.2 The CMap Class Library

The previous section described the independence of the graphical user interface and the abstract constraint graph. This section fills in details of the structure of the independent interface layer and how this independence in achieved.

An independent graphical class library was built to support drawing concept maps on the screen. The class library is called CMap (for Concept Map), and was tested by building a simple stand-alone concept map drawing program called KSIMapper. KSIMapper is only useful as a drawing tool and does not support constraints of any kind. It does no more than provide a basic windowing and interface event environment in which the CMap class library objects can operate.

A slightly simplified version of the CMap class library is shown in Figure 18. Graphical objects are divided into visual graphics and behavioural graphics. All behavioural graphics implement interface behaviours and are never directly drawn to the screen (they are *controllers* in the model-view-controller pattern). Behavioural graphics all *delegate* drawing to an object which is a subtype of visual graphic (the directed arc labeled "refers-to" in Figure 18). Since all programmatic references are always to behavioural graphics, this scheme allows graphical objects to alter the specific drawing class without destroying the integrity of any enclosing structure. For example, a Node may use an *EllipseShape* or the *RectangleShape* (both subclasses of *Shape*) as its visual delegate, but a user can easily swap between the two because no object (other than the Node) refers directly to the delegate.

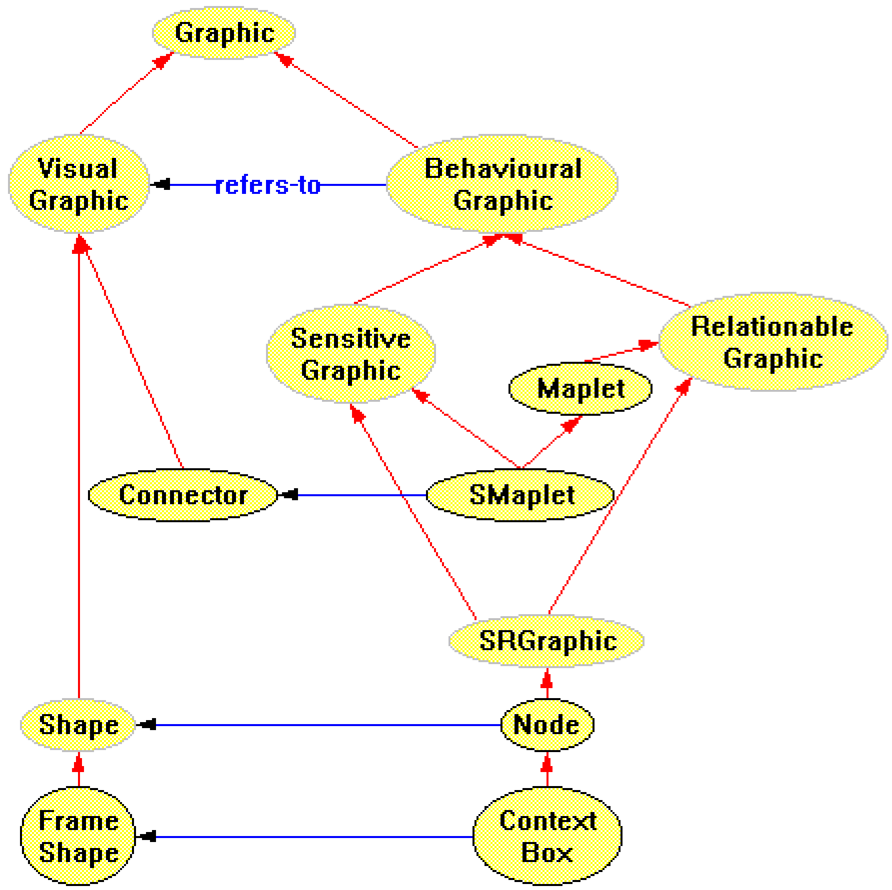


Figure 18: A simplified version of the CMap class library. Abstract classes are pictured with grayed borders; concrete classes are pictured with solid black borders.

The horizontal directed arc labeled "refers-to" in Figure 18 represents the actual delegate pointer between behavioural graphics objects and their delegate visual graphics. However, specific concrete behavioural graphics demand references to specific subtypes of VisaulGraphic. The unlabeled directed horizontal arcs in Figure 18 represent these "virtual" references between specific behavioural graphics and specific visual graphics. The behavioural graphics all use the more general delegate pointer in BehaviouralGraphic, but always access the visual graphic through a local method which

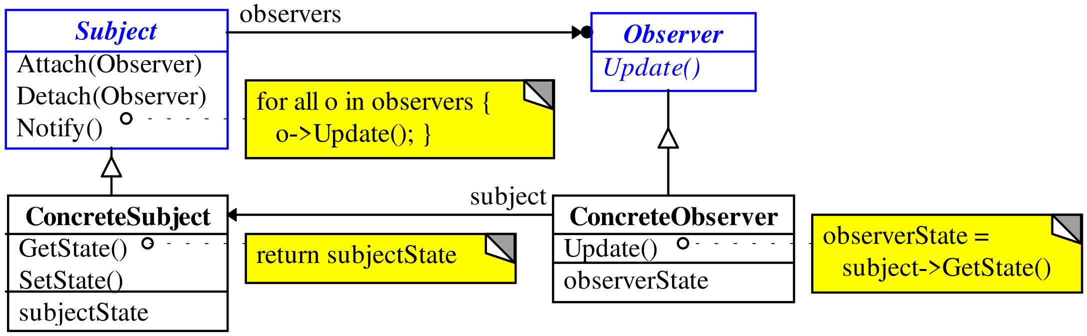


Figure 19: The observer pattern (Gamma et al. 1995)

ensures the type safety of the reference. For example, Node never uses the delegate pointer BehaviouralGraphic::Visual, but instead uses its local method,

```
Shape* Node::getShape()
```

which does the necessary run-time type checks and dynamic type caste to ensure the type safety of any operations.

There are specific behavioural classes which map onto the base types in Constraint Graphs: CMap's Nodes, ContextBoxes, and SMaplets (Figure 18) map onto Constraint Graphs' Nodes, Contexts, and Arcs (Figure 11 on page 64). There is no need for a special CMap class to model Constraint Graphs' *Isa* arcs: plain SMaplets will do.

Every component of the CMap graph is an observer of the corresponding component in the Constraint Graphs graph (Figure 20) according to the Observer pattern (Gamma et al. 1995) shown in Figure 19. Whenever a component in the Constraint Graph changes in any way, it calls its Notify() method, which sends the Update() message to each of the CMap components which are observing it. In this way, the visual image of every Constraint Graphs component is always easily and efficiently updated to reflect state changes, even if there are multiple views (CMap graphs) or multiple occurrences of the image of a Constraint Graphs component on a single view.

