Table of Contents

1 Introduction

Cost++[4][3][7] is a semi-automated tool for generating hypertext structures (hypertexts) for the Kiosk[4][3] system. Cost++ structures *workproducts* to make them easier to find and understand when used with the Kiosk browsing, querying, and editing facilities. Workproducts are all the products of work created in performing some task. They can include any text file (e.g., code, e-mail, paper reviews, tests, build files). Structuring is performed by building a hypertext representation that overlays these workproducts to help in their classification, clustering, and cataloging. Kiosk then uses this representation to present different ways of finding and viewing this information. Because of the close relationship between Cost++ and Kiosk, it is important to have a fairly detailed understanding of the Kiosk system. A quick overview of Kiosk will be given here; however, it is recommended that you read the documentation in [3][4] before proceeding.

Kiosk is a system built to improve the finding and understanding of technical information. At its heart, is a *Unix*-based, open hypertext system that contains powerful methods to:

- search based on the content or structure of information,
- navigate through information aided by sophisticated filtering, and
- build abstract collections of information that better suit user needs.

Kiosk only works when the technical information of interest is structured and represented in a hypertext form. For very small hypertexts, Kiosk itself can be used to manually edit nodes and links to produce the hypertext. For larger hypertexts, manual entry becomes impractical. For example, consider building a hypertext structure for a library of reusable components. It would be very time-consuming and error-prone to hand build library classification lattices, hand link all library workproducts, and then repeat this for each release of the library. To solve this problem, the Cost++ tool was created. To generate or modify a hypertext, Cost++ uses the declarative language instructions within *config files.* These instructions specify:

- Exactly which workproducts should be linked together and where they should be persistently saved.
- Creation of structure nodes to help classify, cluster, and catalog workproducts.
- *Feature Extraction*—the ability to find interesting features within workproducts and link them to interesting features in other workproducts or structure nodes.
- New terms that allow Cost++ to be extensible for structuring and linking new types of workproducts.
- Forward references and multiple reference to nodes.
- Control over saving user annotations and links that were created for previous versions of workproducts.

The main focus of this report is on describing this language and clarifying how Cost++ can be used to generate hypertexts for applications used with Kiosk.

1.1 When Cost++ Can be Used

Effective use of Cost++ requires a clear understanding of the types of information it can support. If you have information you believe would fit well with a hypertext representation and you wish to use it with Kiosk, a major question for determining its use is: "How will this information evolve?" Will you have an *evolving hypertext*—one that persists over time and whose structure or content change? If so, there are limitations with what Cost++ can perform. The rest of this section will more clearly define the space of evolving hypertexts and consider the support and limitations provided by Cost++ within this space.

1.1.1 The space of evolving hypertexts

The changes to an evolving hypertext can be both *tool-based* and *user-based*. A tool-based change is one in which a tool (e.g., Cost++) is used to construct or modify an existing hypertext. This can include automatically linking existing nodes, as well as generating new nodes. A user-based change is one in which a user interactively inserts, deletes, or modifies nodes or links within the hypertext. Difficulties arise when a hypertext can evolve through both user-based and tool-based modifications. The main problem is ensuring that the nodes and links (re)generated by tool-based modifications do not lose or invalidate any user-based modifications. Of course, if all changes can be made through the use of the tool, this is no longer a problem; unfortunately, users frequently wish to add new, idiosyncratic information that is difficult or impossible to capture mechanically.

For tool-based interaction, we can form a scale from least interactive to most interactive. As shown on the vertical scale in Figure 1, at level 0 for tool interaction, a hypertext is built, completely disregarding any previous hypertext that might have existed. It may be built once, or many times, but if users begin to evolve it, there is never any interaction using the tool. At level 1, the tool can link in new nodes to an already existing hypertext, or unlink and remove nodes from the hypertext. At level 2, the tool can integrate new versions of already existing nodes into the hypertext. This could include anything from simply adding a new version of the node, to intelligent merging of new and prior versions of a node to replace it in the hypertext. To perform level 1 and level 2 interactions automatically, some mechanism must exist to specify when to run the tool. Currently, Cost++ can only handle level 1 tool interaction, unless the hypertext is static in nature, in which case it can handle level 2.

For user-based interaction (the horizontal scale in Figure 1), level 0 represents completely static hypertexts, that are created once as read-only structures that users navigate. At level 1, users have the ability to annotate nodes; however, the predefined nodes and links that comprise the hypertext cannot be modified. At level 2, user-

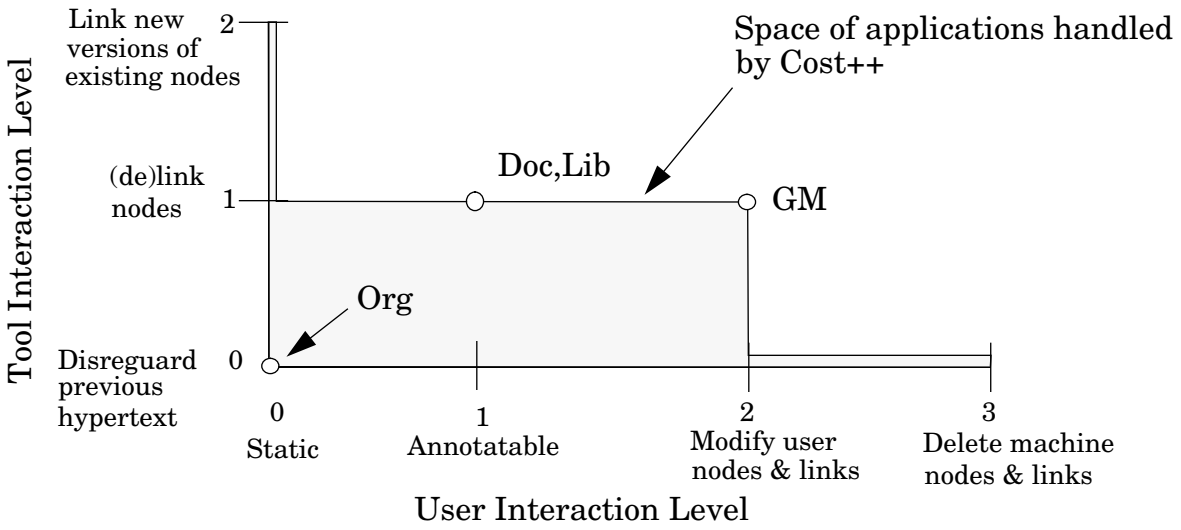


Figure 1. Space of Evolving Hypertexts

created nodes and links can be inserted, deleted, and modified along with annotations. Also, the text of existing machine generated nodes can be edited. At level 3, users also have the ability to delete mechanically generated nodes and links. Cost++ can only handle level 2 user interactions, unless the hypertext is only generated once (level 0 tool interaction), in which case it can handle level 3.

In Figure 1, the shaded area represents the space of possible applications that can be handled by Cost++. Now let's look at some applications and see where they fit in this space.

1.1.2 Sample applications in the space of evolving hypertexts

- A read-only organizational chart used for browsing purposes.

  This application consists of a person and project node hypertext that is used with the Kiosk navigation facilities to help find information on a particular person or project. Each time the organization changes, the config file for generating this hypertext would be edited and Cost++ re-executed to rebuild the chart.

  In this situation, we would have level 0 tool interaction since the org-chart is completely rebuilt each time Cost++ is executed—disregarding the previous version. The user interaction level is also 0, since users don't modify the org-chart (see Org in Figure 1).

- On-line hypertext documentation.

  For this application, a set of hypertext documents are produced. Users read them using Kiosk where they can quickly jump to the definitions of terms they read and to related documents. The hypertext for this documentation requires linking

entries in the table of contents of each document to their corresponding sections in the actual document, linking terms defined in one document to references to these terms in other documents, and linking each term within a glossary to the terms definition within each document. Each time the source documentation is updated, Cost++ is executed to build a new linked version of the documentation.

If users are not allowed to annotate this documentation, we have the same situation as with the organizational chart—namely level 0 tool and level 0 user interaction. If users are allowed to annotate the documentation, Cost++ can only be used to add new documentation to the hypertext (level 1 tool and level 1 user interaction, see Doc in Figure 1). It cannot handle the case of updating the documentation for any changes that occur to it; while, at the same time, preserving user annotations (level 2 tool integration with level 1 user integration). The only way we can both have user annotations and regenerate new versions of the documentation is to have separate annotations for each version of the documentation.

- A "group memory" (GM) of structured mail messages.

  In this application, each mail message is classified and clustered according to its free-text content and keywords. Users submit messages of interest via a program or script that invokes Cost++ to automatically link these messages into the existing GM. Messages of interest are found using Kiosk's browsing and querying facilities. Kiosk is also used to manually add new nodes and links to the GM.

  For this application, level 0 tool interaction is inadequate since users manually add new nodes and links. Level 1 is acceptable, since the same mail messages are not modified and resubmitted (level 2). The minimum user-based interaction needed is level 2, since users will annotate, add, delete, and modify their own nodes (see GM in Figure 1). They will also edit machine-generated nodes (e.g., add synonyms). Note that Cost++ cannot handle the preferable user level 3, where users don't have to concern themselves with user- versus tool-created nodes.

- Reusable libraries of software components.

  In this application, workproducts that make up a component, or module, (e.g., source code, header file, documentation) are clustered together by cluster nodes. These cluster nodes are also placed into multiple classification hierarchies to present different views of the components within the libraries. The structuring of these libraries also includes direct workproduct linking, as in linking manual page "see also" sections to the manual pages they reference. Users can also annotate nodes to specify information about bug reports or concerns. When used with Kiosk, cluster nodes and direct workproduct linking give rapid access to related workproducts. Using Kiosk to navigate and filter over the classification

hierarchies allow users to find components of interest and gain a better understanding of the structure of particular libraries.

The config file for a library is edited when new components are to be added, or when a change is made to the classification of existing components. Cost++ is then rerun, using this config file, to change the library's hypertext structure, while keeping user's annotations intact. When a new version of a library is released, a new hypertext structure is built—independent of the previous versions of this library.

For these libraries, we have the same situation as the annotated on-line hypertext documentation—level 1 tool and level 1 user interaction.

In the applications we have examined, Cost++ needs to produce different linking behavior. For the org-chart, links from a previous run of Cost++ are completely ignored. While for the on-line documentation and reuse libraries, Cost++ saves previous user generated links. And for the GM, Cost++ saves all previously generated links. A global option for Cost++ exists for controlling how to handle previous links. For details on this option, see page 27 or, section 7.2.1 on page 47.

