Constraint-based Document Presentation

Wayne A. Christopher *
Computer Science Division (EECS)
University of California
Berkeley, CA 94720

October 1990

ABSTRACT

Constraint-based programming has been used for a wide variety of applications where declarative specification and general solution mechanisms are desirable. This paper describes a prototype document preparation system, **Ensemble-C**, that utilizes constraint satisfaction as a mechanism for presentation maintenance and incremental formatting. The goal of the overall **Ensemble** project is the integration of a wide variety of media, including computer programs and dynamic media such as sound and animation, into a coherent framework that provides incremental formatting, multiple-representation editing, and separate structure and presentation specification.

*Sponsored by the Defense Advanced Research Projects Agency (DARPA), monitored by Space and Naval Warfare Systems Command under Contract N00039-88-C-0292, and in part by the National Science Foundation Infrastructure Grant number CDA-8722788.

Ensemble is a multi-media multi-representation language based editing system. It is intended to be a framework for the integrated support of interactive development of complex natural language and formal language documents. The system will provide multi-media capabilities, as well as facilities for formal description of syntax, semantics, and transformations of structured objects and representations.

The user will be able to view and edit compound documents composed of many types of components represented in a variety of media. The editing and viewing capabilities will handle these heterogeneous structures in a natural and convenient way.

The system will enable the user to edit and view documents in many views simultaneously. For example, views might present certain portions of a document and hide others, as in program holophrasting. They might display the logical structure of a document explicitly, as in a tree-structured representation. They might display formatting or other directives to the transformation process, as in the source view of a formatting system. They might display the output of such a transformational process, as in the proof view of a formated document or the pretty-printed view of a program. The support for a rich set of media types will provide an opportunity to enrich the possible kinds of views significantly.

The version of **Ensemble** described here is **Ensemble-C**, where the "C" stands for "Constraints". Another version of **Ensemble** is being developed in our group which focuses on the editing of programs and is based on the **Pan 1** system [Ballance *et al.* 1990]. This report will not discuss the language-editing mechanisms of the system. A general overview of the **Ensemble** project will be given in a forthcoming technical report.

The central ideas used in **Ensemble-C** are the clear separation of structure from appearance, integrated editing for all media types, simultaneous editing of multiple representations of a single document, and the use of constraints as a unifying mechanism for presentation.

Systems such as LaTeX[Lamport 1986], **Scribe**, [Reid 1980] and **Grif** [Quint and Vatton 1986] have all emphasized the independence of logical structure and appearance, and **Quill** [Chamberlin *et al.* 1987] has as a goal the seamless integration of different object types. Both **VorTeX** [Chen 1988] and **Lilac** [Brooks 1988] allow the editing of text documents in both a direct manipulation view and a structural language view at once.

Constraints have been used for graphical user interfaces in **Garnet** [Myers *et al.* 1989], physical simulation in **ThingLab** [Borning 1979], and electrical simulation [Steele 1980]. Other work in constraint-based programming is described in [Leler 1988], but our notion of constraints is closest to that of **ThingLab**. Constraints will be described further in Section 5.

1 Overview

In **Ensemble-C**, documents are represented as trees. The nodes of the trees are objects such as paragraphs, words, and statements in a programming language, and the arcs correspond to logical inclusion.

Document trees cannot be edited or looked at directly. Rather, the document is presented to the user via one or more **views**, or ways of looking at a document. A view may elide certain parts of the document, or present them in an order different from that in the base document tree. Views always display the most current presentation of a document available, and incrementally update this presentation when the base document changes.

The basic document tree contains only the minimal information necessary to specify the document, and is common to all views. All other data is kept in the views that need it. This includes formatting data for proof views of text and rendering information for graphical views. Most view data is transient – it can be recreated from the base document when needed – but there can be permanent view data that should not be thrown away, such as user modifications to the appearance of a particular view.

A view can appear in a **window**. If a view is thought of as an infinite plane, a window can be thought of as a rectangular portion of this plane. A window also has a cursor, which may point to any part of the document tree. The notions of windows and buffers found in Emacs [Stallman 1986] roughly correspond to ours, except for the intermediate view structure placed between them.

Note that for dynamic media such as animation and sound, more general models of windows and views will have to be formulated. Although this work is currently in progress, in this paper we will limit our discussion to the facilities necessary for supporting static media.

Since the goal of **Ensemble** is multi-media editing, it must support many different types of nodes in the document tree – text, graphics, sound, video, and so forth. Although each of these major types has many sub-types, such as words and paragraphs, these divisions are basic enough to allow us to group together a great deal of functionality for each one. We call these major types **genres**, and encapsulate all genre-specific code and data in **genre modules**. These include the following types of information.

- **Structural schemas**, which are essentially grammars that direct the editor in creating documents. A structural schema defines an allowable document structure for a particular genre.

- **Presentation schemas**, which are sets of rules for enforcing the relationships between document components in views. In some other systems, this function is carried out by "style files". Since **Ensemble** is designed to be interactive and to support a wide variety of media types, we require a more complex model than a simple listing of parameter values. Our presentation schemas utilize constraints and rules for when these constraints should be activated.

- **Rendering methods**, used by the graphical output system to draw objects that have a visual representation. The window package provides certain primitives for the rendering methods to use, but in general every visible facet type must have its own rendering methods.

- **I/O and translation mechanisms**, which are required for saving different object types to disk and importing or exporting them to other representations, such as TEX.

2 Views and Facets

One way to think of the relationship between views and the base document is that each view annotates the document tree with its own view data, such as font information for a pretty-printing view, or exact positions of objects on the screen for a view of a figure. This was the approach taken in the **Pan 1** system [Ballance *et al.* 1987].

Another way to think of this is that each view constructs its own parallel tree, which is usually similar to the basic document tree, to hold its own data. Since the nodes in these trees can have explicit pointers to the corresponding nodes in the other tree, the functionality is the same as that in the annotations model. As will be seen, however, there are other benefits to the parallel trees model that make up for its increased complexity and overhead. For example, if we wish to construct views of other views, rather than views of the basic document, or if we wish the view tree to have a slightly different structure than that of the base tree, which is a common occurrence, having separate trees is very useful.

Ensemble-C uses the second model of views. The base document tree is also treated by the system as a view, and is referred to as the **base view**. We say that one view is **derived from** another if it depends on that view.

We call the nodes in these trees **facets**. This is most apt for the nodes in view trees, since each can be thought of as one way of looking at a many-sided object. Every facet has an **owner**, which is in the base view, or, more generally, in the view that this facet's view is derived from. Note that more than one facet in a particular view may be owned by the same facet in the base view. For example, in the base document structure, an *if* statement contains three sub-parts: the condition, the *true* block, and the *false* block. In a pretty-printed view, it contains these three parts, but it also contains facets that represent the words "if", "then", and "else". They are owned by the *if* facet in the base document tree, whereas the other three facets are owned by the corresponding facets in the base view.

An example of a view that doesn't depend on the base view is a paginated view of a text document. The first level view uses an infinite scroll model, without page breaks. The paginated view, which depends on this one, arranges the paragraphs, possibly breaking them in the middle, into fixed-size consecutive pages. The major structural difference is that one paragraph may own two or more paragraph segments, and the words that are children of the paragraph will be children of these segments in the paginated view.

It is important here to distinguish the concepts of facet parenthood and facet ownership. The parent of a "paragraph-proof" facet might be the "section-proof" facet it belongs to, in the same view. Its owner will be the corresponding "paragraph" facet in the base document tree. A "paragraph" base facet might own facets of type "paragraph-proof", which in turn might own facets of type "paragraph-proof-paginated".

Thus there are a number of ways to look at the tree-structuredness of a document, as pictured in Figure 1. First, one may consider the tree of views currently in use. Second, one may pick a particular view and look at the hierarchical structure of the document within that view. Third, one may

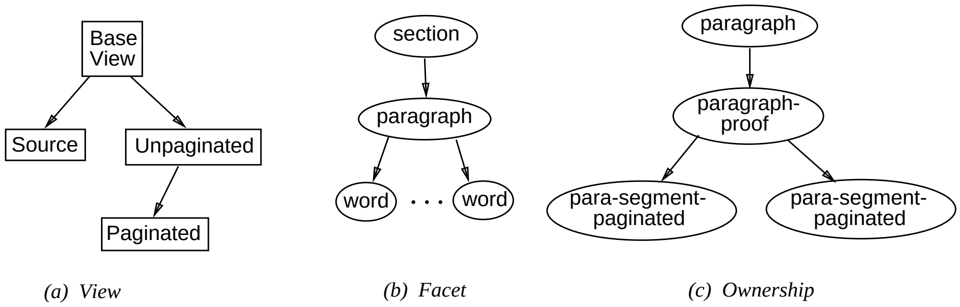

Figure 1: Tree Structures

pick a particular object, say a **paragraph**, and, following the tree structure of the view hierarchy, look at the facets which are indirectly owned by that object. Note that this tree may not be isomorphic to the view tree in (a), since a facet may own more than one facet or no facets at all in a particular view.

3 Architectural Overview

In this section, the basic structure of **Ensemble-C** will be described. There are four main components of the system. The relationships between these components are shown in Figure 2.

1. **User interface.** This reads commands from the user and passes them to the editor core. It also accepts graphical output requests from the genre-specific rendering code and manages windows.