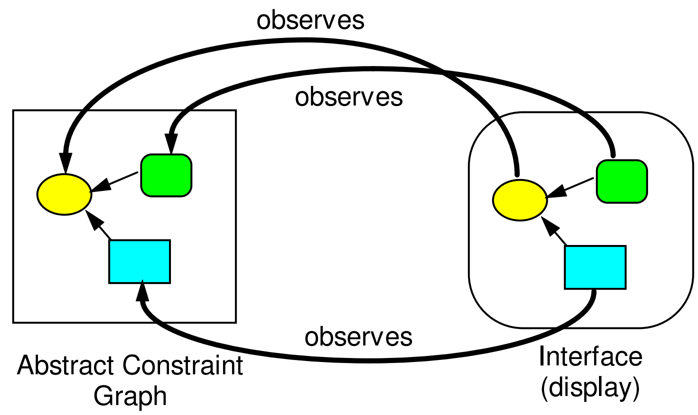
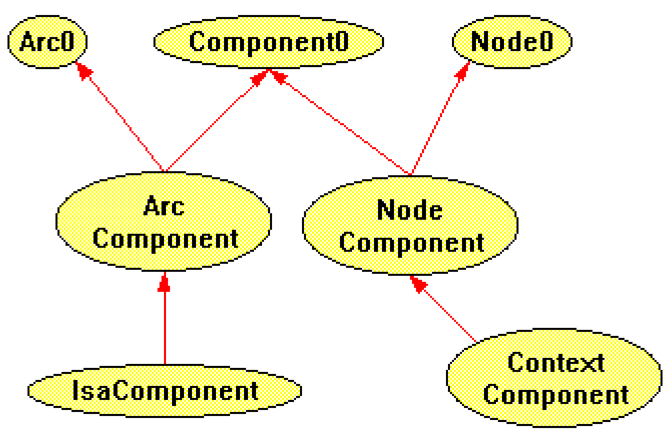


Figure 20: Interface elements are observers of corresponding components of the abstract constraint graph. Although only observations of nodes are pictured here, arcs also observe the corresponding arcs in the constraint graph.

6.3 The Constraint Graphs Class Hierarchy

This section describes the class hierarchy used in the Constraint Graphs implementation based on the specification in chapter 5. Figure 21 is a diagram of the class hierarchy.

The central part of Figure 21 is the sub-graph consisting of the *isa* tree rooted at the Component0 class:



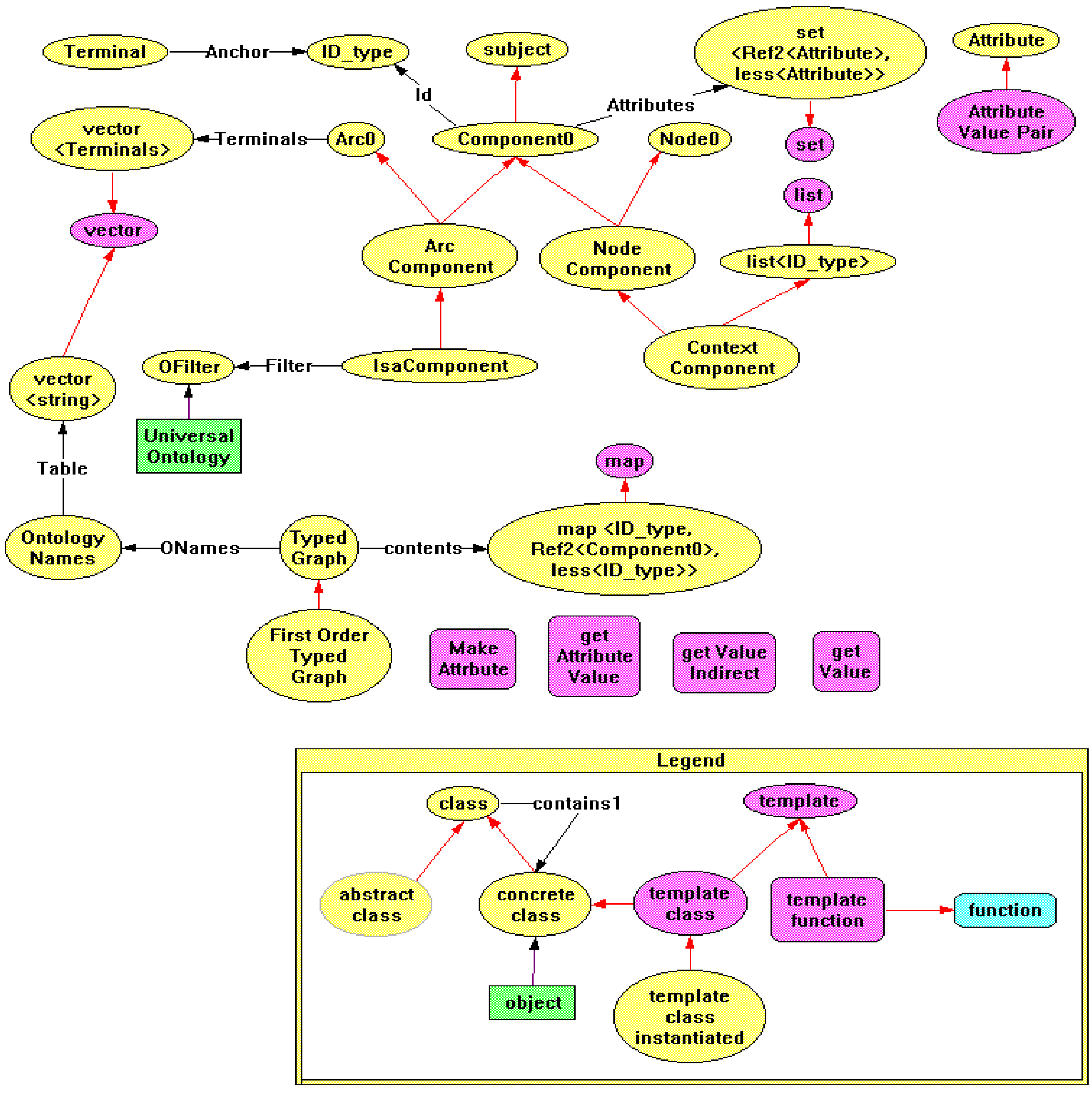
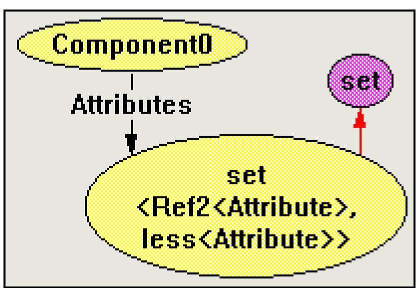


Figure 21: The class hierarchy for the ConstraintGraphs implementation

which is a close match with Figure 11 on page 64 (the base type lattice) and also very closely tracks the Z schemata in Sections 5.4 through 5.6. Technically, there is no reason why Arc0 and Node0 have to exist: the two classes should normally be removed and their functionality moved to ArcComponent and NodeComponent respectively. However, the constraints of the Z specification language require their existence (see the introduction to chapter 5), and the design merely tracks the specification's structure in this case. Having the class structure of the implementation model the structure of the specification also helps the hypermedia documentation (Section 6.5.4) in that one can hyperlink between corresponding structures in the specification and implementation.

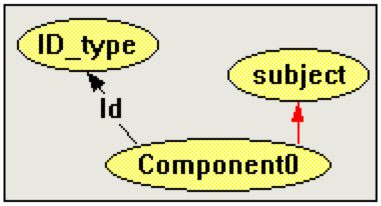
6.3.1 Component0

Component0 forms the base class for ArcComponent and NodeComponent and maps to Component in the specification (Section 5.6). As such, Component0 must have a name, a level, a set of attributes, and a set of constraints (or validators). Both name and level are left out of Figure 21 to avoid clutter. The constraints are also conspicuously absent: the implementation differs from the specification in that the implementation considers a constraint to be just a particular type of attribute/value pair. In other words, instead of having a specific constraint list, this implementation merely looks up the attribute "constraints" to find the constraint list.



Component0's attribute/value pairs are implemented as an STL set containing references to elements of the Attribute class, which is merely an attribute label (modeling attributes without values). The Attribute class is itself subclassed by a template which extends the base attribute to include values which are strictly-typed, according to **Requirement 21**. Variably-typed attributes are very difficult to express in Z, so this detail is glossed over in the specification.