1.1.3 Summary of when Cost++ can be used

Based on the evolving hypertext restrictions we have seen, the following rules apply. In order to use Cost++, the hypertexts generated must either:

- Not change between separate executions of Cost++, where Cost++ is being re-executed to rebuild or add to these hypertexts.
- Only change through the use of Kiosk[1] to edit existing nodes and links and deletion only occurs on user built nodes and links.

The following sections discuss the hypertext structures built by Cost++, and how to understand and create Cost++ config files within the context of several examples.

2 Foundations

This section will give an overview of the hypertext structures built by Cost++, along with an overview of the config file execution process Cost++ uses to structure information. If you already know how Cost++ config files work, or you just want information on how to run Cost++, see section 8 on page 70.

2.1 Hypertext Structures Built by Cost++

Cost++ builds a hypertext representation that overlays the workproducts it structures. This representation is used by other Kiosk tools. To use Cost++, it is necessary to understand this representation and how it is used by Kiosk.

1. Actually, anything can be used that ensures that link offsets are kept up-to-date. For example, a *Unix* Perl script could be written that edits a node and link file—keeping the link offsets consistent.

We will now present a rather terse summary of this representation—introducing many terms that are used throughout this document. The end of this section includes an example that shows the use of these terms.

The basic hypertext structures are *text nodes* and *binary links*. Groupings of these linked nodes form *lattices* or *webs* of interconnected nodes that may form general graph structures.

A text node (or just "node") is a *Unix* text file plus any links that point into or out of that file. Nodes contain a string that represents the contents of the file and a name that is the full pathname of the file.

A binary link (or just "link") connects two nodes. A links has a *role* that represents the relationship between the nodes connected by it. It also has an *owner* that specifies who created it.

Binary links are *bi-directional* in that they can be seen from either of the nodes they connect; however, they do have a direction. This direction is determined by the *source* and *destination* of each link. When the destination of a link points at a given node, this link is said to be *inbound* into this node. When the source of a link points at a given node, this link is said to be *outbound* from this node. If both the source and destination of a link point at the same node, the link is said to be *circular*. To see these directions, consider the example of two nodes connected by a link—the node `node1` and the node `node2`. This link has a source of `node1` and a destination of `node2`. Thus, this link would be inbound to node `node2` and outbound from `node1`. There is also another link whose source and destination point at `node1`. This link is circular to `node1`.

A Link can just connect two nodes (a *global* link), or connect from a point *inside* one node to a point inside another node (a *point-to-point* link), or any combination (e.g., *global-to-point* and *point-to-global*). Point-to-point links are useful for linking at a specific place in a text file, like from the definition of a class within a C++ header file, to its parent class definition in another C++ header file. These positions are represented as integer character offsets into the nodes connected by the link. The offset into the source node is the *source offset* and the offset into the destination node is the *destination offset*. A link whose source or destination is global to a node has an offset of zero.[2]

Another type of link exists called a *value link*. Value links act as "attributes" or "property-lists" for a node. A value link connects to just one node (source) and contains a simple string value instead of a destination. It can be anchored to any point within a node, but is usually just a global link. Value links are used heavily within Kiosk. An example of a value link is one with role `author` and a value of `Michael L. Creech`. Value links are also used to give types to nodes as discussed below.

2. This leads to a current ambiguity in Kiosk between a global link and a point-to-point link with offset 0.

Within Kiosk and Cost++, all nodes and links are represented as C++ objects. Outside of Kiosk and Cost++, nodes are the text files they represent and links are stored in "shadow" files adjoining the nodes they link to.[3] This representation makes links *non-intrusive* which means a text file need not be modified to add links to it. Thus, you need not worry about modifying the nodes (files) you wish to link. Files to link can therefore be read-only.

We will now turn to an example of this hypertext representation. Assume we wish to make it easy for a software engineer to use Kiosk to bring up related manual page information. More specifically, we would like to mouse click on a manual page referenced in the "see also" section of another manual page and be able to immediately bring up this manual page. We will use the manual pages for two different software components within the InterViews[5] library.[4] These are `Interactor.3I` and `Canvas.3I`. These two manual pages are related in that the `Interactor.3I` manual page refers to the `Canvas.3I` manual page in its "see also" section. We would like to make a hypertext representation that links the reference to `Canvas.3I` within `Interactor.3I` to the actual `Canvas.3I` manual page. We would also like to add a value link to `Interactor.3I` that specifies that this node is a workproduct.[5]

Figure 2 shows such a hypertext representation in a graphical form. The diagram shows a web, or lattice, of two nodes and two links. One of these links is a binary link that connects the node `Interactor.3I` to the `Canvas.3I` node. The source of this link is `Interactor.3I`, thus this link is outbound from `Interactor.3I`. The link is anchored at the 'C' in `Canvas (3I)` within the `See Also` section of the manual page. The source offset of this position is 1347. The destination of this link is `Canvas.3I` and is thus, inbound into this node. The link is anchored to the beginning of this node, making the destination offset 0. Because the link points within `Interactor.3I` and not into `Canvas.3I`, the link is a point-to-global link. The role of this link is `see_also` and the owner of this link is `Dennis Freeze`.

The other link is a value link that globally connects to `Interactor.3I`. It has a role of `Type` and a value of `Workproduct`. This link specifies that this node is a workproduct node.

2.1.1 Major node types

The hypertext representation built by Cost++ consists of linked *workproduct* nodes and *structure* nodes. The workproduct nodes are the text nodes that represent the workproducts being structured. Structure nodes are text nodes *generated* by Cost++

---

3. These files currently have the name of the node with a dot ('.') in front of it, followed by the suffix ".links" (e.g.,.banana.links).
4. InterViews is a public-domain user-interface construction toolkit written in C++.
5. Kiosk uses Type value links to distinguish different types of nodes to present them in different ways. Three major node types are generated by Cost++ and recognized by Kiosk (they are discussed below).

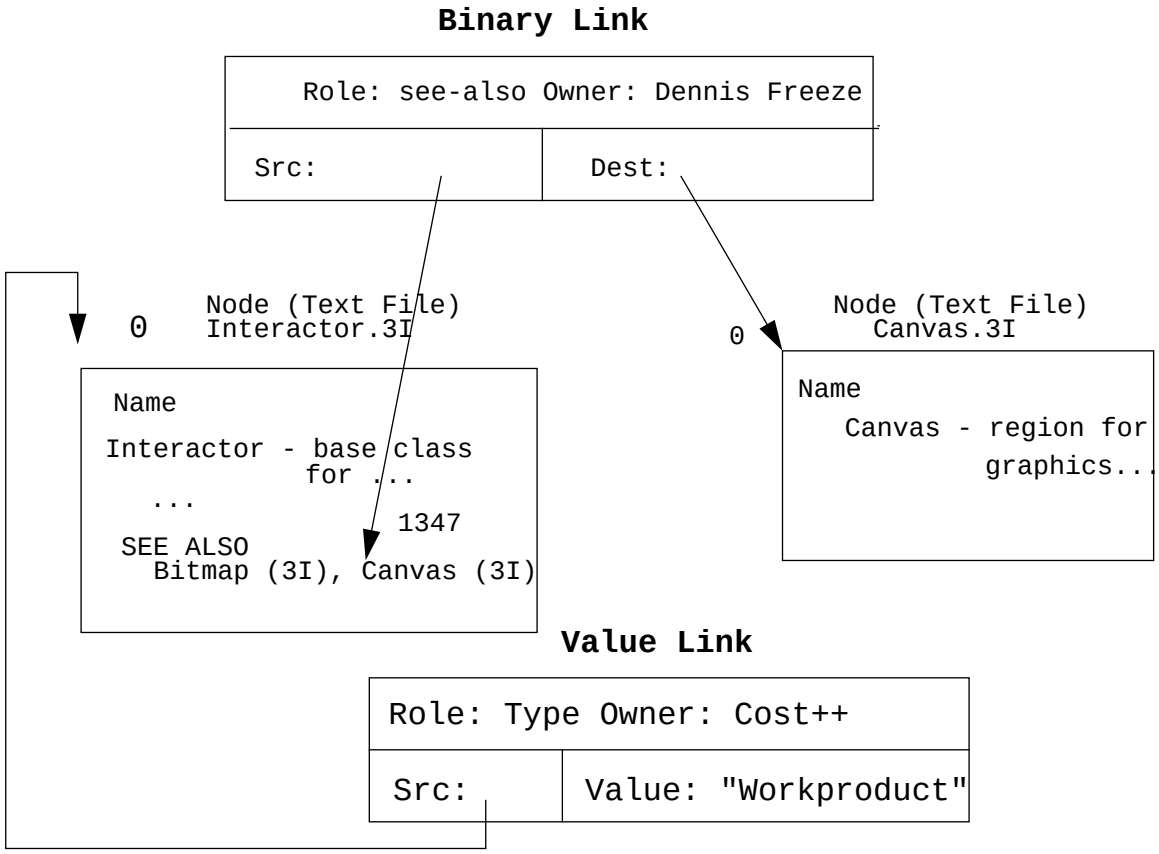


Figure 2. Hypertext Representation of Two Linked Manual Pages

to help classify, cluster, and catalog workproducts. Structure nodes come in two forms:

Classification Nodes—classify and catalog other nodes. When linked together, they can display the structure of workproducts that would otherwise not be apparent. They are also used to create different views of workproducts or clusters. They are commonly used to build structures that allow you to navigate from very general information to more and more specific information until specific workproducts of interest are found. For example, we could build a “functional” view of software workproducts based on the functions performed by these workproducts as well as an “architectural” view based on the purpose these workproducts serve in the architecture of a system. In the functional view, we may have classification nodes like “Computer Software” that represent very general concepts and information. These would then be linked to more and more specific classification nodes like “Data Structures” until actual workproduct nodes were reached.

Cluster Nodes—gather related workproducts into clusters for easy access. Within Kiosk, cluster nodes act as "switching stations" that allow very quick access to related information. Within the software domain, cluster nodes usually link workproducts that perform the functionality described by a manual page or equivalent documentation. For example, in the InterViews example above, a cluster exists for `Canvas` that represents all the workproducts directly relate to the functionality described in the `Canvas.3I` manual page. This cluster links together all of the `Canvas` header files, source code, documentation, tests, and configuration files. These clusters usually contain text fields that are linked to the workproducts they represent.

It may seem that there is a fine line between cluster nodes and classification nodes. There is—anything that can be performed with cluster nodes can be performed using classification nodes. So, it is a matter of the intended use of these nodes versus functionality that determines when they are used.