2. **Editor core.** The editor accepts commands from the user interface and makes appropriate modifications to the base document tree. When it is finished, it asks the constraint system to bring all the views of the document up to date.

3. **Constraint system.** Consistency between views is maintained by means of constraint satisfaction. After the editor has modified parts of the graph, the constraint system must first bring the structure of the constraint graph up to date, and then must modify the values of the

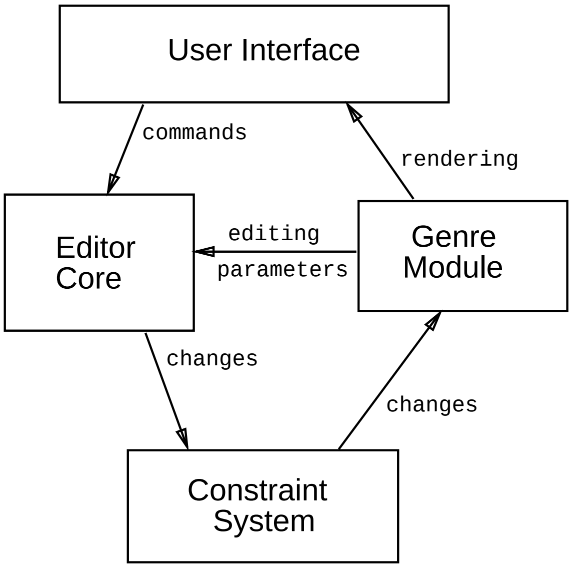

Figure 2: The Basic Structure of Ensemble

objects in the graph so that they satisfy the constraints. This will be explained in the section on constraint satisfaction.

4. **Genre-specific modules.** The genre module contains rendering code, which is invoked by the constraint system whenever a facet is determined to have changed in such a way as to require redisplay. Additionally, it contains structural and presentation schemas, and facet-specific routines and commands that can be called by the editor.

3.1 User interface

The user interface consists of two packages, one to handle user input, and one to handle graphical output. For simplicity, these are rather loosely coupled. When the user makes a modification to the document, the flow of control passes from the input module to the editor core, to the constraint system, and finally to the graphical output package. There is no provision for short-circuiting between the user input package and the graphics package yet, although this will probably turn out to be necessary for adequate performance.

Each window has a *cursor*, which can point at any facet in the document tree. In the case of a leaf facet with further substructure, such as a word, the cursor may have a location within the facet. In the case of a word this is an index into the string. All operations are done using the cursor, although a selection mechanism is currently being developed [Munson and Pan 1990].

Currently, the user interface maps control keys to commands by means of a static table. Other interaction with the user is done through the standard input and output of the **Ensemble-C** process, which is rather awkward. Instead of improving the current user interface, we expect to replace it and the editor core with the *Pan* editor [Ballance *et al.* 1990, Ballance and Van De Vanter 1987].

3.2 Editor core

The editor core is responsible for interpreting and performing commands given by the user interface, and managing the event queues that drive the presentation system.

3.2.1 Commands

There are a number of types of commands that the editor handles.

- **Cursor movement** commands such as "move up" or "move to next leaf node" are relatively straightforward – the editor calculates the correct facet to move to, and tells the window module to move the cursor appropriately.

- **Object modification** commands cause fields in the base facets to change in facet-specific ways. These changes trigger the constraint satisfaction process.

- **Structure modification** commands include creation and deletion commands. These cause the base document tree to be modified, along with the facets that are owned by the affected base facets. These operations trigger both constraint graph maintenance and constraint satisfaction.

- **System** commands, such as "create document", or "output PostScript file", are dealt with on an ad-hoc basis.

The structure modification commands take a number of forms. The simplest one, which is bound to the space character, creates a sibling of the current facet with the same type. A command to create a sibling of the parent facet is bound to the return key. In the case of words and paragraphs, these are the usual semantics of these keys. Also, a command to allow the user to select from a range of facet types is bound to the combination *control-space* – if the parent of the current facet is a **sequence** node [1], and it can contain children of more than one type, the user is prompted to choose one of them. The *control-return* function is similar.

3.2.2 Facet Creation

The mechanism for creating new facets is rather complex. This is because of the wide variety of options that can be given, both for the positioning of the new facet in the tree, and for the actions that should be taken with regard to the subtree below the newly created facet. We create new facets in the following situations.

- Creating a new (empty) document. We would like to create the root facet in the base view, and let the structural schema guide the creation of all the required children of the root, and their required children recursively.

- Reading a document in from a file. In this case, the entire structure of the document is specified, and we want to create exactly the facets described. However, we may want to check the correctness of the structure with respect to the structural schema to which it claims to conform.

- Building a view tree in response to changes in the base tree. In this case, we also want to create facets without referring to a structural schema, since only the base view is governed by structural schemas.

3.2.3 Event Queues

The internal process of presentation maintenance is event driven. Every change made to a view tree is noted in a number of queues – the **presen-**

[1] As a matter of policy, whenever a list of unspecified length is called for, a special node called **sequence** is used to contain the facets in the list. No other facet types are allowed to have arbitrary numbers of children.

tation queue for that view, and the **structure queues** for all the views that are derived from it. Events in the presentation queues cause constraint graph maintenance, as described in Section 5, and events in the structure queues cause the structure of the views to be updated.

Events in **Ensemble-C** are serviced according to the following priorities.

1. Input from the user.
2. Events in structural queues.
3. Events in presentation queues.
4. Constraint graph solution.
5. Facet redisplay.

3.3 Genre-specific modules

A number of things must be done specially for every genre.

- Facet types must be defined. Some facet types are built-in, such as "sequence", but most have special properties. For instance, words have strings as values, whereas most facets don't have values.
- Methods must be defined to modify facets that are capable of modification. The system defines a few generic change types such as "add characters", and lets the facets implement the specifics themselves. Note that other types of changes such as "delete node" are done by the editor directly.
- Facets that can draw themselves must have their own "render" methods.
- Constraint types and presentation rules must be defined for every view of every genre.
- Structural schemas must be provided for the editor.

4 Structural Schemas

In several places in **Ensemble-C**, it is necessary to describe either an actual structure, or a grammar rule for building a structure. In the structural rewrite rules for maintaining view trees, we need to describe the facets that must be created in response to a change in the base view. A printed representation of the document tree must be generated or parsed when a document is saved to or read from a file. In the `defgrammar` form used to define a structural schema, we also have to describe a tree structure, although we are specifying rules instead of an actual tree. These cases are similar enough that the same mechanism and notation is used for all of them.

The basic element of a structure description, the **facet entry**, has the following form.

$$(\textit{type}\ [\ \textit{keyword value}\ ]\ \ldots)$$

The *keyword* may be any of the following.

- **:children.** The corresponding *value* is a list of child specifications. These are of the form

  $$(\textit{name type}\ [\ \textit{min}\ ]\ [\ \textit{max}\ ])$$

  where *name* is the name of the child, such as `:title`, *type* is either a symbol that specifies the type of child that can go in this position or another facet entry, and *min* and *max* determine the permissible number of elements that can be present here. The *min* and *max* arguments can only be used if the *type* argument is a symbol, and are really only a convenient abbreviation for explicitly describing the sequence facet that should go in this position. *max* can be `*`, which indicates no limit on the size of the sequence.

- **:id.** The *value* is the unique id of this facet. This is required in some cases where we need to preserve information such as the links between a citation and the bibliography entry it refers to. If an id isn't given the system will generate a unique one for the facet when it is created.

- **:owner.** The *value* is the unique id of the facet that should be the owner of this one. This is required for structural rewrite rules, described below.

- **:slots.** The *value* is a list of *slot-name, slot-value* pairs. Each *slot-value*, which must be a a string, number, keyword, or list structure containing these, is inserted into the named slot in the facet. The value of a slot may also be another facet, in which case the id of that facet should be given instead.

An example of a structural description used in a grammar is:

```
(section :children ((:title block)
                    (:blocks block 0 *)
                    (:sections section 0 *)))
```

The "*" max argument denotes that there are no limits on the number of children that may appear in the sequence.

An example of a structural description used in a structural rule is:

```
(paragraph :id g0 :children
    ((:body (sequence :children
                ((nil (word :id g1 :slots ((value "Test"))))
                 (nil (word :id g2 :slots ((value "Stuff")))))))))
```

This differs from the previous example in a number of ways. First, all child entries are full facet descriptions rather than just names. Since we want to express the detailed structure of the subtree rooted at this point, this is necessary. Second, facet ID's are included, which are needed for preserving certain structural information like cross references. Third, slot information is given for the words.

Structural descriptions are also used as document tree grammars, or *structural schemas*. A structural schema is created with the `make-grammar` function. For example:

```
(make-grammar *text-genre* "article"
             '((document :children ((:title paragraph)
                                    (:body block 1 *)))
               (block -> paragraph foreign)
               (paragraph :children ((:words word 1 *)))
               (foreign)
               (word)))
```

The first argument, `*text-genre*`, specifies a **genre** object that this grammar should be associated with. The second argument is the name of the document type being defined, in this case `"article"`. Then follows a structural description of the valid structure for this document type.