An obvious alternative design is to implement attribute/value pairs as an STL map from a class string to a value class; but this design does not easily address the problem of type restriction on the value parts without breaking encapsulation.

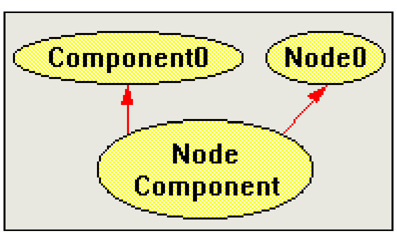


In addition, Figure 21 shows Component0 as having two relationships which do not appear in the specification. First, Component0 is a subtype of the subject class, which makes it's objects "observable" by other observer-class objects in the interface according to the observer pattern described in Section 6.1. Secondly, Component0 has a member, "Id", which is a unique identifier used to identify the object as a specific individual. In the Z specification, the identity function is implicit. Implementations often use the memory address as identity; but this implementation makes no such assumptions about address space because

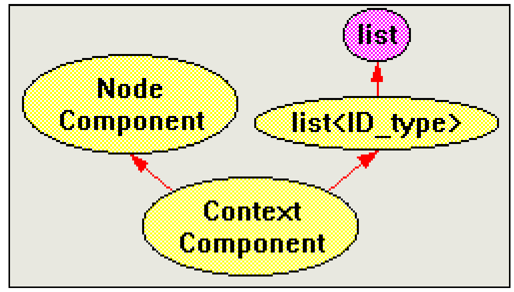
- objects may be copied and still retain their identity (common in STL containers), and
- objects may migrate among address spaces and among machines (see Section 8.3).

Thus, a unique identifier is required for every component. The identifiers are only unique within a single Constraint Graph, and the TypedGraph class is responsible for generating them.

6.3.2 Nodes

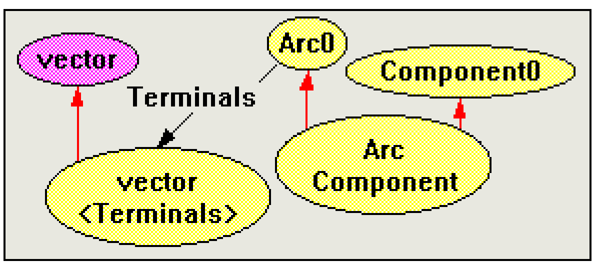


The NodeComponent class itself does not do a lot. It is not much more than a placeholder in the type lattice for a specialization of a Component0 that is mutually exclusive from an ArcComponent. NodeComponent can be subclassed to provide specialized behaviour. A case in point is ContextComponent.

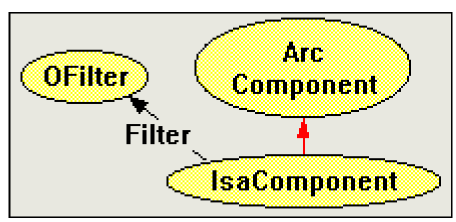


ContextComponent is a subclass of NodeComponent which extends it by adding the ability to contain sub-graphs (collections of other components). This is done by specializing an STL list of ID_types, which constitutes a list of references to other components. The list is restricted to contain only valid identifiers of components of the same graph which the ContextComponent object is a part of, as per specification Section 5.8. In addition, the list must not contain any unique identifiers that are contained in any other ContextComponent objects, which forces the ContextComponents to be strictly nested as per specification Section 5.10.

6.3.3 Arcs



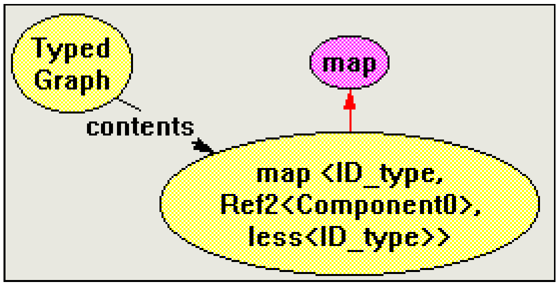
The ArcComponent class is another subtype of Component0, and extends it by adding an STL vector of Terminals. The Terminal class is simply an association of a component reference (a unique identifier) and a direction as required by the specification in Section 5.4.



The IsaComponent class extends ArcComponent by adding an ontology filter as per the specification in Section 5.5 and **Requirement 18**. The ontology filters are implemented as a long integer and are treated as a bit vector. This treatment is detailed in Section 6.3.4.

IsaComponents are also restricted to be binary and directed (Section 5.5 and **Requirement 8**).

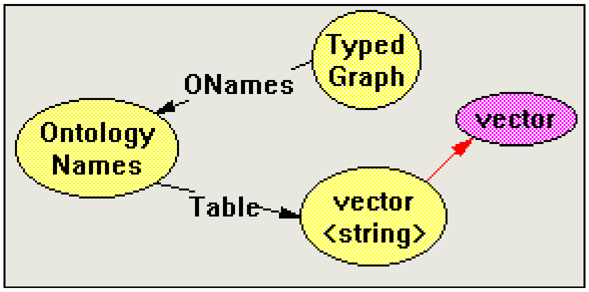
6.3.4 Typed Graph



The TypedGraph class is the implementation of the TypedGraph specification in Section 5.10. In fact, it is the merge of both the TypedGraph specification and the Graph0 specification in Section 5.8: Declaration dependencies could not be accommodated in the specification as easily as in the actual implementation. The TypedGraph class is primarily responsible to be a container for all the graph components. It does this using a STL map member that maps between ID_type (unique component identifiers) and indirect references to the corresponding components.

Since the TypedGraph class is frequently used to look up components by their unique identifiers, the choice of data structure here is critical. An STL map is a good choice because it is a fast version of a balanced binary tree – a red-black tree (Nelson 1995). Binary tree lookups and insertions have $O(\log_2 n)$ complexity. This is worse than a vector's constant complexity, but a vector is not appropriate because unique identifiers (the logical index) cannot be re-allocated after a component is deleted (since this may confuse external references). A graph implemented with a vector may eventually end up with much more empty space than actual data. At the other end of the spectrum is a list, but its linear lookup complexity is not efficient enough for the frequent lookups required. An STL hash table might be a better choice than a map, but hash tables are not yet part of the STL (Barreiro, Fraley & Musser 1995).

The TypedGraph class is also responsible for maintaining the invariants given in the specification in Sections 5.8 and 5.10. Also, as per Section 5.8, all typed graphs always contain the Constraint Graph base types (Node, Arc, Context, and Isa), as well as the *isa* arcs that constitute their type lattice (Figure 11 on page 64).



The specification deals with the ontology tags on *isa* arcs in a very abstract manner. The implementation has no such luxury. In the implementation, the IsaComponent's ontology tag set is implemented as a simple long integer which is treated as a bit vector. This is definitely not appropriate for the user, for the user will want ontology names to be strings instead of bit offsets. But string names are far too inefficient – both in terms of time and space – to tag on each IsaComponent. The compromise is to use bit vectors internally, but translate them transparently to strings for the user. This is one of the responsibilities of the TypedGraph class.

In order to do the translation, the TypedGraph class relies on the OntologyNames class which keeps a table of ontology names. This table is implemented as a simple STL vector of ontology names (as strings), where the index is the bit-offset of the IsaComponent's ontology tag bit vector.