Although workproduct, cluster, and classification nodes are the major types of nodes created by Cost++, any type of node can be created by having Cost++ generate a given `Type` value link (see `NODE-TYPE` on page 15).

2.2 Stages of Execution

As mentioned above, Cost++'s behavior is determined by the declarative instructions placed in config files. Before we turn to the details of what these files contain, it is important to have an overall picture of what Cost++ does when it executes a config file.

When Cost++ is executed, a set of config files are read that control which workproducts will be read in, what links and structure nodes to create, and the final lattices of links and nodes to persistently store. This is a six stage process of:

1. Reading in the *definition* section, *linking* section, and the *post-processing* section of the config file (these are all optional).

   The definition section defines the meaning of actions used in the structuring section (see below) of a config file, values of global defaults to be used, along with which nodes will be handed to feature extractors (see below).

   The linking section defines exactly when and how feature extraction should be performed on nodes. For example, we might want to link all member function definitions to their class declaration within C++ code. This could be accomplished through the use of the linking section, along with some specifically written *feature extractors* that determine where a class declaration and member function definition can be found within a text file.

   The post-processing section defines operations to be performed after all linking and other operations have been performed (see below).

2. Reading and global linking of memory resident workproducts according to the *structuring section* of the config file.

   The structuring section gives the exact plan of what workproducts are to be read in, how they should be clustered, what global links should be created, and where these links and nodes should be saved. Many structuring section statements consist of action calls to actions defined in the definition section. It is also the time that localized feature extraction may take place.

3. Performing feature extraction between lattices.

   Certain types of feature extraction cause the linking of nodes found in different lattices. To perform this, all the nodes involved must be read in or created; thus, this style of linking occurs after workproducts are read in and after structure nodes are created.

4. Cleaning out undesired structure node contents.[6]

   After all linking has taken place, it may be desirable to remove certain template generated lines from structure nodes. These are usually lines that were used as link *placeholders* and are removed because no links were created for them.

5. Perform post-processing operations.

   Post-processing operations are used to perform certain actions on the created lattices after they are complete. For example, we might want to build hypertext documentation by linking “mark-up” language terms defined in documents. As a final step, we may need to remove these mark-up language statements from the documents.

6. Persistently saving out the lattice(s) of links and nodes created.

   At this point, all the links and nodes created in the previous steps are saved into text files. All the links for a given node are saved in the shadow file associated with the given node. Nodes that are written out include structure nodes created during the lattice building process, as well as workproduct nodes read in that are to be saved to a different location (full pathname). If Cost++ is running in multi-user mode (see section 8.2 on page 70), it may abort this step if nodes that it is modifying are being modified by another Kiosk or Cost++ program.

2.3 General Config File Format

The format of config files is loosely structured in that only one structuring section can exist and must be found at the end of a config file. Multiple occurrences (or no occurrences) of the other sections can be specified in any order before the structuring section. Note, however, that the linking section usually uses the information setup by

6. This will be merged with the post-processing operations in the future.

the definition section. Therefore, the definition section usually precedes the linking section. Because Cost++ reads config files in a single pass, when the same section is specified more than once, it will only be defined in terms of the previous sections read. Here are some examples:

Format of a typical config file:

```
<definition section>
<linking section>
<post-processing section>
<structuring section>
```

A less typical, but legal config file format:

```
<post-processing section>
<definition section>
<linking section>
<definition section>
<linking section>
<structuring section>
```

An illegal config file format:

```
<structuring section>
<structuring section>
```

2.4 Simple Examples

Let's now look at a few examples of config file usage in light of the hypertext structures built. For our first example, imagine we want to build a hypertext of people's first names based on their language of origin. For each name, we wish to have its meaning and links to variant and pet forms of a name. Using Kiosk, we can then search for names based on their meaning and quickly look at variant and pet forms of the same name. We will start with the following very simple config file fragment and expand it throughout this document:

```
(STRUCTURES
  (:CLASS ("Latin" ROLE:language)
    (:CLASS ("Miles" ROLE:variant)
      (:CLASS ("Myles"))
    )))
```

Figure 3. Very Simple First Names Fragment

The structuring section starts with a set of parentheses where the opening paren is followed by the word `STRUCTURES`. Everything in between these points is interpreted as part of this section. In our example, we have three statements that begin with `(:CLASS...)`. These statements tell Cost++ to build classification nodes. The name

of the node to build is the string following the classification node specifier. Thus, we have classification node `Latin, Miles,` and `Myles`. The `ROLE:language` specifies the link role to use in connecting all direct child nodes of this node. Thus, `(:CLASS ("Latin" ROLE:language)` builds a classification node with name `Latin` and links it to all its child nodes (`Miles`) with the link role `language`.

The result of this config file is a lattice with three classification nodes and two binary links. The links have the role `language` and `variant` and are global-to-global links (offsets are zero). This structure can be seen in Figure 4.

Figure 4. Very Simple First Names Tree

As an example from the realm of reusable software libraries, here is a fragment that generates a `Canvas` cluster node for the Canvas component discussed above, along with part of a surrounding classification:

```
(STRUCTURES
  (:CLASS ("Graphics-Output" ROLE:object-view)
    (:CLASS ("Medium" ROLE:object-view)
      (:CLUSTER ("Canvas" ROLE:workproducts)
                (NROFF-DOC "Canvas.3I")
                (C++-SRC-CODE "X11-canvas.c")
                (HEADER "canvas.h")
      ))))
```

Figure 5. Simple Canvas Cluster Fragment

Here we have the new form `:CLUSTER` which causes a cluster node to be generated. It has the same form as `:CLASS`—it will generate a cluster node with the name `Canvas.` This cluster node specifier has three *action-calls* that follow it—`NROFF-DOC`, `C++-SRC-CODE`, and `HEADER`. These action calls normally read in workproduct nodes. In this example, `NROFF-DOC` reads in the Nroff document `Canvas.3I`, `C++-SRC-CODE` reads in the C++ source code file `X11-canvas.c`, and `HEADER` reads in the header file `canvas.h`. All of these workproducts are linked to the `Canvas` cluster node by links with role `workproducts`.

The result of this config file fragment is two classification nodes, one cluster node, and three workproduct nodes. They are all globally linked by links with role `object-view` and `workproducts`. This can be seen in Figure 6.

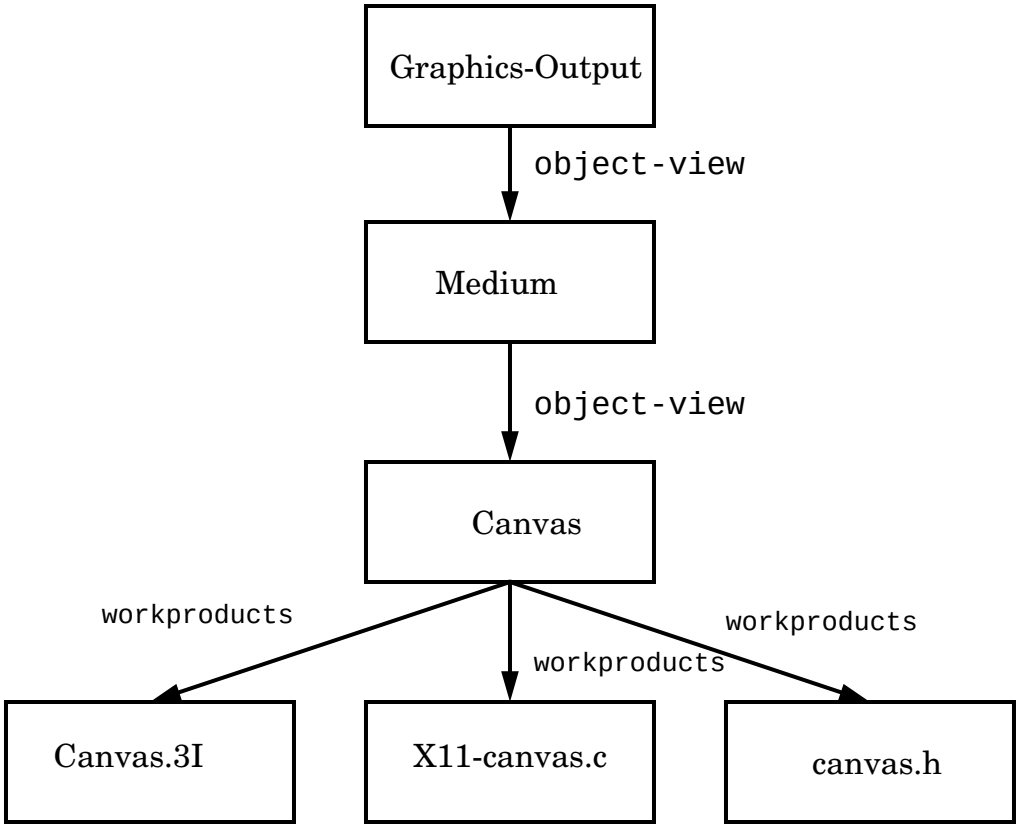


Figure 6. Simple Canvas Cluster Tree

3 Cost++ Structuring Section

Now that we have considered a few structuring section examples, let's look at this section in more detail. The structuring section gives the exact plan of what workproducts to read in and how they should be structured. It consists of any combination of 4 types of *statements.* Each of which begins with a colon (:). Two types of statements have already been presented—the `:CLUSTER` and `:CLASS` statements (also known as *structure* statements)—which build cluster and classification nodes respectively. The two other statements are the `:INCLUDE` and the `:SET` statements. The `:INCLUDE` statement reads config file information from other files. This is useful in making config files easier to read and allowing different config files to share information.The `:SET` statement is used to set options and defaults for use during the processing of the structuring section. The use of the `:INCLUDE` and `:SET` statements will be discussed further in section 7.2 on page 47.

Comments can be placed in config files and will be used in the following examples. A comment begins with a '%'. The rest of the line following the comment is ignored. Comments can be placed anywhere between config file statements.

3.1 Structure Statements

Structure statements build structure nodes used to link together workproducts and other structure nodes. Let's continue looking at some examples and define some terminology.