In addition to this form of structure description, macros may also be used in `make-grammar` forms. An entry of the form

$$(name_0 \ \rightarrow \ name_1 \ldots name_N)$$

makes it possible to use $name_0$ in a child entry, in which case the system will allow any of the succeeding $name_i$'s in that position.

The last two facet types given in the example, `foreign` and `word`, have no information specified other than their name. This is because the `foreign` facet name is really a placeholder for a subtree of another, unspecified genre, and words are leaf nodes in the text genre and thus have no children.

More complete examples are given in the appendices.

5 Presentation Schemas

In **Ensemble**, the appearance of documents is described using **presentation schemas**. These schemas allow the document designer to specify formatting, and in general, mappings from one view to another, in a convenient and general way. The language described here, henceforth referred to as **PSAL** for **Presentation Schema Assembly Language**, is general and powerful, but is neither especially convenient nor readable.

It is expected that different types of documents will require different sorts of higher level schemas. For instance, the mechanisms for describing the formatting of text documents are quite different from what would be useful for describing animation. **PSAL** is intended to be general enough to include the semantics needed for these different applications, and straightforward enough that a schema compiler can easily translate from a higher-level language to **PSAL**.

In addition to a schema compiler, it is expected that a set of *library routines* will be provided for use by the schemas. For instance, a text genre might have a linebreaking algorithm coded as a library routine. Some examples of library routines are mentioned in this paper, but facilities for managing them are not described.

In this discussion, examples are drawn primarily from the simple text genre presented in Appendix A, although none of the presentation system as it currently stands is text-specific. However, as a wider variety of document types are included in **Ensemble**, especially dynamic media, **PSAL** will probably evolve substantially from its current form.

5.1 Basic Principles

The fundamental concept used in **PSAL** is that of constraint satisfaction. Conceptually, constraints are declarations about what relations must hold between document components. For instance, "two paragraphs in a sequence must be 15 points apart" is one constraint that might be given. The constraint system sees the document as an unstructured collection of nodes that are linked by constraints among them. It has no intrinsic notion of either parent-child relationships or facet ownership. Rather, it is up to the editor and the genre-specific code to embody the intended semantics of these relationships in appropriate constraints. For example, between every facet and its children, there is a constraint that states "the parent's bounding box must contain the bounding boxes of all children", but this constraint is created as the result of an externally-specified rule.

Maintaining presentations by means of constraints has a number of important advantages. First, constraints are very convenient, since they are declarative and the document designer or the user need not concern himself with how they are satisfied, as long as they are. This places the burden on the system designers to ensure that **Ensemble-C** can maintain the system of constraints correctly and efficiently. Fortunately, there are general mechanisms for accomplishing this, which will be discussed in the sections on constraint graph maintenance and constraint solution.

Second, constraints are a powerful mechanism for expressing a wide variety of relations. For example, the animation system can provide rules for creating the equations of motion required for a particular system of objects, and allow the constraint system to select an appropriate method for solving them. This has the advantage that these constraints, which are really differential equations, can be easily mixed with time-independent constraints that tie the animation to other types of media in the document. Once we have resolved the issues of specification, the problem then becomes one of ensuring that the constraint system is able to solve these mixed systems efficiently.

Constraints have been used for graphical user interfaces in **Garnet** [Myers *et al.* 1989], physical simulation in **ThingLab** [Borning 1979], and electrical simulation [Steele 1980]. Our concept of constraints is closest to that of **ThingLab** – a constraint is like a black box that can be attached to a number of variables, and has an **error function** and a number of **satisfaction functions**. The error function tells whether the constraint is satisfied, and the satisfaction functions may be called in order to modify the arguments to satisfy the constraint.

While constraints are a mechanism for enforcing a set of relationships among the values in a system, if the structure of the system is being dynamically modified, there must be a mechanism to maintain the *structure* of the constraints. When we change the structure of our document, we will usually have to create some new constraints and remove or modify old ones. In this paper, we will refer to this process as **constraint graph maintenance**, and the process of modifying the values that are related by the constraints in such a way as to satisfy the constraints as **constraint satisfaction**.

Maintaining a presentation of a document as it is being modified thus has a number of aspects. First, we have to ensure that the structure of each view tree is consistent with the structure of its owner (that is, of the tree that represents the view it is derived from). Note that since "consistent" does not mean "identical", this is a non-trivial task.

Second, we have to ensure that the appropriate constraints are in place for a view tree. What constraints are appropriate must be expressed by means of **presentation rules**. These rules can be thought of as a predicate, which is a statement about the structure of the view tree, and a constraint that should exist for each location in the tree where the predicate is satisfied.

Third, we have to make sure the constraints are satisfied by the values in the tree. Some of the values shouldn't be modified by the system, such as user-supplied text, but others can be, such as the positions of words. Two things require constraint satisfaction to be done. One is the addition of new constraints, which might happen when a word is added to a paragraph. The other is the modification of a value that is constrained, such as the textual value of a word (and thus its bounding box).

These tasks suggest a three-stage process for presentation maintenance. First, we map any changes in the base tree to the view trees that depend on it. Second, we examine these changes and modify the constraint graph appropriately. Third, we invoke the constraint satisfaction process. Of course, in many cases we can skip the first two steps.

5.2 PSAL Descriptions

PSAL is built on top of the Common Lisp Object System (CLOS). Facets are instances of classes that inherit from a base `facet` class.

The two types of forms found in a **PSAL** description are **constraint definitions**, which use the `defconstraint` form, and **facet definitions**, which use the `deffacet` form.

The `deffacet` form contains the name of the facet, a list of the CLOS classes that this facet should inherit from, the name of the presentation schema that this facet definition belongs to, and a list of keyword-value pairs that provide additional information about the facet type. Most of the rules discussed in the rest of this paper are expressed in this way.

An example of a **PSAL** description for a text genre will be given in A. Here is a typical `deffacet` statement from this description:

```
(deffacet itemize-proof *article-proof-default* (block-proof)
  (:structure :replace-sequence-child :element-list :element
             (element-proof
              :children ((:tag (word-proof
                                :slots ((value "*"))))
                         (:item (child)))))

  (:rule (y-separation (:sequence-child (:self) :element-list)
                       (:sequence-child (:self) :element-list)
                       (:next-child (:self) :element-list) 10))

  (:owner-type itemize)
  (:documentation "Itemize environment."))
```

5.3 Structural Rules

Structural rules provide mappings from the base tree to the view tree. These rules are given in the `deffacet` forms for the particular view, since it is not feasible for the schema for the base view to know about all the views that might be derived from it.

5.3.1 The :structure Form

In some cases the base tree and the view tree are very similar. For example, a **paragraph** facet in the base tree generally corresponds to a **paragraph-proof** facet in the view tree for a **proof** view type. In such cases, when the facet is created in the base tree, the system by default creates the appropriate facet in the view tree, and sets up a link between them. The facet in the base tree then becomes the "owner" of the view facet.

In other cases, the structure of the view tree differs somewhat from that of the base tree. For instance, in our example genre, the items in an itemized list are represented as a list of blocks (that is, paragraphs, itemized lists, figures, and similar objects). In a proof view, however, each item must have a bullet. The structure of the **itemize-proof** sub-tree reflects this: instead of a list of blocks, it contains a list of "elements", each of which contains a tag and a block. The rule for creating this structure, which is contained in the **deffacet** form for **itemize-proof**, looks like this:

```
(:structure :replace-sequence-child :element-list :element
            (element-proof :children
                ((:tag (word-proof :slots ((value "*"))))
                 (:item (child)))))
```

This example illustrates the use of the `:structure` keyword. Its operands consists of: (1) an operation, in this case `:replace-sequence-child`, (2) the name of one of the children of the facet, in this case `:element-list`, (3) the new name to be used for the child, `:element`, and (4) a description of what should be done with objects that appear in the named child. The interpretation of the last part depends on what the operation is. In this case, it is a structural description of what should replace a child of the **sequence** node that is the `:element-list` child of the **itemize** facet.

The special facet type name `child` is replaced by the facet that would have been created had there been no special `:structure` rule. Thus the effect of this structure rule is to wrap each block in the list with an **element-proof** facet.

There are a number of other structural operations that can be specified.

- Insert something into the facet's child list before (or after) the named child. For instance, when we format an abstract, the simplest way to get the word “ABSTRACT” before the text is to add another child to the document facet. This is accomplished with the following rule in the **document-proof** facet definition:

```
(:structure :insert-before :abstract :abstract-word
            (word-proof :slots ((:value "ABSTRACT"))))
```

The `:insert-after` operation is similar. There are also `:insert-start` and `:insert-end` operations that place the specified objects at the beginning and end of the facet's child list.

- Replace a child of the facet with something different. There are two operations that do this: `:replace-child` and `:replace-sequence-child`. The second one actually replaces a grandchild of the facet. An example of `:replace-sequence-child` is given above for the **itemize-proof** facet.

- Sort the items in a sequence. One might want to do this in a bibliography, for instance. The declaration for this is:

```
(:structure :sorted :entries :entries
           (lambda (ent1 ent2)
              (string-lessp (bibentry-proof-tag ent1)
                            (bibentry-proof-tab ent2))))
```

  The description part of the declaration is a comparison function that is used in the sort, the arguments of which are elements of the sequence.