6.3.5 Attribute Access Function Templates

Figure 21 shows several function templates: MakeAttribute(), getAttributeValue(), getValueIndirect(), and getValue(). These template functions specialize on the type of an attribute value. Recall from **Requirement 21** and the discussion in Section 6.3.1 that the type of the value in an attribute/value pair is arbitrary, but is strictly fixed when the attribute/value pair is inherited. The template functions provide type-safe access to these indeterminate types at run-time:

- MakeAttribute() is responsible for creating and associating an Attribute-subtype object with a component.
- GetAttributeValue() is more primitive than getVaue() and getValueIndirect() in that it is passed a component and an attribute name and extracts a particular attribute value

from the component, provided its type matches the expected type (an output argument).

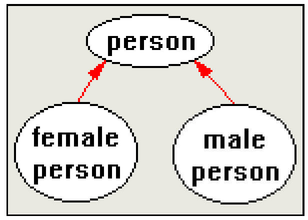
- GetValue() is passed a graph, a component ID, an attribute name, and an output type argument. It uses GetAttributeValue() to return the value if it exists and if it exactly matches the output argument type.
- GetValueIndirect() is similar to GetValue, but uses indirection in its output argument to allow the return of any subtype of the output argument type.

6.4 Using the Tool

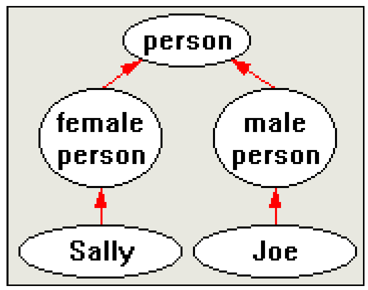
The previous sections in this chapter described the structure and design of the Constraint Graphs software. This section describes some of the pragmatic aspects of Constraint Graphs: how to use it. The next chapter will detail use of the tool further by describing how several other graphical formalisms are implemented in Constraint Graphs. In the interests of brevity, this section does not describe the details of the basic graph drawing interface; the interested reader is referred to the documentation on Constraint Graphs' predecessors (Gaines 1991b; Kremer 1993).

6.4.1 Creating a Type Lattice

In order to impose a type lattice on top of Constraint Graphs graph, it is only necessary to draw *isa* arcs among the relevant components. For example, the simple ontology of Section 4.3.3.1 is easily created by drawing *isa* arcs between *female-person* and *person*, and between *male-person* and *person*:



This world model can be just as easily extended to include some individuals:



Multiple inheritance is also allowed: one can draw more than one *isa* arc from a component. Furthermore, unlike some graph systems, arcs are first-class objects that may be the terminals of other arcs (this is illustrated in the next section). For Constraint Graphs, first-class arcs are a necessity: since *isa* arcs are subtypes of arcs, one could not create a type lattice of arcs unless arcs could terminate on other arcs.

As already described (**Requirement 19**, Section 5.11.1), attributes are inherited through the type lattice. One may edit arbitrary attributes via the attributes dialog box. For example, in the above example one might want to visually distinguish *Sally* and *Joe* as individuals instead of types by giving them rectangular surrounds and coloring them differently. An easy way of doing this to invent a new type called "individual", giving this the required visual attributes, and then having *Sally* and *Joe* multiply inherit from *individual*. The new component, *individual*, is created and assigned the appropriate attributes using the attributes dialog box as in Figure 22.

Attributes are annotated with a priority to disambiguate conflicts. In this case, *Sally* and *Joe* will inherit a shape value of "ellipse" from *person*, and a conflicting shape value of

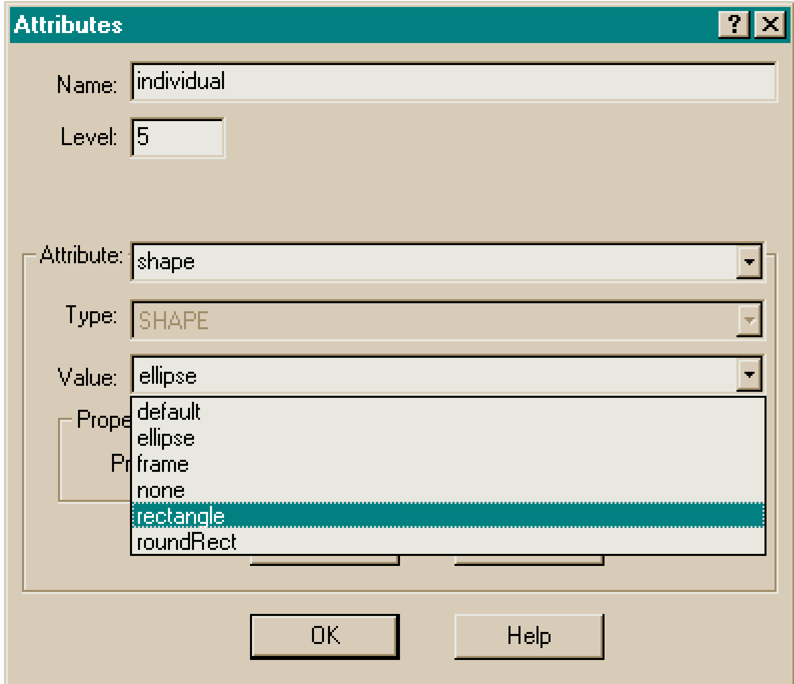
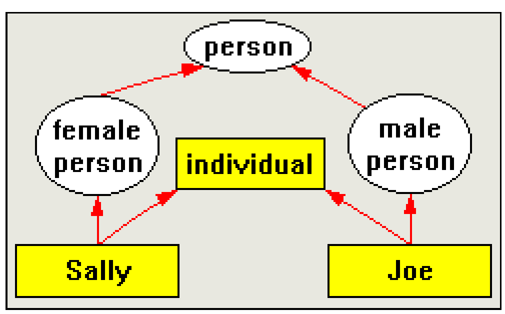


Figure 22: The Constraint Graphs attributes dialog box

"rectangle" from *individual*. One can resolve the conflict in favour of "rectangle" by assigning the shape attribute of *individual* a priority value less than that of *person*.

Finally, *isa* arcs can be drawn from *Sally* and *Joe* to *individual*:

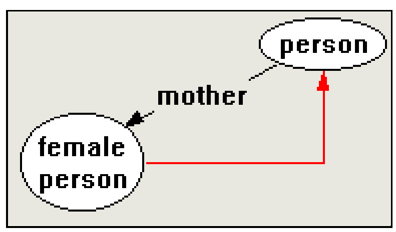


The inheritance system (Section 5.11.1) ensures that the visual attribute values of Sally and Joe are automatically changed. The observer pattern (Section 6.2) ensures that those values are automatically and immediately updated in the display.

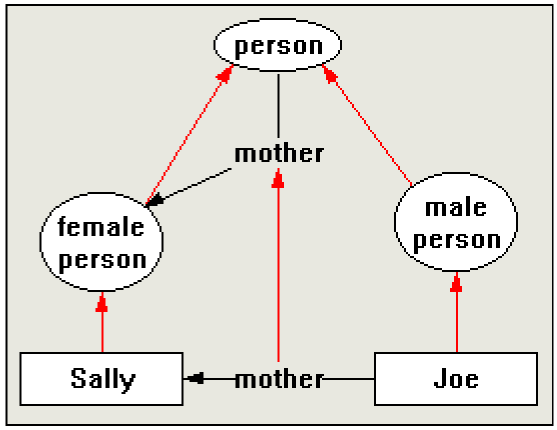
6.4.2 Constraining Arcs

Most graph-based visual language's syntax constraints revolve around constraining the component types on which arc types may terminate. Consequently, one of the fundamental operations in Constraint Graphs is restricting the component types on which arcs of particular types may terminate. This is quite easy to do in Constraint Graphs. Recall that there is no real difference between a component (object) and a type (class) in Constraint Graphs. It is therefore a simple matter of using the defining arc as the prototypical arc and connecting each of its terminals to components representing the most general legal types on which they may terminate. The ArcConformance validator (specified in Section 5.9) assures that no subtype can have a terminal on any component that is not a subtype of the component at the corresponding terminal of the supertype.