3.1.1 Structure statement form

Here are some fragments from examples shown above:

```
(:CLUSTER ("Canvas" ROLE:workproducts)
            (NROFF-DOC "Canvas.3I")
            (C++-SRC-CODE "X11-canvas.c")
            (HEADER "canvas.h")
)
(:CLASS ("Latin" ROLE:language)
   (:CLASS ("Miles"))
)
```

The first parenthesized expression following `(:CLUSTER` is known as the *cluster-action-call*. It consists of a *cluster-name* (`"Canvas"`) followed by zero or more *structure-option*s. The cluster-name is the name of the cluster we are defining. It is used to identify a given node and determine the exact file to read information from and write information to. Structure-options specify characteristics about this node and its relationship with other nodes. We'll discuss these in detail shortly.

Following the cluster-action-call are one or more *action-calls* that define actions to perform—usually reading in and linking workproduct nodes. In our example, this includes:

```
(NROFF-DOC "Canvas.3I")
(C++-SRC-CODE "X11-canvas.c")
(HEADER "canvas.h")
```

Each action-call begins with an *action-name* (e.g, `NROFF-DOC`) and is followed by zero or more string arguments (e.g., `"Canvas.3I"`). The exact behavior of action-calls is determined by their corresponding definitions found within the definition section of the config file (see section 4 on page 23).

Following the action-calls are zero or more child structure statements which may be linked to this cluster (this example has none). If a `ROLE` structure-option is specified, a global link will be created between this node and each child node.

Classification statements have the same form, but the cluster-action-call is called the *classification-action-call* which consists of an equivalent *classification-name* followed by zero or more structure-options. In our example above, `"Latin"` is the classification-name and `ROLE:language` is the structure-options. This example has no actions-calls and one child structure statement.

3.1.1.1 Structure-options

Structure-options have the form `<option-name>:<value>`. Thus, in our `Canvas` example, the `ROLE` structure-option has the form `ROLE:workproducts`. There are several structure options that affect the definition and linking of structure nodes. A few of these are:

ROLE—the role to use in linking this node to child structure nodes and to workproduct nodes read through action-calls. If the value is `UNDEFINED`, no linking occurs. A role has a *scope* which determines which of several roles is used to link child nodes to their descendents (see section 4.1 on page 24 and section 7.2.2 on page 48 for details). The default value is `UNDEFINED`.

INBOUND—the direction of links created to child nodes. This direction is with respect to child nodes. A value of `Yes` will cause links to be outbound from this structure node and inbound to the child nodes. A value of `No` will cause links to be inbound to this structure node and outbound from the child nodes. The default is `Yes`.

OUT—the location to save this structure node. It consists of the full pathname of a file or directory. When a file, this is the pathname of where to save this node. When a directory, the location is determined by concatenating this directory with the name of the structure node (e.g., `OUT:$rge` for the `Canvas` node would yield `$rge/Canvas`). The default is the current working directory.

NODE-TYPE—the type of the structure node being created. This corresponds to creating a value link with role `Type` that has the specified value. These links are used by Kiosk to treat nodes according to their type (e.g., presenting nodes in a different manner). A value of `UNDEFINED` causes no value link to be created. A value of `*DEFAULT*` will cause the value to be based on the type of this structure node (a value of `Cluster` for cluster nodes and a value of `Classification` for classification nodes). Any other value specified will be used as the value of this link. The default is `*DEFAULT*`.

Note that the default values mentioned above are used when the given structure-option is *not* specified. These default values can be changed through the use of the definition section and the `:SET` command (see section 4 on page 23 and section 7.2.1 on page 47). More structure-options will be discussed below. Let's now look at some examples using these options:

If we take the Very Simple First Names Fragment, from Figure 3, we can change the direction and meaning of the `variant` link.

```
(STRUCTURES
  (:CLASS ("Latin" ROLE:language)
 % Reverse direction of variant link and change its link role:
    (:CLASS ("Miles" INBOUND: No ROLE:variant-of)
      (:CLASS ("Myles"))
    )))
```

In this new fragment, we've changed the role of the `variant` link to `variant-of` and we have added `INBOUND: No` which will cause the `variant-of` link created to point in the opposite direction. Thus, this fragment generates:

The nodes created will be saved in the current working directory because we have no `OUT` structure option, so the default is used. The names of these files are `./Myles`, `./Miles`, and `./Latin`.

If we wished to save these nodes in `/tmp` instead, and also have Kiosk treat these nodes as cluster nodes instead of classification nodes, we could embellish this fragment to be:

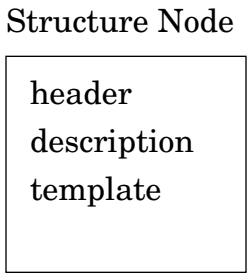
```
(STRUCTURES
   % Type as cluster nodes and save out files to /tmp:
  (:CLASS ("Latin" NODE-TYPE: Cluster ROLE:language OUT:/tmp)
    (:CLASS ("Miles" INBOUND: No NODE-TYPE: Cluster
                    ROLE:variant-of OUT:/tmp)
      (:CLASS ("Myles" NODE-TYPE: Cluster OUT:/tmp))
    )))
```

Figure 7. Embellished Very Simple First Names Fragment

3.1.2 Structure node contents

Although we have seen how structure nodes are created and linked, we have not yet considered their contents. The content of these nodes is determined by a combination

of three structure-options—all of which are optional (thus, a structure node can be empty). These options give structure nodes the form:



The *header* is a line that specifies the type and name of this structure node. The *description* is a short description of this node (e.g., its purpose or meaning), determined by reading in the contents of a *description file*. The *template* is text mixed with a set of *fields* and *placeholders* used to store links to other nodes. It is determined by the contents of a *template file*. Here are some examples:

The contents of the `Canvas` cluster, from Figure 5 on page 12, might have the form:

```
Cluster Name:  Canvas

  region of graphics output

----------------------------

  IMPLEMENTATION LANGUAGE: C++

  DOCUMENTATION:<+>
    SYNOPSIS:<+>
    DESCRIPTION:<+>
    SEE ALSO:<+>
  SOURCE CODE:<+>
  HEADER FILE:<+>
```

The header of this cluster is:

```
Cluster Name: Canvas
```

The description is:

```
region of graphics output
```

and the rest comprises the template, of which, lines like:

```
----------------------------
```

are simple text; whereas, lines like:

```
DOCUMENTATION:<+>
```

are a field (`DOCUMENTATION:`) followed by a placeholder (`<+>`). If links are to be created that relate to a given field, the link will be anchored to the location of the placeholder. In this example, if we decided to link the manual page `Canvas.3I` to this `Canvas` cluster node, we would do so on the placeholder following the `DOCUMENTATION:` field.

As another example, consider the `Miles` classification node:

```
Name: Miles

From the Latin militatus, meaning “a warrior, a soldier.” Used as
a short form of Michael in England.
```

This node consists of just a header (`Name: Miles`) followed by a description. It has no template. The choice of when to use these different structure node pieces depends on your application. Usually clusters have a template, and classification nodes have a description. When prototyping structure nodes, we might only want to include a header without a description. This allows us to quickly build structure nodes without having to immediately fill in their contents.

3.1.2.1 TEMPLATE file details

So far, we have not shown how generalized text is separated from field-placeholders within template files. This is done by special interpretation of the first line within the template file. This line has the form:

```
PLACEHOLDER='<placeholder-regexp>'
```

`<placeholder-regexp>` is the symbols that distinguishes a field-placeholder from general text within the template. It is used to determine exactly where to anchor links as well as remove unlinked field-placeholders (see below). In our `Canvas` cluster example, the template file would have a first line like:

```
PLACEHOLDER='<\+>'
```

Note that the placeholder is used in a regular expression pattern matching sequence within Cost++, where ‘+’ has special meaning. The ‘\’ before the ‘+’ says to literally search for a plus sign. For more details on the regular expressions used, see the *Unix regcmp(3x)* manual page.

After the linking process, not all field-placeholders may have links associated with them. For example, if the Canvas cluster above didn't have source code available, the SOURCE-CODE:<+> field-placeholder would not have any links. It is many times desirable to have such lines removed from a structure node. This is done using the REMOVE-CLASS-EMPTIES and the REMOVE-CLUSTER-EMPTIES action-options for removing non-linked placeholders. For details, see section 4.2 on page 27.

3.1.2.2 Structure-options for building the contents of structure nodes

Causing Cost++ to generate structure nodes to have the contents you desire is performed through the use of the following three structure-options:

HEADER—specifies whether a header should be created for this structure node. If Yes, a header is created, if No, it is not created. The default is Yes. When created, the header has the form Name: <classification-name> for classification nodes and Cluster Name: <cluster-name> for cluster nodes. <classification-name> is the name of this classification node (e.g., Miles) and <cluster-name> is the name of the cluster node (e.g., Canvas).

IN—the file or directory from which to read the description section of the structure node. This option has the same form as the OUT structure-option (see section 3.1.2.2 on page 19). When its value is UNDEFINED, no description is read. When a directory, the full pathname of the file to read is <directory>/<structure-node-name>. When the name of a file, this name is used as the full pathname of the description section file. The default value is UNDEFINED.

GENERATE-WHEN-MISSING—allows the IN structure-option to have a specific value, but if such a description section is not found, it will generate a new one (like IN:UNDEFINED). This is very useful for prototyping the construction of lattices, where the description files for nodes can be slowly written, while the remaining nodes are created to act as "stubs."

TEMPLATE—the full pathname of the file used to create the template for this node. If UNDEFINED, no template is created. The default is UNDEFINED.

Using these structure-options, let's respecify the Canvas cluster to produce the contents discussed above:

```
    (:CLUSTER ("Canvas" ROLE:workproducts
                        HEADER: Yes
                        IN: $rge/doc
                        TEMPLATE: $rge/doc/interviews-template)
                  (NROFF-DOC "Canvas.3I")
                  (C++-SRC-CODE "X11-canvas.c")
                  (HEADER "canvas.h")
     )
```

We also need to specify the contents of the template and description files. The template file `$rge/doc/interviews-template` would contain:

```
   PLACEHOLDER='<\+>'

----------------------------

  IMPLEMENTATION LANGUAGE: C++

  DOCUMENTATION:<+>
    SYNOPSIS:<+>
    DESCRIPTION:<+>
    SEE ALSO:<+>
  SOURCE CODE:<+>
  HEADER FILE:<+>
```

The description file, determined by the `IN` structure option, would be `$rge/doc/Canvas,` and would contain:

```
region of graphics ouput
```

To produce the contents of the `Miles` node discussed above, we might have:

```
   (:CLASS ("Miles" IN:$rgt/names))
```

along with a description file, `$rgt/names/Miles`, containing:

```
From the Latin militatus, meaning “a warrior, a soldier.” Used as
a short form of Michael in England.
```

3.2 Multiply Referencing Nodes

From time to time, the need arises to reference a node more than once. For example, the `Miles` node discussed above is both an English and Latin name. If we want to

reflect this fact in our Very Simple First Names Fragment (see Figure 4 on page 12), we need to add an `English` node that also points at the `Miles` node—multiply referencing the `Miles` node. Such references can be *backward* or *forward* in that the node being referenced might already be defined (backward), or will be defined in the future (forward).