- Make arbitrary changes to the structure rooted at this facet. There may be arbitrarily complex rules for such operations as formatting a bibliography entry, where the logical components are the title, author, and so forth but the formated version is basically a paragraph composed of the words contained in these components.

  In order to deal with such cases, we allow an escape into Lisp. Here is the structural rule for creating bibliography entries, which is given in the `deffacet` form for **bibentry-proof**.

```
(:structure :rewrite t
            (lambda (body)
              (format-bib-entry body)))
```

This rule should be interpreted as follows. The argument after `:rewrite` specifies which children should be used as input to the rewrite process. The value `t` means that all children of the **bibentry** facet should be fed to the function given. This function takes as its argument a structural description of the requested children, for example:

```
((:author (paragraph :id g0 :children
              ((:body (sequence :children
                       ((nil (word :id g1 :slots ((value "Bob"))))
                        (nil (word :id g2 :slots ((value "Dobbs"))))
                       ))))))
 (:title (paragraph :id g3 :children
             ((:body (sequence :children
                      ((nil (word :id g4 :slots ((value "The"))))
                       (nil (word :id g5 :slots ((value "Title"))))
                      ))))))
 (:year (word :id g6 :slots ((value "1492")))))
```

Its return value should be the description of the new children that should be created:

```
((:body (paragraph-proof :children
     ((:body (graphical-seq :children
           ((nil (word-proof :slots ((value "[Dobbs 1492]"))))
            (nil (word-proof :owner g1 :slots ((value "Bob"))))
            (nil (word-proof :owner g2 :slots ((value "Dobbs"))))
            (nil (emphasis-proof :children
                ((:items (graphical-seq :children
                       ((nil (word-proof :owner g4
                                 :slots ((value "The"))))
                        (nil (word-proof :owner g5
                                 :slots ((value "Title"))))
                       ))))))))))))
 (:tag (word-proof :slots ((value "[Dobbs 1492]")))))
```

Note that the input description is in terms of the base view, and the output description is in terms of the derived view.

The details of how the `format-bib-entry` function works are not illuminating. As more experience is gained in writing presentation descriptions, the role of the `:rewrite` construct hopefully will diminish, since other constructs will be added to cover the more common cases that haven't been thought of yet.

5.3.2 Default Facet Types

Facets in the derived view are associated with ones in the base view by means of the `:owner-type` keyword. The keyword-value pair

```
(:owner-type paragraph)
```

means that this facet is in general owned by **paragraph** facets in the base view for this view type. When no `:structural` rules have control over a particular facet, the default action is to create the one that is owned by the one created in the base facet.

5.3.3 Base View Forms

Most of the above forms are useful only for derived views. In the base view, there are a number of things we need to describe that are not required for derived views.

When an object is created, sometimes it is necessary to fill in some slots in arbitrary ways. For instance, the **citation** facet has a slot to hold a reference to the **bibentry** object being cited. The `:creation` keyword is used for this purpose:

```
(:creation :initvalue citation-entry
           (lambda (facet)
             (get-facet-from-user 'bibentry)))
```

The argument `:initvalue citation-entry` means that the initial value of the slot should be obtained by evaluating the code given. In this case, it calls a routine to prompt the user for a bibliography entry to be cited.

5.4 Presentation Rules

After the new structure of the view tree has been determined, we must decide how to modify the constraint graph. In **PSAL**, the rules for doing this are stated in the form of constraint prototypes. The constraint arguments may either be constants or patterns. As an example, consider the rule that the numbers of items in successive elements of an enumerated list differ by 1. This might be expressed as a rule for an **enumerate-proof** facet:

```
(:rule (value-add (:child (:sequence-child (:self) :element-list)
                          :tag)
                  (:child (:next-child (:self) :element-list)
                          :tag)
                  1))
```

The interpretation of this is as follows. The pattern

```
(:sequence-child (:self) :element-list)
```

matches any child of the **sequence** node that is the `:element-list` child of the **enumerate-proof** facet. In this schema, this child will be a facet of type **element**, which has two children, the tag or numeric label of the item, and the text of the item. Thus the enclosing pattern, which is

```
(:child (:sequence-child (:self) :element-list) :tag),
```

matches the tag, which we would like to have printed as a number that is the index of this item in the enumerated list.

The second argument is similar, except that the `:next-child` pattern will match the *next* member of a sequence after the last one that was matched by a pattern in the current rule.

Therefore this rule will match the tags of all successive pairs of elements in the enumerated list. The semantics of this rule require a `value-add` constraint to be created between all such matching pairs, with a third argument (the difference in values) of 1. If another rule specifies that the first element of the list should have a value of 1, all the elements will be numbered correctly.

It is important to node that rules are specified as part of a facet definition, and contain only patterns that refer to descendants of that facet. There are no "free-standing" rules, or rules that can match parents or more distant ancestors. This simplifies the implementation quite a bit, since when one creates or deletes a facet, it is relatively easy to determine what other facets in the tree might have rules that are affected by this – one need merely visit all the ancestors of the affected node.

5.4.1 Patterns

All rules are introduced by the `:rule` keyword in the `deffacet` form. Patterns may be arbitrarily nested. In the following list of patterns, the form *pat* can be any pattern. Note that the only pattern that does not contain another pattern is `(:self)`. This serves to enforce the restriction that all instances of rules must be rooted at a facet of the type that the rule is being defined for.

The examples in Appendix A may be helpful in understanding the following explanation, since most of the patterns described here are used for relatively straightforward explanations.

- (**:self**). This matches a facet of the type this rule is defined for.
- (**:child** *pat child-name*). This matches the child of the node that matches *pat*, which must have the name *child-name*. For example,

  ```
  (:child (:self) :abstract)
  ```

  defined for a **document** node matches the abstract child.
- (**:first-sequence-child** *pat child-name*). This matches the first child of the sequence node that is the child of *pat* with name *child-name*. For example,

  ```
  (:first-sequence-child (:self) :author-list)
  ```

  would match the first author in the author list of a document.
- (**:last-sequence-child** *pat child-name*). Like `:first-sequence-child`, except it matches the last element of a list.
- (**:sequence-child** *pat child-name*). This matches any element of the given sequence.
- (**:next-child** *pat child-name*). This pattern allows one to refer to the results of a previous pattern matching. One and only one **:first-sequence-node** or **:sequence-node** pattern with the same *pat* and *child-name* as this one must have appeared already. The `:next-child` pattern matches the element immediately after the one that this pattern matched.
- (**:prev-child** *pat child-name*). This is like `:next-child`, except it matches the element before the previously matched one.
- (**:owner** *pat*). This matches the facet that owns the facet matched by *pat*. This facet will be in the view that the current view is derived from.

- (**:our-facet** *pat*). This is the opposite of **:owner**. It matches the facet in this view that is owned by the one matched by *pat*.

- (**:type** *pat type-name*). This matches everything that *pat* matches, if that happens to be of type *type-name*. This is useful in a few places – in particular, when a word in a proof view is owned by a corresponding word in the base view, we would like to create a constraint between them that causes their values to be the same. When it is owned by something else, as is the case for text that appears only in the presentation, we don't want to set up any such constraint. In this case we would wrap a `(:type` *pattern* `word)` around the thing we are matching.

- (**:same** *number*). This matches the same facet that was matched by the *number*'th argument to this rule. Arguments are numbered starting at 1.

- (**:all-descendants** *pat facet-type*). This matches all of the descendants of the facet matched by *pat* that are of the given type. This pattern corresponds to multiple arguments to constraints, which are described in the section on constraint satisfaction. This is a rather odd pattern, since one instance of a rule using it can match many objects simultaneously. However, it seems to be the most straightforward way to implement such useful things as linebreaking rules and figure numbering.

- (**:slot-value** *pat slot-name*). This fetches the value of the named slot, which should be a reference to a facet. This is required for citations – the following rule links the `:tag` children of a citation and its bibentry object:

```
(:rule (word-proofs-equal
         (:child (:our-facet (:slot-value (:owner (:self))
                                          entry))
                 :tag)
         (:child (:self) :tag)))
```

5.4.2 Variable bindings

One other form may appear in rules, the `:binding` form. This looks like a pattern, but it actually a simple mechanism for accessing dynamically scoped variables.

In a `deffacet` form, a variable binding such as

```
(:bind :font-family "Times")
```

or

```
(:bind (:font-type :title) "Bold")
```

may appear. This causes the named variable to be bound to the given value for all descendants of the facet, or, in the second case, only descendants of the child named `:title`. The values must either be constants or lambda-expressions that compute the actual value from other bindings available at that facet. An example of the second sort of binding is

```
(:bind (:width :title) (lambda (doc)
                          (let ((width (get-binding doc :width)))
                            (if width (- width 100) 200)))
```

The interpretation of this is as follows. The `doc` argument corresponds to the **document-proof** facet this rule appears in. The `get-binding` function returns the value of the variable at this node in the tree, or `nil` if there is no such binding. The function then returns either the current width minus 100, if there is a current width, or 200, if there is none. Note that in this example there will only be a width defined for the document from above if it is nested inside of another object of a possibly different genre.

A presentation rule may access the bindings visible at the current facet using the form `(:binding` *var-name*`)`. In the title of a document, for instance, the form `(:binding :font-type)` might evaluate to `"Bold"`.