Assuming the *person* ontology of the previous section, if one wishes to create an appropriate constraint relationship called "mother", one need only draw in a *mother* relationship between *person* and *female-person*:



This automatically constrains any subtype of the *mother* arc to run between some subtype of *person* and subtype of *female-person*. For example, one might want to declare that *Sally* is the mother of *Joe*:



It would not be possible to draw the *mother* arc in the reverse direction because *Joe* is not a sub-type of *female-person*: the Constraint Graphs system would reject the transaction as shown in Figure 23.

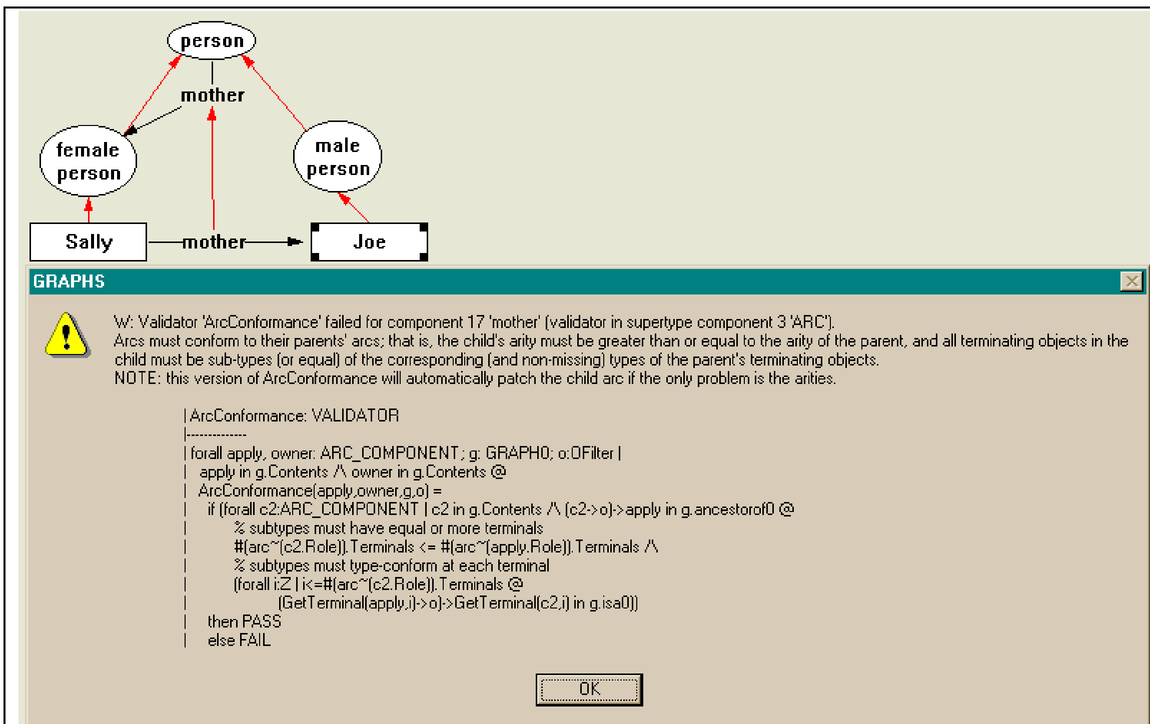


Figure 23: An attempt to draw an erroneous arc

6.4.3 Ad-hoc Constraints

Constraints on arc terminals are not the only constraints that arise in visual languages. Arbitrary constraints, such as the "first-order" constraint where no arc may terminate on another arc (described in Section 5.12), often arise. Other common "ad-hoc" constraints include:

- Arc arity constraints, where the number of terminals may be restricted to a specific number, or greater than or less than some number. Binary (arity equal to two) is by far the most common.
- Terminals of an arc may not be left dangling. (By default, terminals may be left dangling.)
- A particular component type may not participate in a multiple inheritance relationship.
- A component is considered an "object", and may *not* be a supertype to any other component.
- A component's label (or name) may be restricted to some particular syntax.

In addition, many other ad-hoc constraints (called Validators in the specification) that are used in the Constraint Graphs specification are described in Section 5.9.

It is likely not possible to express all conceivable constraints in the graphical syntax. It easy to argue that one should not attempt to: That would make the graphical syntax far too complicated to be useful. Furthermore, since future constraints may not be predictable, one would need the full expressiveness of a general-purpose programming language.

Therefore, arbitrary constraints are treated as arbitrary Boolean function objects which return false if the constraint is violated, and true otherwise. Constraint function objects can be written in the native language (C++) by a programmer and placed in Constraint Graphs' constraint library (see Section 6.5.2).

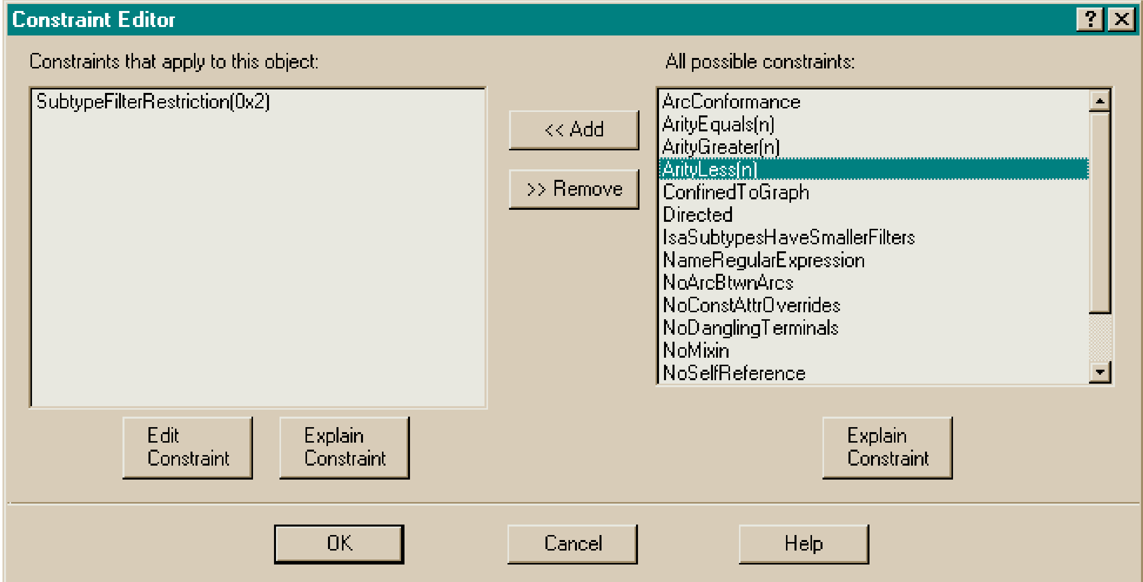


Figure 24: The Constraint Graphs constraint editor

A visual language designer (who isn't necessarily a C++ programmer) can choose constraints out of the constraint library and associate them with particular components (see Figure 24). Because the constraint function objects, like all functions objects, can have persistent state, constraints can have parameters. The selected constraint, ArityLess(n), in Figure 24 is an example: when the user presses the "Add" button, a dialog box will prompt the user for the parameter value. Each constraint may implement its own parameter dialog, so parameters may be arbitrarily complex, ranging from a simple integer parameter (for an arity constraint), to a set of regular expressions (for a label syntax constraint), to a script in a general-purpose programming language.