Any node can be referenced (backwards and forwards) using the form:

```
#"<node-path>"
```

`<node-path>` is the absolute or relative pathname of where the node being referenced will be saved (e.g., $rd/foo.C, ./foo.C). As a simplification, backward references to structure nodes can be performed by re-referring to the same node two or more times.

Let's now attempt to expand the First Names example to include an `English` node:

```
(STRUCTURES
   (:CLASS ("Latin" ROLE:language)
     (:CLASS ("Miles" ROLE:variant)
       (:CLASS ("Myles"))
     ))
   (:CLASS ("English" ROLE:language)
     % A backward reference to the Miles structure node:
     (:CLASS ("Miles"))
   )))
```

Figure 8. Simple First Names Fragment

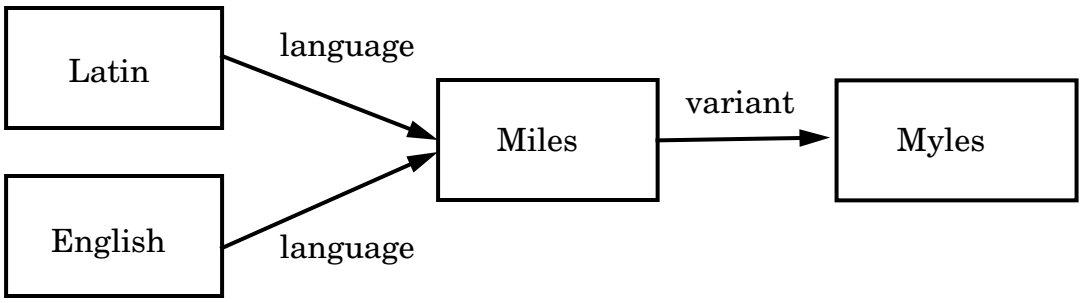
We could equivalently replace the second reference to:

```
(:CLASS ("Miles"))
```

with:

```
#"./Miles"
```

We use "." for the directory because the output location of the `Miles` node will be the current working directory since we didn't specify an `OUT` structure-option. In both cases, our new first names family tree looks like:



Remember that structure-options are always interpreted—even when re-referring to a node through a backward reference. It is important to realize that using the `OUT` structure option, in these situations, can determine whether a structure statement is a backward reference or not.

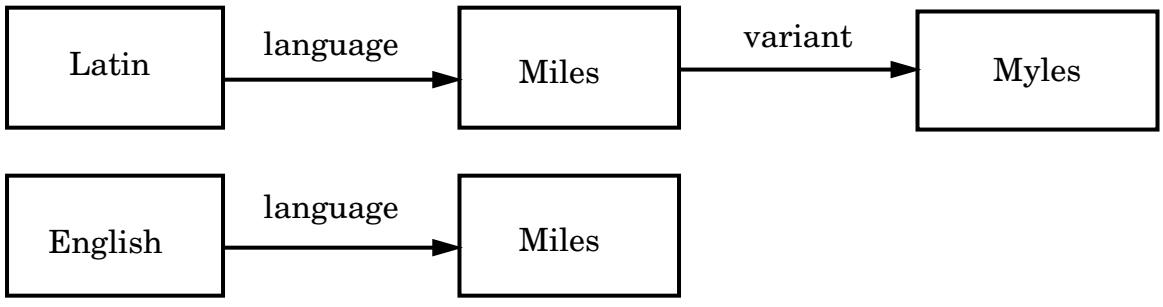
Consider the results of our last example when the current working directory is not `/tmp`, and we replaced the second reference to:

```
(:CLASS ("Miles"))
```

with:

```
(:CLASS ("Miles" OUT:/tmp))
```

We would no longer have a backward reference to the `Miles` node, but another node named `Miles` that resides on `/tmp`. The result of this structure would be:



4 Cost++ Definition Section

The definition section is used to define the meaning of *actions* called within the structuring section, to set options that affect the global execution of Cost++, to set general defaults for the structure-options of structure nodes, and to determine which nodes are read by feature extractors.

Let's begin with an example. So far, when we've built structure nodes, we've had to specify more and more structure-options. Many of these have been the same options repeated for different nodes. For simple examples, this is ok, but for config files containing many structure nodes, this becomes cumbersome and makes config files hard to read. Through the use of the definition section, we can alleviate this problem. As an example, let's revisit the Embellished Very Simple First Names in Figure 7 on page 16. In this example, we specified the structure-options `INBOUND`, `NODE-TYPE`, `ROLE`, and `OUT`. Using the definition section, we could define:

```
(DEFINITIONS
      % Make it so classification nodes have these options as a
      % default:
     (CLASS_MAIN ROLE:language NODE-TYPE: Cluster OUT:/tmp)
)
```

With such a definition section, the equivalent structuring section becomes:

```
(STRUCTURES
    (:CLASS ("Latin")
      (:CLASS ("Miles" ROLE:variant-of INBOUND: No)
        (:CLASS ("Myles"))
      )))
```

As you can see, we have eliminated the need to specify several of the structure-options for each classification node. The only options left are `ROLE:variant` and `INBOUND: No` for the `Miles` node.

Similar to the structuring section, the definition section is wrapped in a set of parentheses where the opening paren must be followed by the word `DEFINITIONS`. In between these points are zero or more *`action-descriptors`*. In our example, we have one:

```
     (CLASS_MAIN ROLE:language NODE-TYPE: Cluster OUT:/tmp)
```

`CLASS_MAIN` is the *action-name* followed by one or more *action-options*. These action-options include all of the structure-options we defined in section 3.1 on page 14, plus some new options that will be described below.

All the listed action-options will be used as the default for each action-call that has a matching action-name within the structuring section. Thus, the definition section

acts to set the defaults when specific structure-options are *not* specified in the structuring section. `CLASS_MAIN` is the action-name for the classification-action-call and `CLUSTER_MAIN` is the action-name for the cluster-action-call. In our case, all classification nodes created will have the defaults listed in the action-options.

Only one action-descriptor with a particular action-name can be specified. This includes action-descriptors specified in two or more definition sections within the same config file.

4.1 Defining New Action-descriptors

When building hypertexts, we may need to represent each type of workproduct differently. For example, we may wish to read in, write out, and link Pascal code workproducts differently than C++ code workproducts. These situations are handled by creating new action-descriptors that are used through action-calls made within structure statements. New action-descriptors take the same action-options of structure node action-descriptors (e.g., `CLASS_MAIN` and `CLUSTER_MAIN`), with the addition of a new value for the `ROLE` option (see below).

Let's start by more clearly defining the `Canvas` cluster example from Figure 5 on page 12 and from section 3.1.2.2 on page 19. In this example, we had:

```
(:COMPONENT ("Canvas" ROLE:workproducts
                      HEADER: Yes
                      IN: $rge/doc
                      TEMPLATE: $rge/doc/interviews-template)
            (NROFF-DOC "Canvas.3I")
            (C++-SRC-CODE "X11-canvas.c")
            (HEADER "canvas.h")
)
```

We have not defined action-descriptors that determine the behavior of the action-calls to `NROFF-DOC`, `C++-SRC-CODE`, and `HEADER;` specifically, we have not said where workproducts are read from and saved to. But first, let's clean up this cluster's structure-options by using defaults found in the `CLUSTER_MAIN` action-descriptor, similar to the `Latin` example, above. The assumption here is that all InterViews clusters would need similar defaults. If they didn't, we could override these defaults using structure-options. Our example, now becomes:

```
(DEFINITIONS
   (CLUSTER_MAIN ROLE:workproducts HEADER: Yes IN: $rge/doc
                  TEMPLATE: $rge/doc/interviews-template)
 )
(STRUCTURES
    (:CLUSTER ("Canvas")
                (NROFF-DOC "Canvas.3I")
                (C++-SRC-CODE "X11-canvas.c")
                (HEADER "canvas.h")
    ))
```

Now let's define where Nroff documents, C++ source code and C++ header files are read from and saved to. This is done by adding three action-descriptors to the definition section, giving:

```
(DEFINITIONS
   (CLUSTER_MAIN ROLE:workproducts HEADER: Yes IN: $rge/doc
                  TEMPLATE: $rge/doc/interviews-template)
   (NROFF-DOC    IN:$rlsi/nroff-doc  OUT:/tmp/test ROLE:*DEFAULT*)
   (C++-SRC-CODE IN:$rlsi/src        OUT:/tmp/test ROLE:*DEFAULT*)
   (HEADER       IN:$rlsi/headers    OUT:/tmp/test ROLE:*DEFAULT*)
)
```

From this definition section, the `Canvas` component action-call to `NROFF-DOC` will read the file `$rlsi/nroff-doc/Canvas.3I` and save this file to `/tmp/test/Canvas.3I`. Similar behavior occurs for action-calls to `HEADER` and `C++-SRC-CODE`. The purpose of the `*DEFAULT*` role action-option is to ensure that all workproducts read by our new action-descriptors will be linked using the role `workproducts`. It is described in detail below.

It is common to want to have a separate role for each workproduct, so that Kiosk filtering can be used to only view workproducts linked with specific roles. This is easily achieved by changing the `ROLE` action-option for each of our new action-descriptors, giving:

```
(DEFINITIONS
   (CLUSTER_MAIN ROLE:workproducts HEADER: Yes IN: $rge/doc
                  TEMPLATE: $rge/doc/interviews-template)
   (NROFF-DOC    IN:$rlsi/nroff-doc  OUT:/tmp/test ROLE:doc)
   (C++-SRC-CODE IN:$rlsi/src        OUT:/tmp/test ROLE:src)
   (HEADER       IN:$rlsi/headers    OUT:/tmp/test ROLE:header)
)
```

Figure 9. Canvas Example with all Action-descriptors Defined

Notice that the meaning of ROLE for workproducts is somewhat different then what is used for structure nodes. For structure nodes, ROLE specifies the role of the link to use for linking descendents of this node. For workproducts read, via new action-descriptors, there are no children nodes. Instead, ROLE means the role of the link to the direct parent of this workproduct.[7] This raises the issue of the scope of a role—which role takes *precedence* when both roles are specified. In our example, we have the NROFF-DOC workproducts to be linked with the role doc, and the CANVAS cluster node to link to all its workproducts using the role workproducts. As you might have guessed, new action-descriptors that specify a role take precedence over structure node action-descriptors. Thus, in our example, the role of the link to $rlsi/nroff-doc/Canvas.3I will be doc because the ROLE action-option of NROFF-DOC has precedence over the one for CLUSTER_MAIN. Note that the ramifications of this are that if a new action-descriptor uses the default role setting, or sets its role to UNDEFINED, no global links will be created to the workproducts read—even if its containing structure node has a specific role. So, in our example, if we remove the ROLE action-option in NROFF-DOC, no link will be made from the Canvas cluster node to the workproducts read in.