5.5 Constraint Definitions

A constraint can be thought of as a device that can be connected to a set of values and will enforce a relationship between them. A constraint definition must therefore contain a description of the values it may be connected to, and a statement of the relationship to be enforced. This statement may be characterized further as an error function, which takes as arguments the values of the constraint arguments and returns an indication of whether the constraint is satisfied, and a number of satisfaction functions, which can be applied to the arguments of the constraint to cause the constraint to be satisfied.

The constraint system is built on top of CLOS, and the arguments to a constraint are instances of classes. Since in general the values of interest will be bound to sub-parts of an instance, one may specify a *path* for an argument. For instance, if one argument to a constraint is the height of a paragraph, the path might be given as `(bbox yhi)`, where the paragraph object has a slot called `bbox`, which is bound to a `bounding-box` object which itself has a slot called `yhi`.

There are two types of error functions – those that return a boolean value, and those that return some numerical indication of how far the constraint is from being satisfied. The second type is useful for cases where relaxation is used to solve cyclical systems. In general, the error function may give little information about how to modify the arguments to the constraint in order to satisfy it, but in many cases this information can be inferred.

Satisfaction functions have three parts. These are the set of "input" variables, the set of "output" variables, and the body of the function that maps from the input to the output. In many cases, only a few combinations of input and output variables make sense. For instance, in a constraint that enforces line-breaking, the only argument that may be permissibly changed is the list of positions of the words.

Here is an example of a `defconstraint` form.

```
(defconstraint y-separation
  ((pos1    graphical :path (pos y))
   (height1 graphical :path (bbox yhi))
   (pos2    graphical :path (pos y))
   (skip    constant  :path value))

  (lambda (pos1 pos2 height1 skip)
    (- (+ (+ pos1 height1) skip) pos2))

  (((pos1 height1 skip)
    (pos2)
    ((+ skip (+ pos1 height1))))))
```

There are really only three arguments, but since we want to use two of the components of the first object, we repeat it twice. `graphical` is a class that includes all objects that have positions and bounding boxes – in almost all cases the bounding box extends from (0, 0) to (*xhi*, *yhi*), and to get the bounding box with relation to the object's parent, we must translate it by its position.

The error function is a lambda-form that takes a subset of the arguments (in this case, all of them) and returns 0 if the constraint is satisfied and something other than 0 otherwise. The names in the lambda form must match the names given to the arguments.

The satisfaction functions are given in a list of (*inputs*, *outputs*, *function*) tuples. If there is more than one output, the function should return multiple values using the Common Lisp `values` construction. In this case, there is only one satisfaction function because there is only one way to satisfy this constraint that we want to allow.

Here is another constraint definition, which illustrates some other things.

```
(defconstraint bbox-union
  ((container graphical :path bbox)
   (contents  graphical :path bbox yhi :multiple))

  (lambda () nil)

  (((contents)
    (container)
    ((calculate-bbox-union contents)))))
```

The `:multiple` keyword given for the contents argument indicates that instead of one object being constrained, we wish to constrain a whole set of objects. In the error function and the satisfaction functions, whenever the argument appears, it will be bound to a list of the actual values. In this case, the function `calculate-bbox-union` takes a list of bounding boxes as its argument and returns a single bounding box.

The error function above always returns `nil`. Although this would seem to pose a problem for the constraint solver, in fact the error function is only checked when one of the values constrained has changed. Since we have to calculate the enclosing bounding box anyway to check whether it is still the correct one, it is simpler to always induce the constraint to satisfy itself. The solver is careful not to mark values as changed when they are set to the same value they had before, so this technique does not cause extra computation to occur.

5.6 Constraint satisfaction algorithms

There are a number of constraint satisfaction algorithms that have been developed in the last decade. Some rely on term rewriting, such as the system described in [Leler 1988], and others are tied to logic programming, such as [Saraswat 1989] and [Jaffar and Lassez 1987]. For incremental constraint satisfaction, which is what **Ensemble-C** requires, the most convenient techniques are based on *propagation of known values* and *relaxation*.

Propagation of known values is done by starting with the constraints we know we can satisfy, and using them to determine values for further variables, which allows us to satisfy more constraints. This process stops either when all constraints are satisfied or when none of the constraints that are unsatisfied can be solved. In the latter case, we must use relaxation to solve the rest of the system.

Relaxation can be used to solve cyclic systems. If all the constraints are numeric and differentiable, we can use a simple iterated Newton-Raphson approximation method to find the solutions. For other sorts of constraints, we must repeatedly solve them in sequence, each time using the values obtained in the previous iteration. This is rather unreliable and slow, and care should be taken to avoid cycles in the constraint graph if possible.

Ensemble-C uses these two techniques for constraint satisfaction. A more detailed description of constraint satisfaction can be found in [Leler 1988].

6 Discussion

Ensemble-C is a prototype system, and its primary role has been that of a testbed for new ideas, rather than a reliable and efficient document system. A number of experiments remain to be done in order to show that the general approach of the system is appropriate for real-world document preparation.

At this point, it is only possible to create and delete leaf nodes, and the type of a facet cannot be changed once it is created. This restriction makes a number of common and useful operations rather expensive and awkward, such as moving sections of a document, and placing a set of existing objects inside a new environment. In order to support these operations we need to be able to add and facets into the middle of a tree and move entire subtrees without deleting and rebuilding them from scratch. The algorithms used for structural rules and constraint graph maintenance are quite simple-minded, and would need to be rewritten to handle these cases. The **Colander** system [Ballance 1989] has the facilities for dealing with general tree transformations and truth maintenance, and we may use it for presentation maintenance in the future.

The current version of **ThingLab** [Freeman-Benson and Maloney 1988, Borning *et al.* 1987] can solve *hierarchical* constraint systems, where some constraints are required, and the rest are placed in a hierarchy that determines the order in which they are solved. In general, not all constraints need be satisfied. This strategy allows much more complex systems to be described and solved. In particular, if **Ensemble-C** could solve hierarchical constraints, the management of document layout would be simplified quite a bit. For example, the rule that the bounding box of an object is the union of the bounding boxes of its children could be made into an optional rule, and this would allow us to constrain the width of paragraphs to be their maximum width as determined by the page layout rules, rather than using dynamically-scoped `:width` variables, which is rather awkward.

Another refinement of the constraint solution process involves priority. Whereas hierarchy among constraints makes a statement about the final solution, priority makes statements about which constraints should be solved before others, even if they will later be overridden. This is important for an interactive system, where we may want a "quick and dirty" solution before we have the final solution. Also, priority can be used to cause views to be updated in the order in which we are interested in looking at them – we want the view we are typing in to be dealt with before any others.

If we are to use the same constraint specification mechanism for static documents, for which local propagation is probably the best constraint solution, and dynamic documents, for which numerical solution techniques are usually required, the constraint system must be able to tell the two types of graphs apart. Some work has been done on identifying subgraphs that can be solved using different techniques [Gosling 1983], and we will require mechanisms that are similar to this.

As a test of the generality of the **Ensemble-C** framework, we would like to create an object graphics view with interactive constraint specification, such as that in **ThingLab**. Many of the difficult issues in text document formatting are not problems for object graphics, since the structure and presentation rules are generally much simpler.

An example of an editing session using the current **Ensemble-C** prototype is shown in Figure 3. The prototype is written entirely in Common Lisp, and has not been optimized for speed, but all of the constraint graph maintenance and constraint satisfaction mechanisms described in this paper have been implemented.