A constraint function object is called a Validator and has the following interface:

```
class Validator
  {
  public:
    Validator() {}
    Validator(const Validator& a) {}
    virtual bool operator()(Component0& apply,
                            Component0& owner,
                            TypedGraph& g,
                            const OFilter& f=UniversalOntology,
                            unsigned long flags=0);
    virtual Validator* clone() const;
    virtual int operator< (const Validator& r);
    virtual const string name() const;
    virtual const string specName() const;
    virtual const string doc() const;
    virtual const string spec() const;
    virtual unsigned long applicability() const;
    virtual bool primitive() const;
    virtual ostream& printOn(ostream& o) const;
    virtual istream& readFrom(istream& i);
  };
```

All validators that take parameters are known as EditableValidators. They extend the Validator interface:

```
class EditableValidator : public Validator, public EditableObject
  {
  public:
    EditableValidator();
    EditableValidator(const EditableValidator& a);
    virtual Validator* clone() const;
    virtual int edit();
  };
```

Constraints are attached to individual components and are checked whenever the component changes state. But it is not only the individual component's constraints that must be checked: both the constraints of all supertypes and of all subtypes must be checked to ensure they have not been violated. In addition, adjacent (as defined by arc connections) components must be checked, since a state change of a node may effect the legality of an attached arc and vice-versa. The actual check occurs whenever a state-

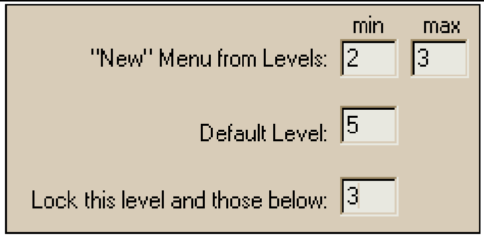


Figure 25: An excerpt from the options dialog box showing the level options

changing method is called; and if any constraint fails, the operation is undone. In this way, constraints always maintain the consistency of a graph.

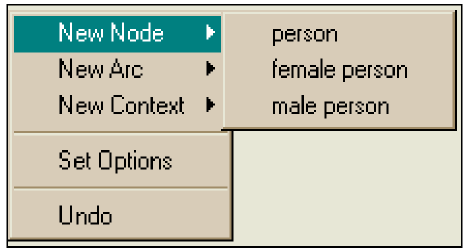
Constraints implemented in this way are reasonably efficient. They are efficient enough that they are used to implement most of the inherent conditions on constraint graphs, such as the binary, directed, and "upward" restrictions on *isa* arcs (see specification Section 5.9).

6.4.4 Using Levels

Levels are described in **Requirement 23** and in specification Section 5.1. This section illustrates some of the utility of levels. Figure 25 shows an excerpt from the options dialog box where a user may manipulate level options.

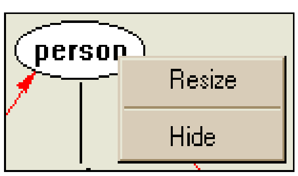
Since Constraint Graphs accepts any object as a potential type description, a user interface typically can display all of the existent components as potential initial types when the user requests creation of a new component. Clearly, this possibility is normally not what is wanted. Instead, the visual language designer would prefer to have the selection list come from a "primitive" set of potential types. In addition, the designer may wish to hide certain primitives (which are important to the system design but not relevant for the end user). Both these goals can be easily accomplished by limiting the potential creation types to a range of levels. An example case might be the *person* graph of this section, where the designer wishes to put the concepts (components) *person*, *female-*

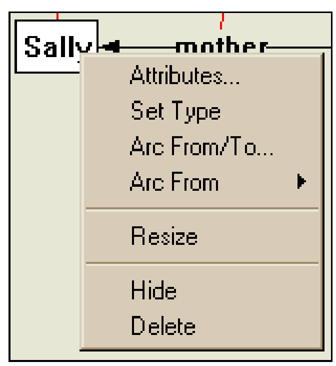
person, and *male-person* in the list of possible types, but wants to avoid doing the same for individuals (which the designer does not consider types) *Sally* and *Joe*. Thus a "new node" menu might look like this:



This can be done by setting the level of *person*, *female-person*, and *male-person* to 3, setting the level of *Sally* and *Joe* to 5, and limiting the range for "new" menus to include level 3 but not level 5 or level 1 (which includes the actual *node* type itself). The first line of Figure 25 shows how this is accomplished.

Visual language designers may also wish to lock out end users from editing fundamental constructs and accidentally disrupting the integrity of the language formalism. The following two figures show popup menus for *person* (level 3) and *Sally* (level 5) that result from level option settings shown in the last line of Figure 25 (where level 3 and below are locked):





Since *person* is a member of a locked level, it cannot be edited (except superficially), so its popup menu has been reduced to eliminate all modifying commands. On the other hand, *Sally* is not a member of locked level, so all commands are enabled on its popup menu.

6.5 Technical Details

The previous sections of this chapter discuss the structure of the Constraint Graphs implementation and the fundamentals of how it can be used to describe visual formalisms. This section is a catch-all which describes some of the more interesting technical details of the design and implementation. The topics include the flexible storage format, the flexibility and extendibility made possible by use of the librarian pattern, the cache optimization, and the hypermedia documentation.

6.5.1 Storage Format

The storage format is based on multi-part MIME (Borenstein & Freed 1993). A file (or network stream) is encoded as a simple multi-part MIME document where each part is a self-contained document describing a concept map. A primary advantage is that many concept maps may be stored or transmitted in a single file or stream. The main disadvantage is a significantly less compact format than is theoretically possible.

Just as Constraint Graphs is carefully kept independent of the interface, the Constraint Graphs data is kept separate from the interface (layout) data. This is accomplished by storing the Constraint Graphs data in its own multi-part MIME section with its own MIME type (called "application/x-ConstraintGraphs"), and the interface data in a separate MIME part (called "application/x-CMap"). See Figure 26. This scheme allows a great deal of flexibility, since not only may multiple x-ConstraintGraph/x-CMap pairs be stored in the same file, but also several separate x-CMap parts may follow a single x-ConstraintGraph part, which corresponds to several views of a single Constraint Graph. In addition, the multi-part MIME format allows for annotation with comments and a huge

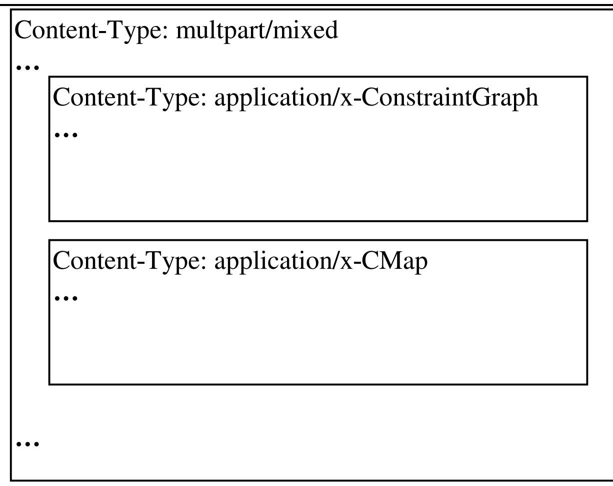


Figure 26: A schematic view of the Constraint Graphs storage format

variety of multimedia parts without disturbing the simple layout. (A Constraint Graphs program may simply ignore MIME parts for MIME types it does not understand.)