If we wanted to have a link generated from a structure node to a workproduct using the link role of the structure node action-descriptor, we must specify a ROLE of *DEFAULT* for the new action-descriptors that are to read the workproducts. This tells an action-descriptor to "default" its link role to what is specified by its parent structure node. Thus, if we specified ROLE:*DEFAULT* for NROFF-DOC, Canvas would link to the NROFF-DOC workproducts with role workproducts. Note that since structure nodes never directly read in workproducts, the value of *DEFAULT* is not legal for these nodes.

To find out more about role precedence and scope, see section 7.2.2 on page 48.

4.1.1 The ARGS action-option

Action-descriptors can control the number of legal arguments passed to action-calls through the use of the ARGS action-option. In the default case, action-calls can take an arbitrary number of arguments (a value of -1). Thus, if we had two source code files in our Canvas example, we could read them in by changing the C++-SRC-CODE action-call to:

```
(C++-SRC-CODE "X11-canvas.c" "X11-canvas2.c")
```

It is common for clusters to only have one header file. We can use the ARGS action-option to ensure that no more than one argument is given. Our HEADER action-descriptor would change to:

7. In the future, we may change these to ROLE-TO-DECENDENT, for structure nodes, and ROLE-TO-PARENT, for workproducts.

```
(HEADER ARGS:1 ROLE:header IN:$rlsi/headers    OUT:/tmp/test)
```

In this case, if more than one header file workproduct were given as an argument to a HEADER action-call, an error message would be issued.

4.1.2 Shell-style wildcards and the READ-LINK-FILES option

Unix Shell-style[8] wildcard characters can also be used to read in multiple files without explicitly specifying each one. Thus, we could read in all the source code files for Canvas through:

```
(C++-SRC-CODE "X11-canvas*.c")
```

When wildcards are used, ambiguities can arise as to whether actual link files should be read in as workproducts. This is because link files are named and can be picked up by the wildcard. In general, however, this behavior is not desirable, so the default is for wildcard expansion to throw out link files. For the few cases when link files are to be processed as workproducts, the action-option, READ-LINK-FILES, can be used with the value Yes (the default is No). For example, if the action-descriptor for C++-SRC-CODE were changed to:

```
(C++-SRC-CODE ROLE:src IN:$rlsi/src
              OUT:/tmp/test READ-LINK-FILES: Yes)
```

And we had an action-call like:

```
(C++-SRC-CODE "*")
```

All link files on $rlsi/src would be read in as workproducts.

4.2 The *GLOBAL* Action-descriptor

The special action-descriptor, *GLOBAL*, can be modified to set options that globally affect the execution of Cost++. This includes the amount of debugging information issued, how field-placeholders are managed in structure nodes, and which links should be read in with nodes. *GLOBAL* action-options are:

DEBUG—should more information be printed for debugging post-processing functions and feature extractors? If Yes, more information will be printed as config files are executed, if No, standard information will be issued. Default is No. For more details, see section 8.2 on page 70.

REMOVE-CLUSTER-EMPTIES—should unlinked field-placeholders be removed from cluster nodes (see section 3.1.2.1 on page 18 for details)? If Yes, after all linking has taken place, cluster nodes are searched to find field-

8. The wildcards are compatible with K-shell format.

placeholders that have no links within their placeholders. This search is performed by finding lines within the cluster that match the placeholder specified within the template file used to build each cluster. When such a match is found, the line is removed. If this action-option's value is `No`, no removal will take place. Note that if a template was not used in the construction of a cluster, the value of this option is irrelevant. Default value is `Yes`.

REMOVE-CLASS-EMPTIES—Same as `REMOVE-CLUSTER-EMPTIES`, above, but applies to classification nodes built with templates.

KEEP-LINKS—determines which links will be read in and saved with nodes. This option is very dependent on the applications Cost++ is being used for (see section 1.1 on page 2).

A value of `NONE` causes no existing links to be read in with a node. This is useful for applications that wish to cleanly rebuild lattices when Cost++ is run—throwing out all previously existing links (level 0 tool interaction). This option was necessary for the read-only organizational chart application discussed on page 3.

A value of `USER-GENERATED` will only cause links not generated by Cost++ to be read in with nodes. More specifically, only links not owned by `Cost++` are read in. This value is very useful for evolving user applications. For example, suppose we have built a reusable library of software components (see page 4) where we periodically want to run Cost++ to add new workproducts. In this situation, users can annotate and add links to this lattice and we can avoid losing their changes when Cost++ is run to add a new workproduct to the lattice. Note that this only works for links and nodes *added* to the structure. User deletions of Cost++ generated links will not be saved and these links will be regenerated the next time Cost++ executes.

A value of `ALL` causes all previous links to be read in with a node. This is useful for incrementally changing an existing lattice. For example, in the GM of structured e-mail messages (page 4), when a new mail message is incrementally added to the GM, we could read in a classification node, along with all of its links, and then link this new mail message to this classification node.

Default value is `USER-GENERATED`.

Some examples of the `*GLOBAL*` action-descriptor are:

```
(*GLOBAL* REMOVE-CLASS-EMPTIES: No KEEP-LINKS: User-Generated)
(*GLOBAL* DEBUG: Yes REMOVE-CLUSTER-EMPTIES: Yes)
```

4.3 Special Action-descriptors

Several other tasks are commonly performed besides building structure nodes and reading and linking workproducts. These tasks are performed through special action-descriptors. Let's first look as some of these different tasks and then see how special action-descriptors are used to perform them.

So far, we have seen no mechanism for generating value links for structure nodes. Value links are very useful for defining node properties and are heavily used to determine node types and other information within Kiosk. Some examples of useful value links we might want to create are:

- Adding value links to our `Canvas` cluster that specify the operating system and machine this component can execute on, as well as the library this component belongs to.
- Giving nodes symbolic names that are used within Kiosk instead of pathnames.
- Holding syntactic information about the beginning and ending of regions of text within a node for use by an editor.

Another task that comes up is the need to link together existing lattices that have been generated. For example, we might have several config files that each link together a specific software library. We might want to then link all these libraries together with a meta-level classification that provides quick access to each different library. This can be performed by generating the structure nodes of the meta-level classification and linking them to the root nodes of each of these libraries.[9]

4.3.1 Creating and using special action-descriptors

Cost++ distinguishes regular action-descriptors (i.e., all the ones we've seen so far) from special ones according to which *parse function* the action-descriptor uses. This parse function determines the exact semantics of what actions to perform and the meanings of the string arguments that follow the action-name within an action-call. All regular action-descriptors use the default parse function, `dir_assist_general_node_parse_func`, to determine their behavior. In this case, the string arguments are interpreted as nodes to read in based on the value of the `IN` action-option. To specify special action-descriptors, a different parse function is specified in a `FUNC` action-option within the sequence. Currently, two other parse functions exist—`value_link_parse_func` and `external_link_parse_func`:

9. This is not easy to perform with the existing machinery because we need to read in and save all links associated with each root library node *`except`* the previous link to the meta-level node we are creating (if it exists).

value_link_parse_func defines value links for structure nodes. It reads an arbitrary number of <role>-<value> pairs and builds one value link for each pair. These value links are globally linked to the structure node where this action-call appears.

external_link_parse_func globally links a structure node to another node external to the nodes that have been generated during this session with Cost++. It takes an arbitrary number of <full-pathname>-<role> pairs and generates a global link from the structure node where this action-call appears to the node referred to in the <full-pathname>. This link is given the role <role>.

Let's look at some examples. First, we will define a VALUE_LINK action-descriptor adding to our Canvas example from section 4.1 on page 24:

```
(DEFINITIONS
   (CLUSTER_MAIN ROLE:workproducts HEADER: Yes IN: $rge/doc
                 TEMPLATE: $rge/doc/interviews-template)
   (NROFF-DOC    IN:$rlsi/nroff-doc  OUT:/tmp/test ROLE:doc)
   (C++-SRC-CODE IN:$rlsi/src        OUT:/tmp/test ROLE:src)
   (HEADER       IN:$rlsi/headers    OUT:/tmp/test ROLE:header)
   % Special action-descriptors, using FUNC:
   (VALUE_LINK   FUNC:value_link_parse_func)
)
```

We can now add the value links, we described above, to our Canvas example with:

```
(STRUCTURES
   (:CLUSTER ("Canvas")
             (NROFF-DOC "Canvas.3I")
             (C++-SRC-CODE "X11-canvas.c")
             (HEADER "canvas.h")
              (VALUE_LINK "OS" "HP-UX/7.0"
                          "Machine" "HP9000/300"
                          "Library" "InterViews2.6")
    ))
```

Figure 10. Canvas Cluster with Value Links

The Canvas cluster node would now have three value links attached to it: one with role OS and value HP-UX/7.0, one with role Machine and value HP9000/300 and one with role Library with value InterViews2.6.

Now let's build part of a meta-level classification for two libraries—InterViews and Codelibs[6]. Assume the root node for each of these libraries is $rlw/roots/InterViews2.6 and $rlw/roots/Codelibs, respectively. We can now build a small classification structure using the following definition section:

```
(DEFINITIONS
   (CLASS_MAIN    OUT:$rlw/func_view ROLE:func_view)
   % Special action-descriptor, using FUNC:
   (EXTERNAL_LINK FUNC:external_link_parse_func)
)
```

Our classification structure has the form:

```
(STRUCTURES
   (:CLASS ("Views")
      (:CLASS ("Functional_View")
              (EXTERNAL_LINK "$rlw/roots/Codelibs" "func_view"
                             "$rlw/roots/InterViews2.6" "func_view")
      )))
```

With this, Cost++ would generate a structure of the form seen in Figure 11.