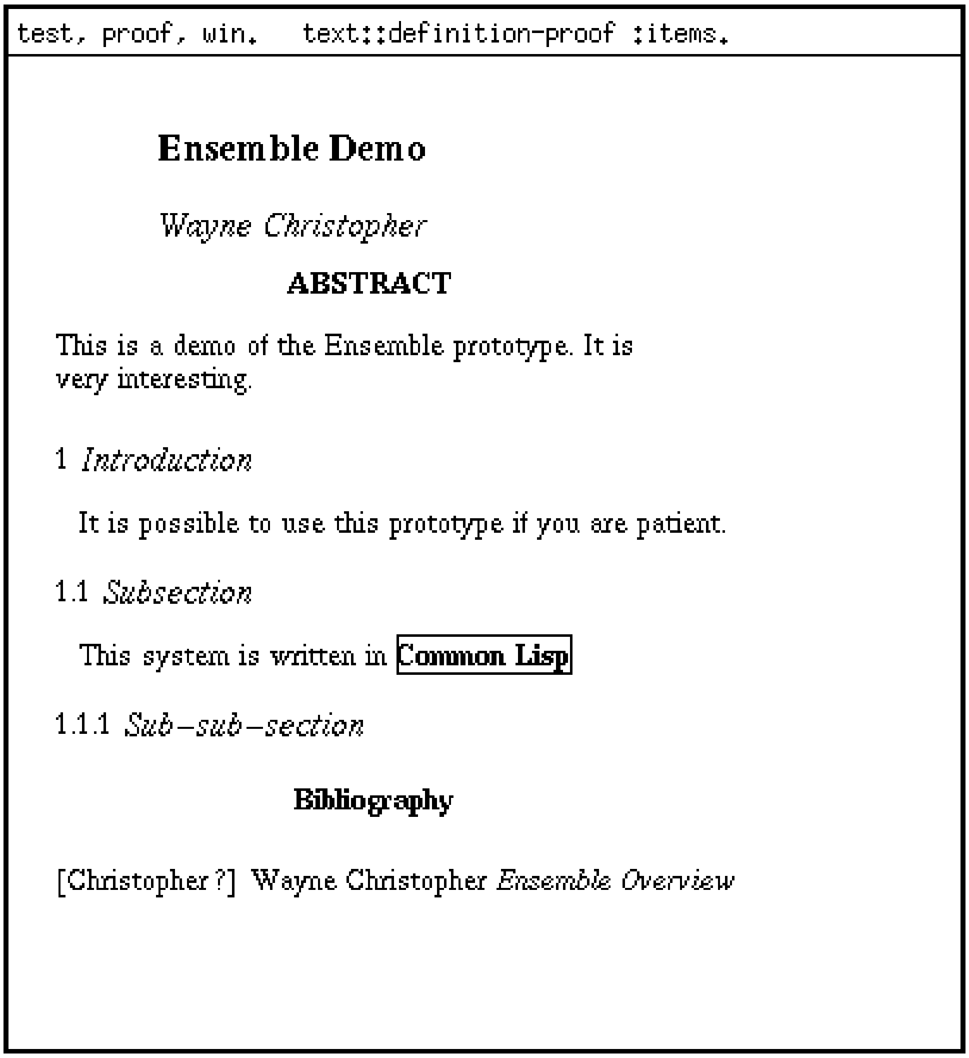

Figure 3: The Ensemble-C Prototype

References

[Ballance and Van De Vanter 1987] Robert A. Ballance and Michael L. Van De Vanter. Pan I: An introduction for users. Technical Report UCB/CSD 88/410, University of California, Berkeley, CA 94720, September 1987. PIPER Working Paper 87-5.

[Ballance *et al.* 1987] Robert A. Ballance, Michael L. Van De Vanter, and Susan L. Graham. The architecture of Pan I. Technical Report UCB/CSD 88/409, University of California, Berkeley, CA 94720, August 1987. PIPER Working Paper 87-4.

[Ballance *et al.* 1990] Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter. The Pan language-based editing system for integrated development environments. In *ACM SIGSOFT Symposium on Software Development Environments*, 1990.

[Ballance 1989] Robert A. Ballance. *Syntactic and Semantic Checking in Language-Based Editing Systems.* PhD thesis, University of California, Berkeley, CA 94720, December 1989. Technical Report UCB/CSD 89/548.

[Borning *et al.* 1987] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. In *Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1987, pages 48–60.

[Borning 1979] Alan Borning. *Thinglab — A Constraint-Oriented Simulation Laboratory.* PhD thesis, Stanford University, July 1979. Technical Report STAN-CS-79-746, also Xerox SSL-79-3.

[Brooks 1988] Kenneth P. Brooks. *A Two-view Document Editor with User-definable Document Structure.* PhD thesis, Stanford University, May 1988. Also Digital SRC Research Report 33, November 1, 1988.

[Chamberlin *et al.* 1987] D. D. Chamberlin, H. F. Hasselmeier, A. W. Luniewski, D. P. Paris, B. W. Wade, and M. L. Zolliker. Quill: An extensible system for editing documents of mixed type. In *Proceedings of the 21st Hawaii International Conference on System Sciences*, Washington, D.C., 1987. IEEE Computer Science Press.

[Chen 1988] Peehong Chen. *A Multiple-Representation Paradigm for Document Development*. PhD thesis, University of California, Berkeley, CA 94720, July 1988. Also TR UCB/CSD 88/436.

[Freeman-Benson and Maloney 1988] Bjorn N. Freeman-Benson and John Maloney. The deltablue algorithm: An incremental constraint hierarchy solver. Technical Report 88-11-09, University of Washington, Seattle, WA 98195, November 1988.

[Gosling 1983] James Gosling. *Algebraic Constraints*. PhD thesis, Carnegie-Mellon University, Pittsburgh PA 15213, May 1983. TR CS-83-132.

[Harrison and Munson 1990] Michael A. Harrison and Ethan V. Munson. Numbering document components. Technical Report 90/568, University of California, Berkeley, CA 94720, June 1990.

[Jaffar and Lassez 1987] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *ACM Symposium on Principles of Programming Languages*, 1987, pages 111–119.

[Lamport 1986] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, 1986.

[Leler 1988] Wm Leler. *Constraint Progamming Languages*. Addison-Wesley, 1988.

[Munson and Pan 1990] Ethan V. Munson and Derluen Pan. Selections and snapshots in ensemble. Ensemble Working Paper, May 1990.

[Myers *et al.* 1989] Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, Ed Pervin, and John A. Kolojejchick. The Garnet toolkit reference manual: Support for highly-interactive, graphical user interfaces in lisp. Technical Report CMU-CS-89-196, Carnegie Mellon University, Pittsburg, PA 15213, November 1989.

[Quint and Vatton 1986] V. Quint and I. Vatton. Grif: An interactive system for structured document manipulation. In J. C. van Vliet, editor, *International Conference on Text Processing and Document Manipulation*. Cambridge University Press, 1986, pages 200–213.

[Reid 1980] Brian K. Reid. *Scribe: A document specification language and its compiler.* PhD thesis, Carnegie-Mellon University, October 1980. Also TR CMU-CS-81-100.

[Saraswat 1989] Vijay Anand Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, January 1989. Technical Report CMU-CS-89-108.

[Stallman 1986] Richard M. Stallman. *GNU Emacs Manual.* Free Software Foundation, Cambridge, MA, fourth edition, February 1986.

[Steele 1980] Guy Lewis Steele, Jr. *The Definition and Implementation of a Computer Programming Language.* PhD thesis, Massachusetts Institute of Technology, Cambridge MA 02139, August 1980. Also MIT AI Lab TR 595.

A Examples

The following are examples of structural and presentation schemas for a text document genre.

The first file defines a simple grammar for an article style. Note that the grammar rule for the `section` facet type is recursive – a section contains a title, some text, and then a list of sections. Since the minimum number of subsections is 0, the facet creation code won't get into an infinite loop. This type of document structure is described more fully in [Harrison and Munson 1990].

A.1 The Text Module

```
;;; Set up the package stuff.  Note that we don't export anything.
;;; The require's and use-package's are omitted for brevity.

(provide "text")
(in-package "text")

;;; Make a genre object.  This is mainly a placeholder for
;;; structural schemas.

(defvar *text-genre*
  (make-genre "text"))

;;; Make a structural schema.  This is a simple article style.

(defvar *article*
  (make-grammar *text-genre* "article"
                '((document
                   :children ((:title block)
                              (:author-list block 1 *)
                              (:abstract block)
                              (:section-list section 1 *)
                              (:bibliography bibliography)))

                  (section
                   :children ((:title block)
                              (:blocks block 0 *)
                              (:sections section 0 *)))
```

```
                  (bibliography
                   :children ((:entries bibentry 1 *)))

                  (bibentry
                   :children ((:author paragraph)
                              (:title paragraph)
                              (:journal paragraph)
                              (:month word)
                              (:year word)
                              (:pages word 0 2)))

                  (block -> paragraph itemize enumerate figure foreign)

                  (paragraph
                   :children ((:items item 1 *)))

                  (itemize
                   :children ((:element-list block 1 *)))

                  (enumerate
                   :children ((:element-list block 1 *)))

                  (figure
                   :children ((:stuff foreign)))

                  (item -> word emphasis definition citation foreign)

                  (emphasis
                   :children ((:items item 1 *)))

                  (definition
                    :children ((:items item 1 *)))

                  (citation
                   :children ((:tag word)))

                  (word)

                  (foreign))))

;;; Define the facets we use for the text base view.

(defclass text-base (facet)
  ()
```

```
  (:documentation "All text facets inherit from this."))

(deffacet document nil (text-base)
  (:documentation "The top-level text thing."))

(deffacet section nil (text-base)
  (:documentation "A section.  Sections can be arbitrarily nested."))

(deffacet bibliography nil (text-base)
  (:documentation "A bibliography contains bibentries."))

(deffacet bibentry nil (text-base)
  (:documentation "A bibliography entry."))

(deffacet paragraph nil (text-base)
  (:documentation "A paragraph."))

(deffacet itemize nil (text-base)
  (:documentation "An itemized list."))

(deffacet enumerate nil (text-base)
  (:documentation "An enumerated list."))

(deffacet figure nil (text-base)
  (:documentation "A figure environment."))

(deffacet emphasis nil (text-base)
  (:documentation "An emphasis environment."))

(deffacet definition nil (text-base)
  (:documentation "A definition font environment."))

(deffacet citation nil (text-base)
  (:slots ((entry :type bibentry
                  :accessor citation-entry
                  :documentation "The bibliography entry we're citing.")))

  (:creation :initvalue citation-entry
             (lambda (facet)
               (get-facet-from-user "what this citation should point to"
                                    'bibentry facet)))
  (:documentation "A citation, which points to a bibliography entry."))

(deffacet word nil (text-base)
```

```
  (:slots ((value :type string
                  :initarg :value
                  :accessor word-value
                  :documentation "The string that is the word.")))

  (:print-method (lambda (word stream)
                   (format stream "#<word ~S ~S>" (facet-id word)
                           (if (slot-boundp word 'value)
                             (word-value word)))))

  (:documentation "A word."))
```