The multi-part MIME format also makes it easy for other programs to read CMap data up to the level that the "foreign" program may understand. The simplest case in point is KSIMapper (Kremer & Gaines 1996) that reads x-CMap data as its native storage format. It can easily read a ConstraintGraph file just by ignoring the x-ConstraintGraph MIME parts. KSIMapper cannot understand the x-ConstraintGraphs part, but there is enough information in the x-CMap part to allow it to do a very good job of displaying the data. Programs that read part of the data in this way should not be able to edit (and write back) the data as they cannot know of interdependencies within the data.

The reader may wonder how the x-CMap interface can infer the individual associations between components in the interface and components in the Constraint Graph. The answer lies with the x-CMap format: the x-CMap format allows each component to be tagged with an extension that may be ignored if not understood by the reading application. In this case the extension is merely the identification number of the

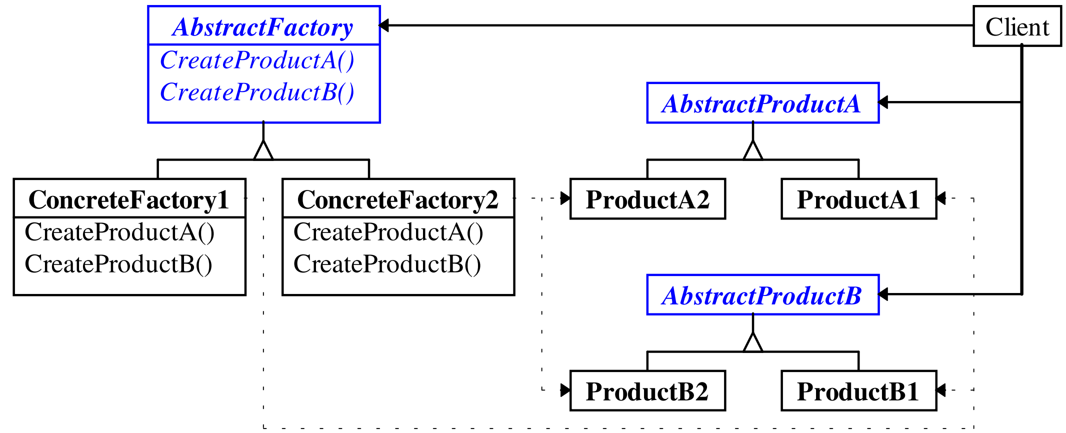


Figure 27: The abstract factory pattern (Gamma et al. 1995)

corresponding component in the Constraint Graph. See Kremer & Flores-Mendez (1996) for a complete explanation.

6.5.2 Flexibility and Extendibility: the Librarian Pattern

The Constraint Graphs software is not only a stand-alone program, but is also a framework (Pree 1995) that may be easily extended to implement other, unforeseen, applications. As such, it must be extremely flexible with respect to the exact classes it handles. The design restricts the superclass of all Constraint Graphs (and CMap) objects, but the layered application may dictate the exact classes (types) of almost all of the objects. Thus, a layered application can modify and extend object behaviour and functionality.

An obvious way to achieve this flexibility is by use of the abstract factory pattern as shown in Figure 27 (Gamma et al. 1995). Abstract factory is used in several places in the Constraint Graphs implementation. However, the abstract factory pattern relies on concrete classes being available at compile time; whereas there are several cases in Constraint Graphs where the exact type of an object to be created is not known until run time. Examples include:

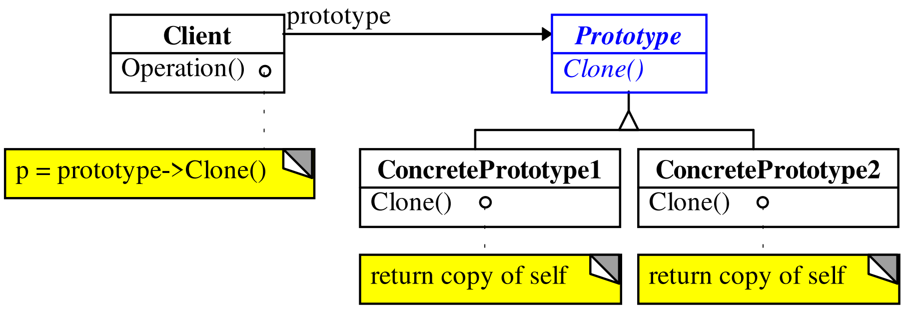


Figure 28: The prototype pattern (Gamma et al. 1995)

- The exact type of a Constraint Graph component is not known until it is read in from a file or a stream at run time. The name of the exact class is read in with each component's record.
- A similar situation arises when both CMap's behavioural and visual graphics objects (see Section 6.2) are read in from a file or stream.
- A CMap behavioural graphic may be required to change the type of its delegate visual graphic object (i.e.: it must discard the visual object in favour of a new visual object) at run time. Typically, this event happens in response to the user selecting a new type in a dialog box. The new visual object must be dynamically constructed based on the text string supplied by the selection.

This problem is addressed by a new pattern called the library pattern.

The library pattern is an extension of the prototype pattern (see Figure 28) documented by Gamma et al (Gamma et al. 1995) and is used extensively in the Constraint Graphs implementation. The library pattern extends the prototype pattern by replacing the client's direct reference to the prototype with a container (ObjectLibrary) which acts as an intermediary (see Figure 29). The ObjectLibrary container holds any number of prototypes which are indexed by string names. Instead of the client directly cloning a prototype, it asks the ObjectLibrary to do the cloning on its behalf, passing a string

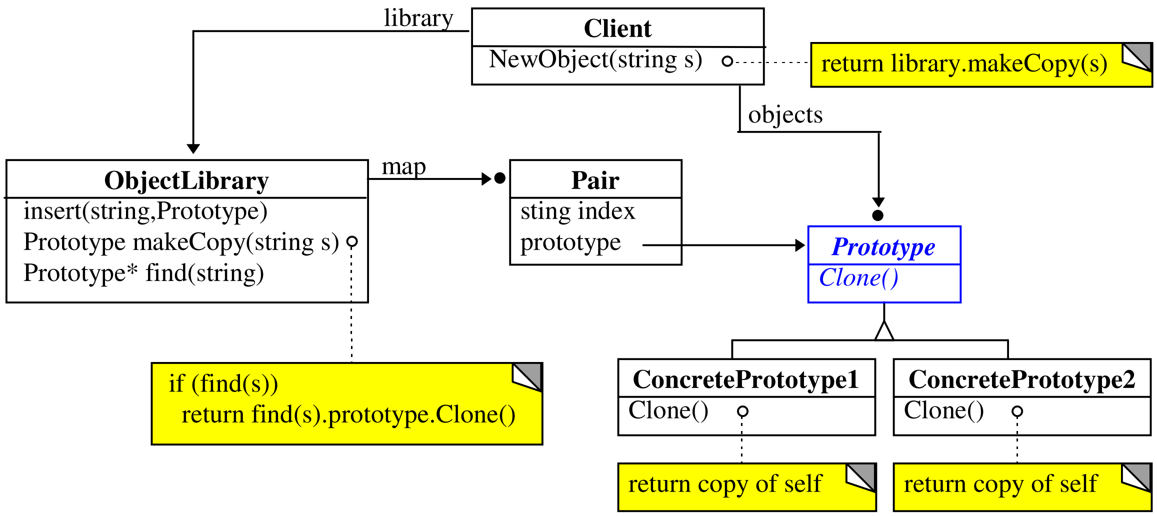


Figure 29: The library pattern

parameter to specify the prototype object it needs. It is easy for the implementation to initialize the ObjectLibrary's contents at run time.