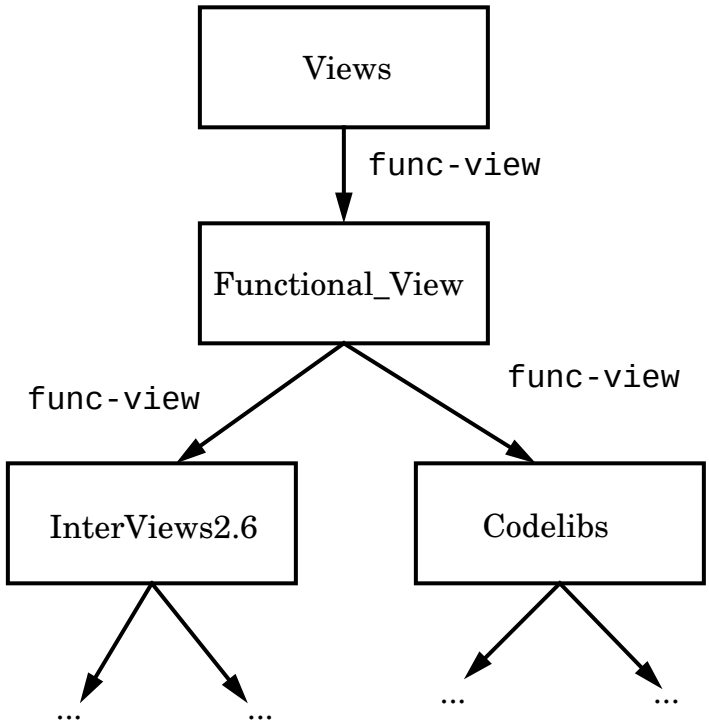


Figure 11. Meta-classification Linking InterViews and Codelibs

Customized parse functions can be built to perform specialized tasks. This is described in section 7.3.1 on page 50.

5 Cost++ Linking Section

So far, we have only been able to globally link nodes by explicitly specifying which nodes to link within a config file. There are, however, many cases when we would like to perform *content-based* linking where features *inside* nodes are linked according to their content. Some examples of content-based linking include:

- Linking the "see also" section of manual pages to the actual manual pages they reference.
- Linking parent to child class definitions in C++ header files.
- Linking the references to terms in one document to their definitions in another document.
- Linking keywords found in one node to nodes that have synonyms that match these keywords.
- Linking placeholders in structure node templates (as described in section 3.1.2 on page 16) to the workproducts they refer to.

The linking section of config files specifies when and how this type of linking occurs. Before looking at the specifics of how this is performed, we need a clearer picture of the overall linking process.

5.1 Overall Linking Process

What follows are the three steps that lead to content-based linking. The numbers used corresponds to the numbering used in the Stages of Execution (see section 2.2 on page 9):

1a. Cost++ first reads the linking section of the config file to set up the linking actions to perform during the structuring section.

2a & 3a. During the structuring section, when structure nodes are created and workproducts are read in, information from the linking section will trigger feature extractor functions and programs to be executed on these nodes to return interesting possible places to link and what identifies them. This information is stored in data objects called *linkspots*.

2b & 3b. According to the times specified in the linking section, Cost++ will attempt to match up linkspots that represent source locations and destination locations, to create internal links between various nodes. At this time, linkspots may also be deleted.

5.2 Syntax and Terminology

Similar to the previous sections, the linking section is wrapped in a set of parentheses, where the opening paren must be followed by the word `LINK`. In between these points are zero or more *node-relators*. Each one specifies two places to search in attempting to build linkspots as well as when and how to link up matching linkspots

found in these two places. As an example, if we wanted to link the manual page “see also” sections to the manual pages they reference (see Figure 2 on page 8), we could use a linking section of the form:

```
(LINK
   % Link SEE ALSO manpage item references to the actual manpages
   % they reference.
   (RELATE "see_also" *GLOBAL*
           (SRC_ITEM  NROFF_DOC m FUNC see_also_linkspots)
           (DEST_ITEM NROFF_DOC 1 FUNC smart_node_name_linkspots
                                                        True")
   ))
```

Figure 12. See Also Linking Section Example

This linking section contains one node-relator that begins with `RELATE`. It specifies that workproducts read using the `NROFF-DOC` action-descriptor will be searched by two feature extractor functions—`see_also_linkspots` (finds references within the see also section of manual pages) and `smart_node_name_linkspots` (finds actual manual pages). Because we may have many references to the same manual pages, this node-relator forms a many-to-1 relationship in that many linkspots returned by `see_also_linkspots` can refer to 1 linkspot returned by `smart_node_name_linkspots`. Any linking based on these linkspots is performed after all nodes from all structuring sections have been read (`*GLOBAL*` scope). When links are created, they will have the role `see_also`.

In general terms, each node-relator has the form:

```
(RELATE <link-role> <link-time>
                    <src-item>
                    <dest-item>)
```

where the specified attributes have the meaning:

link-role—the link role to use when links are created. In our example, this is `"see_also"`. It can be a simple string or the special symbol `*VARIABLE*` that specifies that link roles are to be dynamically determined by the identifier (see section 5.2.1 on page 35) of the matching linkspots when a link is created.

link-time—the time when actual linking should take place.[10] It specifies when stage 2b and 3b of the linking process occur and thus determines the *scope* of this node-relator. Another way of looking at link-time is, how long should

10. Note that forward referencing can currently cause linking to *not* take place if the forward reference is resolved outside of the scope of a node-relator.

linkspots accumulate before performing linking? For example, we might want to link a cluster node's template placeholders to the workproducts read in during the creation of this cluster. After the cluster is built, we would want to link based on all the linkspots found for that cluster, and start anew with the next cluster read. In this case, we would use a link-time of `*CLUSTER*`. Legal link-times include:

- `*GLOBAL*`—generate links after entire structuring section has been processed. This is the value we used in our example.
- `*LATTICE*`—generate links after each top-level lattice within the structuring section has been processed.
- `*STRUCTURE*`—generate links after each structure node is completely read in.
- `*CLUSTER*`—generate links after each cluster node is completely read.
- `*CLASS*`—generate links after each classification node is completely read.

src-item—determines the creation of source linkspots that represent the source of any links built from this relation. Properties specified include the feature extractor to execute, what nodes this extractor should consider, and how the source of links is related to the destination of links within this relation.

dest-item—determines the creation of destination linkspots that represent the destination of any links built based on this relation. This specifies the same information as the src-item.

With this information, let's refine the overall linking process from page 32:

2a & 3a.During the structuring section, when structure nodes are created and workproducts are read, feature extractors will execute over these nodes according to the src-item and dest-item of node-relators—possibly creating source and destination linkspots.

2b & 3b.According to the link-time of each node-relator, an attempt is made to match source linkspots with corresponding destination linkspots. When such matches occur, a link is created based on this source and destination linkspot information. The role of this link is determined by link-role. Source and destination linkspots that have no corresponding linkspot are ignored. After linking over the scope of a node-relator, all linkspots for that node-relator are removed.

Linking caused by a node-relator occurs independent from all other node-relators.[11] Thus, if we had two duplicate node-relators, two links would be generated for each match found.

11. However, for efficiency, linkspots are only generated once for duplicate src-items and dest-items found in different node-relators.

5.2.1 Feature extractors and linkspots

Feature extractors can occur as either built-in functions, programs, or scripts. Feature extractor functions are referred to as *function extractors*. And feature extractors that are scripts or programs are referred to as *program extractors*.

Linkspots are represented as C++ `LinkSpot`[12] objects that contain an integer character *position* in the node where a link might be anchored, a string *identifier* that allows this linkspot to be matched up with corresponding linkspots to create links, and the *Node* associated with this linkspot.

All feature extractors are passed the node to extract features from and zero or more string arguments that can be used as the feature extractor sees fit. All extractors attempt to return a list of linkspots. Function extractors are part of the Cost++ program and, therefore, adding new ones requires a limited amount of code to be written, along with recompiling and relinking Cost++.

Program extractors are executed by Cost++ in a separate process. The named script or program will be called with the pathname of the file to extract features from and zero or more optional string arguments. Linkspot information is returned from program extractors by writing information to standard output, which is then read by Cost++ to generate `LinkSpot` objects. Program extractors are useful in that Cost++ doesn't have to be recompiled to add and debug context-based linking abilities. However, since the extractor is run in a separate process on each node to search, it is much less efficient. For faster linking, function extractors must be used.

When performing content-based linking, using existing feature extractors is highly preferable to writing new ones. A description of existing feature extractors will be given in section 7.4.1 on page 53. Details on building your own feature extractors is given in section 7.4 on page 52.

5.2.2 Src-item and dest-item details

Let's now return to the details of the src-item and dest-item. Each of these has the form:

```
(SRC_ITEM | DEST_ITEM <where-look> <one-or-many>
          <extractor-type> <extractor-name> [<extractor-arg>]*)
```

where these attributes are defined as follows:

where-look—which nodes will be handed to the feature extractor defined by extractor-name (see below) to build linkspots. When where-look matches the name of an action-descriptor, the feature extractor will be called on each workproduct read in by corresponding action-calls on this action-descriptor.

12. In reality, linkspots are represented by a combination of C++ `LinkSpot` and `LinkInfo` objects, but for the purposes of discussion, they will be treated as one object. For a more realistic description, see section 7.4.3 on page 62.

Where-look can also be used to hand structure nodes or any workproduct to the feature extractor. This is done through the use of the values:

CLUSTER—hand all cluster nodes built to the feature extractor.

CLASS—hand all classification nodes built to the feature extractor.

STRUCTURE—hand any structure node (cluster or classification node) to the feature extractor.

WORKPRODUCT—hand any workproduct read in to the feature extractor.

In the see also example, only workproducts read in using the NROFF-DOC action-descriptor are handed to the feature extractors.

The nodes given to program extractors can be either structure or workproduct nodes depending on the value of where-look. Since structure nodes are created, the node is written to a temporary file so that the program or script can read the contents of this node. The temporary file is then deleted after the program extractor terminates.

Note that the type of nodes to hand to a feature extractor can be mutually exclusive with the scope of the node-relator. An error message will be issued for such cases, since the feature extractors would never be called. Such an example is a node-relator with link-time *CLUSTER* and a src-item or dest-item with where-look *CLASS*.

one-or-many—how linkspots generated by the feature extractor of this src-item or dest-item relate to linkspots from the corresponding item. This is used to determine which links to create as well as issue warning messages when feature extractors return values that don't correspond to the expected relationship. A value of 1 means there will be one linkspot with a given identifier. A value of m means there can be an arbitrary number of linkspots with the same identifier.