A.2 The Text Proof Module

```
;;; This stuff is also in the text package.  In lots of places there
;;; should be package specifiers but I've taken them out for clarity.
;;;
;;; Rules for maintaining bounding boxes have been temporarily moved
;;; the presentation system for simplicity.  They really belong here,
;;; though.

(provide "text-proof")
(in-package "text")

;;; Make a presentation schema to hang all the facet stuff off of.

(defvar *article-proof-default*
  (make-pschema *article* "article-proof-default"))

;;; First let's define some classes that don't actually correspond
;;; to facet types.

(defclass text-proof (graphical)
  ()
  (:documentation "All text proof facets inherit from this."))

(defclass block-proof (text-proof)
  ()
  (:documentation "A block-like thing."))

(defclass item-proof (text-proof)
  ()
  (:documentation "An item-like thing."))
```

```
;;; Now define the facets.

(deffacet document-proof *article-proof-default* (text-proof)

  ;; We have to add the word "ABSTRACT" to the document, before
  ;; the abstract paragraph.
  (:structure :insert-before :abstract :abstract-word
              (word-proof
               :slots ((value "ABSTRACT"))))

  (:structure :insert-before :bibliography :bibliography-word
              (word-proof :slots ((value "Bibliography"))))

  ;; First define some rules to maintain the spacing of the
  ;; children of this node.  See the constraint definitions
  ;; at the bottom of this file for a description of the arguments.

  (:rule (at-y-pos     (:child (:self) :title)         20)

         (y-separation (:child (:self) :title)
                       (:same 1)
                       (:child (:self) :author-list)   10)

         (y-separation (:child (:self) :author-list)
                       (:same 1)
                       (:child (:self) :abstract-word) 10)

         (y-separation (:child (:self) :abstract-word)
                       (:same 1)
                       (:child (:self) :abstract)      5)

         (y-separation (:child (:self) :abstract)
                       (:same 1)
                       (:child (:self) :section-list)  10)

         (y-separation (:child (:self) :section-list)
                       (:same 1)
                       (:child (:self) :bibliography-word)  10)

         (y-separation (:child (:self) :bibliography-word)
                       (:same 1)
                       (:child (:self) :bibliography)  10)
```

```
       (at-y-pos     (:first-sequence-child (:self) :section-list) 0)

       (y-separation (:sequence-child (:self) :section-list)
                     (:same 1)
                     (:next-child (:self) :section-list) 5)

       (at-x-pos     (:child (:self) :title)         20)
       (at-x-pos     (:child (:self) :author-list)   20)
       (at-x-pos     (:child (:self) :abstract)      20)
       (at-x-pos     (:child (:self) :section-list)  20)
       (at-x-pos     (:child (:self) :bibliography)  20)

       (x-centered   (:child (:self) :abstract-word)
                     (:same 1)
                     (:self))

       (x-centered   (:child (:self) :bibliography-word)
                     (:same 1)
                     (:self)))

;; Maintain section numbers.  These rules are duplicated in the
;; section facet definition for subsections.

(:rule (at-section-number (:first-sequence-child (:self)
                                                  :section-list) 1)

       (add-section-number (:sequence-child (:self) :section-list)
                           (:next-child (:self) :section-list) 1))


;; Bind a bunch of variables used for fonts and linebreaking.

(:bind :font-family "Times"

       (:font-type :title)         "Bold"
       (:font-type :author-list)   "Italic"
       (:font-type :abstract-word) "Bold"
       (:font-type :abstract)      "Roman"
       (:font-type :section-list)  "Roman"
       (:font-type :bibliography-word) "Bold"
       (:font-type :bibliography)      "Roman"

       (:font-size :title)         16
       (:font-size :author-list)   14
```

```
       (:font-size :abstract-word) 12
       (:font-size :abstract)      12
       (:font-size :section-list)  12
       (:font-size :bibliography-word)  12
       (:font-size :bibliography-word)  12

       (:indent :title)        0
       (:indent :author-list)  0
       (:indent :abstract)     0
       (:indent :section-list) 10
       (:indent :bibliography) 0

       :interword-space 6

       (:baseline-skip :title)        18
       (:baseline-skip :author-list)  16
       (:baseline-skip :abstract)     14
       (:baseline-skip :section-list) 14
       (:baseline-skip :bibliography) 14

       (:fill-format :title)        :centered
       (:fill-format :author-list)  :centered
       (:fill-format :abstract)     :flush-right
       (:fill-format :section-list) :flush-right
       (:fill-format :bibliography) :flush-right)

;; Finally, deal with widths.  These are all computed bindings.
;; This is not the best way to do it -- we should have hierarchical
;; constraints.

(:bind (:width :title)        (lambda (doc)
                                (let ((width (get-binding doc :width)))
                                  (if width (- width 100) 200)))
       (:width :author-list)  (lambda (doc)
                                (let ((width (get-binding doc :width)))
                                  (if width (- width 100) 200)))
       (:width :abstract)     (lambda (doc)
                                (let ((width (get-binding doc :width)))
                                  (if width (- width 50) 250)))
       (:width :section-list) (lambda (doc)
                                (let ((width (get-binding doc :width)))
                                  (if width width 300))))

(:owner-type document)
```

```
  (:documentation "The top-level text thing."))

(deffacet section-proof *article-proof-default* (text-proof)
  (:slots ((number :accessor section-proof-number
                   :documentation "This is a list of numbers.")))

  (:structure :insert-before :title :section-number
            (word-proof))

  (:rule (at-y-pos (:child (:self) :section-number)      10)
         (at-y-pos (:child (:self) :title)              0)

         (x-separation (:child (:self) :section-number)
                       (:same 1)
                       (:child (:self) :title)          5)

         (y-separation (:child (:self) :title)
                       (:same 1)
                       (:child (:self) :blocks)           5)

         (at-y-pos (:first-sequence-child (:self) :blocks) 0)

         (y-separation (:sequence-child (:self) :blocks)
                       (:same 1)
                       (:next-child (:self) :blocks)     5)

         (at-relative-section-number (:first-sequence-child (:self)
                                                             :sections)
                                     (:self)            1)

         (y-separation (:child (:self) :blocks)
                       (:same 1)
                       (:child (:self) :sections)           5)

         (at-y-pos (:first-sequence-child (:self) :sections) 0)

         (y-separation (:sequence-child (:self) :sections)
                       (:same 1)
                       (:next-child (:self) :sections)      5)

         (add-section-number (:sequence-child (:self) :sections)
                             (:next-child (:self) :sections) 1)

         (format-section-number (:self) (:child (:self) :section-number))
```

```
        (number-figures (:self) (:all-descendants (:self) figure-proof)))

  (:bind (:font-type   :title) "Italic"
         (:font-size   :title) 14
         (:fill-format :title) :ragged-right
         (:width       :title) (lambda (sec)
                                 (- (get-binding sec :width) 100))
         (:indent      :title) 0)

  (:owner-type section)
  (:documentation "A section."))

(deffacet bibliography-proof *article-proof-default* (text-proof)
  (:structure :sorted :entries
              (lambda (ent1 ent2)
(string-lessp (bibentry-proof-tag ent1)
      (bibentry-proof-tab ent2))))

  (:rule (at-y-pos (:first-sequence-child (:self) :entries) 0)

         (y-separation (:sequence-child (:self) :entries)
                       (:same 1)
                       (:next-child (:self) :entries)       10))

  (:owner-type bibliography)
  (:documentation "A bibliography."))

(deffacet bibentry-proof *article-proof-default* (text-proof)
  (:slots ((tag :accessor bibentry-proof-tag
                :documentation "The tag string for this entry.")))

  ;; :rewrite rules are passed a description, in structural schema format,
  ;; of the stuff below here, and should pass back another structural
  ;; schema that says what to make the thing.

  (:structure :rewrite t t
              (lambda (body)
                "This rule takes a bibliography entry and formats it."

                (format-bib-entry body)))

  (:rule (at-y-pos (:child (:self) :tag)  10)
         (at-y-pos (:child (:self) :body) 0)
```

```
        (at-x-pos (:child (:self) :tag)  0)
        (x-separation (:child (:self) :tag)
                      (:same 1)
                      (:child (:self) :body) 5))

  (:owner-type bibentry)
  (:documentation "A bibliography entry."))

;; The major thing that paragraphs do is linebreaking.  Note the use of
;; :all-descendants.

(deffacet paragraph-proof *article-proof-default* (block-proof)
  (:rule (linebreaking (:self)
       (:binding :width)
                      (:binding :indent)
                      (:binding :baseline-skip)
                      (:binding :fill-format)
                      (:all-descendants (:self) word-proof)
                      (:same 6)))

  (:owner-type paragraph)
  (:documentation "A paragraph."))

;;; These two are interesting because they have extra children.
;;; The abstract has an extra child too, but it's not a separate
;;; facet type.  The itemize-proof facet has a list of elements,
;;; each of which contain a tag and a body.

(deffacet itemize-proof *article-proof-default* (block-proof)
  (:structure :replace-sequence-child :element-list :element
            (element-proof