One danger with using the library pattern as shown in Figure 29 is that the client may expect the returned object to conform to a specific type specification. But there is nothing in the figure to restrict the type of the cloning object (beyond being a Prototype). In order to ensure type safety for clients using ObjectLibraries, the library pattern is implemented as C++ template (see Figure 30). Thus, a client using an ObjectLibrary of type ObjectLibrary<VisualGraphic> can always be guaranteed to retrieve an object of type VisualGraphic*.

6.5.3 Optimization: Caching

An interesting aspect of implementing Constraint Graphs from the specification is the structural differences between the specification and the implementation. There are, of course, differences that spring from the inclusion structuring of the Z specification

```
template <class T, int hasDefault=1>
class ObjectLibrary : public map<string,T*,less<string> >
  {
  public:
    T* makeCopy(const string& x) {
      T* ret = access(x);
      if (ret) ret = ret->clone();
      return ret;}
    T* access(const string& x) {
      iterator i;
      T* ret = NULL;
      if ((i=find(x))!=end())
        ret = (*i).second;
      else if (hasDefault && (i=find("default"))!=end())
        ret = (*i).second;
      return ret;}
    pair<iterator,bool> insert(const string& x, T* t) {
      return map<string,T*,
        less<string> >::insert(pair<const string, T* >(x,t));}
    pair<iterator,bool> insert(T* t) {
      return map<string,T*, less<string> >::insert(
           pair<const string, T* >(typeid(*t).name(),t));}
  };
```

Figure 30: The library pattern template

language versus the object-oriented inheritance structuring of the C++ programming language. This is detailed in the introduction to chapter 5. But there are also efficiency considerations which dictate style differences between the specification and the implementation. Unfortunately, these differences tend to cloud what should otherwise be a tidy one-to-one correspondence between the specification structures and implementation structures. Nonetheless, these differences cannot be avoided.

As a case in point, the algebraic style of the specification is easily translated to C++ code, but leads to extremely inefficient C++ code. For example, the specification of the operator *ancestorof* is specified as

```
forall c,p:Contents; o:OFilter @
   (p->o) ancestorof c <=> paths_btwn(c,p,o) /= {};
```

which would lead to some fairly inefficient C++ code: A direct translation would be a double-nested *for* loop iterating the entire contents of the graph n^2 times (assuming n is the number of component in the graph). Furthermore, the code would compute paths_btwn() n^2 times; since the literal translation of paths_btwn() is itself an polynomial function, the time complexity of *ancestorof* is very high.

An initial implementation which factored out paths_btwn(), but used the double-nested *for* loop, took close to a full second to compute the ancestorof() function for a reasonably large graph. An optimization which cached the *parentof* relationship as local pointers (and used this relationship instead of paths_btwn()) could perform the same operation on the same graph in less than 0.001 second.

This dramatic increase in efficiency comes at the cost of corresponding alienation between the specification and implementation. As the appropriate optimizations are applied to the implementation, there comes to be a less and less clear correspondence between the specification and implementation. Thus, the formal specification gradually becomes less and less useful as a documentation source or as guide to code maintenance. This problem can be partially mitigated by the use of hypermedia documentation, described in the next section.

6.5.4 Hypermedia Documentation

The previous section discusses the problem of alienation between the specification and implementation; this section explores at least a partial solution. The Constraint Graphs specification is written in the form of an HTML document as described in the introduction to chapter 5. To enhance the correspondence between specification and implementation, each part of the specification is tagged with a hypertext anchor. HTML markup tags are added in comments in the C++ source code and header files of the implementation. Thus, both the specification and implementation files can easily be given hypertext links that link to the corresponding passages in the implementation or specification. For example, the implementation of the TypedGraph class is tagged as follows:

```
/*<a name="classTypedGraph">
  <a href="GraphsZ.htm#GRAPH0">Z spec (GRAPH0)</a>,
  <a href="GraphsZ.htm#TYPED_GRAPH">Z spec (TYPED_GRAPH)</a>,
  <a href="graphs_cpp.html#classTypedGraph">implementation</a> */
class TypedGraph : public RefCountingObject
  { . . .
```

The code may then be converted into HTML documents (which only involves copying them to files with extension ".html" since the HTML markup tags are already included as comments) and the specification and C++ files refer to one another. This makes it easy for programmers to browse the specification, then look at the corresponding C++ code quickly and easy by following a hypertext link. Likewise, if programmers use WWW browsers to browse the C++ code, they can quickly and easily retrieve the full specification in context. The reader may be interested in seeing this documentation at

http://www.cpsc.ucalgary.ca/~kremer/graphs/GraphsZ.htm

6.6 Summary

This chapter has provided an overview of the Constraint Graphs program design and implementation. The most important point is the clear separation between the abstract Constraint Graph and the user interface – the visualization of the Constraint Graph. Section 6.2 describes the CMap library that acts as the interface and Section 6.3 describes the class library design for Constraint Graphs itself.

Section 6.4 is more pragmatic and describes the actual use of the Constraint Graphs program: how to create a type lattice within the Constraint Graph, how to constrain the types of the objects at arc terminals, how to use ad-hoc constraints, and how to use ontological levels to enhance the user interface for an end user.

Section 6.5 picks up on some of the more interesting technical details including:

- the multi-part MIME storage format, which separates Constraint Graph concerns from user interface concerns
- the library pattern used in the program, which is used extensively to enhance flexibility and extendibility.
- caching of arc relationships, which is a dramatic speed optimization over the "obvious" implementation from the specification
- hypertext documentation, which helps to keep the association between the parts specification and the corresponding parts of the implementation.

This chapter serves as documentation of *one* possible design based on the specification given in Chapter 5. There are always many possible designs that may arise from a good (not over-specified) specification. This design illustrates the many tradeoffs between conflicting "soft" goals such as clear conformity to the specification, easy-to-understand code, run-time efficiency, modularity, modifiability, flexibility, reuse of pre-existing components, and production of reusable components. For example, the design placed higher priority on clear conformity to the specification than on easy-to-understand code, which resulted in a very clear correspondence between specification and implementation, but less-than-desirable class hierarchy where unnecessary C++ classes were introduced to match the (necessary) corresponding Z schemata (as discussed in Section 6.3). In another example, the design placed higher priority on clear conformity to the specification and easy-to-understand code than on run-time efficiency, but this had to be compromised when it was found that the frequent following of *isa* arcs in the style of the specification lead to unacceptably slow run-time performance. Caching of *isa* arc references had to be added to alleviate the problem (Section 6.5.3).

On the other hand, reuse of well-designed pre-existing components took place without compromise both at the design and at the implementation levels. Several of Gamma et al's (1995) patterns were reused in the design which served to clarify the design and help document the code. The C++ Standard Template Library was also reused extensively without any compromise: the STL components are very close models to the constructs of the Z notation, and so translate almost exactly while still maintaining impressive efficiency and type safety. For example, the STL *set* template class corresponds to the Z *set*, and the STL *vector, dequeue*, and *list* template classes all corresponds to the Z *seq* (sequence).

The documentation of the design given here is not complete, but illustrates most of the major decisions. It should be obvious that the design touches on many areas that are not within the domain of the specification. It has already been stated that the specification does not cover the user interface, but there are many other areas that the specification purposefully does not cover but do appear in the design. For instance, the specification should not (and does not) cover the storage format, or even *if* the data should be stored; but this topic is covered in the design (Section 6.5.1). The specification defines *what* a concept map is (in terms of abstract data structures and operations), not *how* it is implemented or the specific data formats. Details such as storage formats are merely design details (unless storage format is what is being specified – which it isn't), and so these decisions are deferred from specification to design.