             :children ((:tag (word-proof
                                 :slots ((value "*"))))
                         (:item (child)))))

  (:rule (y-separation (:sequence-child (:self) :element-list)
                      (:sequence-child (:self) :element-list)
                      (:next-child (:self) :element-list) 10))

  (:owner-type itemize)
  (:documentation "Itemize environment."))

(deffacet enumerate-proof *article-proof-default* (block-proof)
  (:structure :replace-sequence-child :element-list :element
```

```
            (element-proof
             :children ((:tag (word-proof))
                        (:item child))))

  (:rule (y-separation (:sequence-child (:self) :element-list)
                       (:same 1)
                       (:next-child (:self) :element-list) 10)

         (value-set (:child (:first-sequence-child (:self) :element-list)
                           :tag) 1)

         (value-add (:child (:sequence-child (:self) :element-list) :tag)
                    (:child (:next-child (:self) :element-list) :tag) 1))

  (:owner-type enumerate)
  (:documentation "Enumerate environment."))

(deffacet element-proof *article-proof-default* (text-proof)
  (:rule (x-separation (:child :tag) (:child :element) 20))

  (:documentation "An item with its tag."))

(deffacet figure-proof *article-proof-default* (block-proof)
  (:slots ((number :accessor figure-proof-number
                   :documentation "This is a number.")))

  (:structure :insert-before :stuff :figure-number
            (word-proof))

  (:rule (at-y-pos (:child (:self) :stuff)             0)

         (y-separation (:child (:self) :stuff)
                       (:same 1)
                       (:child (:self) :figure-number) 10)

         (format-figure-number (:self) (:child (:self) :figure-number))))

(deffacet emphasis-proof *article-proof-default* (item-proof)
  (:bind :font-type "Italic")
  (:owner-type emphasis)
  (:documentation "Emphasis font environment."))

(deffacet definition-proof *article-proof-default* (item-proof)
  (:bind :font-type "Bold")
```

```
  (:owner-type definition)
  (:documentation "Definition font environment."))

(deffacet citation-proof *article-proof-default* (item-proof)
  (:rule (citation-text-equals (:child (:slot-value (:owner (:self))
                                                     citation-entry) :tag)
                               (:child (:self) :tag)))
  (:owner-type citation)
  (:documentation "Citation."))

(deffacet word-proof *article-proof-default* (item-proof)
  (:slots ((value :type string
                  :initform ""
                  :accessor word-proof-value
                  :documentation "What it says.")
           (font :accessor word-proof-font
                 :documentation "This word's font.")))

  (:rule (word-bbox (:self) (:self) (:self))
         (is-font (:self) (:binding :font-family) (:binding :font-type)
                  (:binding :font-size))
         (word-value-equals (:type (:owner (:self)) word) (:self)))

  (:property :is-visible t)
  (:owner-type word)
  (:print-method (lambda (word stream)
                   (format stream "#<word-proof ~S ~S>" (facet-id word)
                           (if (slot-boundp word 'value)
                             (word-proof-value word)))))

  (:documentation "A word."))

;;; ================================================================
;;; Now we define all the constraints that are mentioned in
;;; the above rules.

(defconstraint y-centered
  ((in-pos graphical     :path (graphical::pos geometry::y))
   (in-height graphical  :path (graphical::bbox geometry::yhi))
   (out-height graphical :path (graphical::bbox geometry::yhi)))

  (lambda (in-pos in-height out-height)
    (- in-pos (/ (- out-height in-height) 2)))
```

```
  (((in-height out-height) (in-pos)
    ((/ (- out-height in-height) 2)))))

(defconstraint at-y-pos
  ((facet    graphical :path (graphical::pos geometry::y))
   (posvalue constant  :path constraint::value))

  (lambda (posvalue facet) (equal posvalue facet))

  nil)

(defconstraint y-separation
  ((first-pos graphical    :path (graphical::pos geometry::y))
   (first-height graphical :path (graphical::bbox geometry::yhi))
   (second-pos graphical   :path (graphical::pos geometry::y))
   (skip constant          :path constraint::value))

  (lambda (first-pos second-pos first-height skip)
    (- (+ (+ first-pos first-height) skip) second-pos))

  (((first-pos first-height skip) (second-pos) ((+ skip
                                                    (+ first-pos
                                                       first-height))))))

;;; -----

(defconstraint x-centered
  ((in-pos graphical    :path (graphical::pos geometry::x))
   (in-width graphical  :path (graphical::bbox geometry::xhi))
   (out-width graphical :path (graphical::bbox geometry::xhi)))

  (lambda (in-pos in-width out-width)
    (- in-pos (/ (- out-width in-width) 2)))

  (((in-width out-width) (in-pos)
    ((/ (- out-width in-width) 2)))))

(defconstraint at-x-pos
  ((facet    graphical :path (graphical::pos geometry::x))
   (posvalue constant  :path constraint::value))

  (lambda (posvalue facet) (equal posvalue facet))

  nil)
```

```
(defconstraint x-separation
  ((first-pos graphical    :path (graphical::pos geometry::x))
   (first-height graphical :path (graphical::bbox geometry::xhi))
   (second-pos graphical   :path (graphical::pos geometry::x))
   (skip constant          :path constraint::value))

  (lambda (first-pos second-pos first-height skip)
    (- (+ (+ first-pos first-height) skip) second-pos))

  (((first-pos first-height skip) (second-pos) ((+ skip
                                                  (+ first-pos
                                                     first-height))))))

;;; -----

(defconstraint linebreaking
  ((para-width     paragraph-proof :path graphical::bbox)
   (width-hack     constant        :path constraint::value)
   (initial-indent constant        :path constraint::value)
   (baseline-skip  constant        :path constraint::value)
   (format         constant        :path constraint::value)
   (word-bbox      word-proof      :path graphical::bbox :multiple)
   (word-pos       word-proof      :path graphical::pos  :multiple))

  (lambda () nil)

  ((t (word-pos) ((linebreak-words width-hack initial-indent baseline-skip
                                   format word-bbox)))))

;;; -----

(defconstraint value-add
  ((first  word-proof :path value)
   (second word-proof :path value)
   (incr   constant   :path constraint::value))

  (lambda (first second incr)
    (let ((fval (read-from-string first))
          (sval (read-from-string second)))
      (equal (+ first incr) second)))

  (((first incr) (second) ((format nil "~d" (+ (read-from-string first)
                                               incr))))))
```

```
(defconstraint value-set
  ((word  word-proof :path value)
   (value constant   :path value))

  (lambda (word value) (equal word value))

  nil)

;;; -----

(defconstraint at-section-number
  ((section section-proof :path number)
   (num     constant      :path constraint::value))

  (lambda () nil)

  (((num) (section) ((list num)))))

(defconstraint at-relative-section-number
  ((section section-proof :path number)
   (parent  section-proof :path number)
   (num     constant      :path constraint::value))

  (lambda () nil)

  (((parent num) (section) ((append parent (list num))))))

(defconstraint add-section-number
  ((prev section-proof :path number)
   (this section-proof :path number)
   (incr constant      :path constraint::value))

  (lambda () nil)

  (((prev incr) (this) ((append (butlast prev)
                                (list (+ (car (last prev)) incr)))))))

(defconstraint format-section-number
  ((number section-proof :path number)
   (word   word-proof    :path value))

  (lambda () nil)
```

```
  (((number) (word) ((do ((str (format nil "~d" (car number)))
                           (nums (cdr number) (cdr nums)))
                          ((null nums) str)
                        (setf str (format nil "~a.~d" str (car nums))))))))

(defconstraint number-figures
  ((section section-proof :path number)
   (figures figure-proof  :path number :multiple))

  (lambda () nil)

  ((t (figures) ((do* ((num 1 (+ num 1))
                        (figs figures (cdr figs))
                        (ret (list (append section (list num)))
                             (cons (append section (list num)) ret)))
                      ((null figs) (nreverse ret)))))))

;;; -----

(defconstraint at-figure-number
  ((figure figure-proof :path number)
   (num    constant     :path constraint::value))

  (lambda (figure num) (equal figure num))

  nil)

(defconstraint format-figure-number
  ((number  figure-proof  :path number)
   (section section-proof :path number)
   (word    word-proof    :path value))

  (lambda () nil)

  (((section number) (word) ((do ((str (format nil "~d" (car section)))
                                   (nums (cdr section) (cdr nums)))
                                  ((null nums)
  (format nil "~a.~d" str number))
                                (setf str (format nil "~a.~d" str
                                                  (car nums))))))))

;;; -----

(defconstraint word-proofs-equal
```

```
  ((src word-proof :path value)
   (dst word-proof :path value))

  (lambda (src dst) (equal src dst))

  (((src) (dst) (src))))

;;; -----

(defconstraint is-font
  ((word   word-proof :path font)
   (family constant   :path constraint::value)
   (type   constant   :path constraint::value)
   (size   constant   :path constraint::value))

  (lambda () nil)

  (((family type size) (word) ((get-font family type size)))))

(defconstraint word-bbox
  ((string word-proof :path value)
   (font   word-proof :path font)
   (bbox   word-proof :path graphical::bbox))

  (lambda () nil)

  (((string font) (bbox) ((window:get-string-bbox string font)))))

(defconstraint word-value-equals
  ((node word        :path value)
   (facet word-proof :path value))

  (lambda (node facet) (equal node facet))

  (((node) (facet) (node))))
```