In the see also example, we might have many references to one particular manual page, thus we have a many-to-one relationship. As a result, the src-item specifies m and the dest-item 1. If the dest-item returned more than one linkspot with the same identifier, a warning message would be issued. This would correspond to two or more different manual pages with the same identifier.

Many-to-many relationships are possible. In this case, a link for each source linkspot with a given identifier is created to each corresponding destination linkspot.

extractor-type—the type of feature extractor defined for this src-item or dest-item. This specifies whether the feature extractor, extractor-name (see below), is a program extractor or a function extractor. A value of FUNC

specifies a function extractor where the extractor must be defined as part of the Cost++ program. A value of `PROGRAM` specifies a program extractor that is a separate program or shell script to execute.

In the see also example, both the src-item and dest-item use function extractors.

extractor-name—the name of the actual function or program extractor. When extractor-type is `PROGRAM`, this field represents the pathname of a script or program to execute.

When extractor-type is `FUNC`, this field is interpreted as the name of a function to call. The nodes given to these functions are determined by the value of where-look.

In the see also example, the src-item extractor function is `see_also_linkspots` and the dest-item extractor function is `smart_node_name_linkspots`.

extractor-arg—an optional string argument. Up to 16 arguments can be given to a feature extractor and are passed as the 2nd through 17th arguments. These are useful for helping the extractor perform different behavior based on attributes found in the config file.

In the see also example, only the dest-item extractor, `smart_node_name_linkspots`, is passed an extractor-arg, whose value is `"True"`.

5.3 Linking Section Example

Here is a linking section example containing four node-relators:

```
(LINK
  % Link from the definition of a term to all of its references.
  (RELATE "term_ref" *GLOBAL*
          (SRC_ITEM       HELP_DOC      1  FUNC markup_linkspots
                                             "@TERM-DEF" "False")
          (DEST_ITEM      HELP_DOC      m  FUNC markup_linkspots
                                            "@TERM-REF"  "False")
  )
  % Link between a class declaration in a header and the
  % top of the class's source code file where method definitions
  % are kept.
  (RELATE "header_to_src_code" *CLUSTER*
          (SRC_ITEM      HEADER           1  FUNC get_class)
          (DEST_ITEM     C++_SRC_CODE     1  FUNC get_memfuncs)
  )
  % Link from CLUSTER node at 'DOCUMENTATION:<' to NROFF_DOCs.
  (RELATE "Manual" *CLUSTER*
          (SRC_ITEM      *CLUSTER*    1  FUNC cluster_link
                                             "DOCUMENTATION:<")
          (DEST_ITEM     NROFF_DOC      m  FUNC workproduct_link
                                             "DOCUMENTATION:<")
   )
  % Link sorted keyword list entries to equivalent keywords in
  % CONTRIBUTIONS.
  (RELATE *VARIABLE* *GLOBAL*
         (SRC_ITEM      KEYWORD_LIST      1
                    PROGRAM $gmm/Admin/keyword-list-linkspots.ksh)
         (DEST_ITEM     CONTRIBUTIONS     m
                     FUNC positioned_item_list_search "KEYWORDS:")
   ))
```

Figure 13. Linking Section Example

Some key points to notice are:

In the first node-relator, a one-to-many relationship is set up between workproducts read in by `HELP_DOC` action-calls. The src-item and dest-item both use `markup_linkspots`, which is passed two arguments besides the node being manipulated.

In the second node-relator, the class declaration workproducts read in by `HEADER` action-calls are linked to C++ source code workproducts. Notice that this is a 1-to-1 relationship—there can only be one header file and one source code file.

The third node-relator links the DOCUMENTATION:< field-placeholder within cluster nodes to the actual manual page they reference. The src-item is handed cluster nodes while the dest-item if given document workproducts.

In the fourth node-relator, each keyword in a list of sorted keywords (KEYWORD_LIST) is linked to a CONTRIBUTIONS workproduct that contains an equivalent keyword. The src-item uses a *Unix ksh* script program extractor called keyword-list-linkspots.ksh. The link-role is *VARIABLE*, thus the role of the links created will be the identifier of each matching linkspot.

5.4 Detailed Linking Example

Let's put all this feature extractor information together in a detailed example by extending the Canvas example, from section 4.1 on page 24. We will start by adding another component called Interactor whose manual page references the Canvas manual page (as in Figure 2 on page 8). These manual pages will be linked using the node-relator in the see also example (see Figure 12 on page 33). For a definition section, the one from Figure 9 on page 25 will be used. Together, this produces:

```
(DEFINITIONS
   (CLUSTER_MAIN ROLE:workproducts HEADER: Yes IN: $rge/doc
                  TEMPLATE: $rge/doc/interviews-template)
   (NROFF-DOC    IN:$rlsi/nroff-doc  OUT:/tmp/test ROLE:doc)
   (C++-SRC-CODE IN:$rlsi/src        OUT:/tmp/test ROLE:src)
   (HEADER       IN:$rlsi/headers    OUT:/tmp/test ROLE:header)
)


(LINK
   % Link SEE ALSO manpage item references to the actual manpages
   % they reference.
   (RELATE "see_also" *GLOBAL*
           (SRC_ITEM  NROFF_DOC m FUNC see_also_linkspots)
           (DEST_ITEM NROFF_DOC 1 FUNC smart_node_name_linkspots
                                                   "True")
   ))
```

```
(STRUCTURES
    (:CLUSTER ("Canvas")
                (NROFF-DOC "Canvas.3I")
                (C++-SRC-CODE "X11-canvas.c")
                (HEADER "canvas.h")
    )
    (:CLUSTER ("Interactor")
                (NROFF-DOC "Interactor.3I")
                (C++-SRC-CODE "interactor.c")
                (HEADER "interactor.h")
    ))
```

Figure 14. Canvas and Interactor Clusters with See Also Linking

Also assume the contents of the `Interactor.3I` manual page contains:

```
...
SEE ALSO
  Bitmap (3I), Canvas (3I)
```

And the `Canvas.3I` manual page contains:

```
...
SEE ALSO
   Interactor (3I), Painter (3I)
```

When Cost++ is executed with this data, it will first read the definition and linking sections, followed by reading the structuring section. Within the structuring section, it first executes the instructions for building the `Canvas` cluster. During this time, when the action-call to `(NROFF-DOC "Canvas.3I")` is executed, it creates a node for the manual page workproduct `$rlsi/nroff-doc/Canvas.3I`. Because both the src-item and dest-item of our node-relator have a where-look of `NROFF-DOC`, this node will be handed to both function extractors. `see_also_linkspots` finds the see also section of the manual page and returns information about the two see also references. This takes the form of two `LinkSpot` objects, the first having an identifier of `Interactor`, position `14,317`,[13] and a node of `Canvas.3I,` and the second having an identifier of `Painter`, position `14,334`, and node `Canvas3.I.` `smart_node_name_linkspots` is also run on this node. It returns the root name of the manual page being scanned. In this case, it produces one `LinkSpot` object with identifier `Canvas,` position `0`, and node `Canvas.3I`. After this point, the other workproducts for the `Canvas` cluster are read in and the `Canvas` cluster will be complete.

13. We are assuming an arbitrary position for the see also section within the manual pages. In this case, the 'I' in Interactor (3I) is at position 14,317 within the file.

Next, Cost++ executes the instructions for building the `Interactor` cluster, and in a similar way, it runs the function extractors over the node `$rlsi/nroff-doc/Interactor.3I`. When the `see_also_linkspots` extractor is run, it returns two `LinkSpot` objects—one with identifier `Bitmap` at position `1334` and one with identifier `Canvas` at position `1347`. `smart_node_name_linkspots` returns one `LinkSpot` object with identifier `Interactor` and position `0`.

At this point, we have four source linkspots and two destination linkspots for the node-relator (see Figure 15).

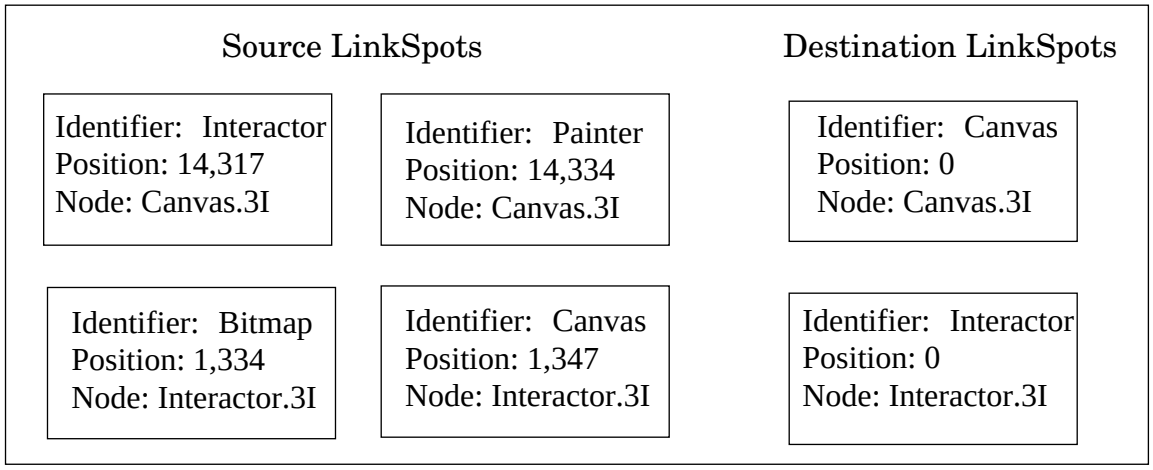


Figure 15. Canvas and Interactor Linkspots for See Also

After all the statements in the structuring section have been executed, Cost++ will attempt to link the manual pages according to the linkspots generated. This occurs after all nodes have been created because the link-time of our node-relator is `*GLOBAL*`. Because we have a many-to-1 relationship, there can be many source linkspots with the same identifier, but only one destination linkspot with the same identifier. Links are created when a source linkspot identifier matches a destination linkspot identifier. In our case we have two—linkspots with identifier `Interactor` and `Canvas`. The source of each link is the node associated with the source linkspot along with this linkspot's position. Similarly, the destination of the link is the node associated with the destination linkspot along with this linkspot's position. This leads to the linked structure shown in Figure 16.

After performing the linking, the six `LinkSpot` objects will be deleted.

If the node-relator had its link-role changed to `*VARIABLE*`, the role of link #1 would be `Canvas` and the role of link #2 would be `Interactor`, instead of `see_also`.

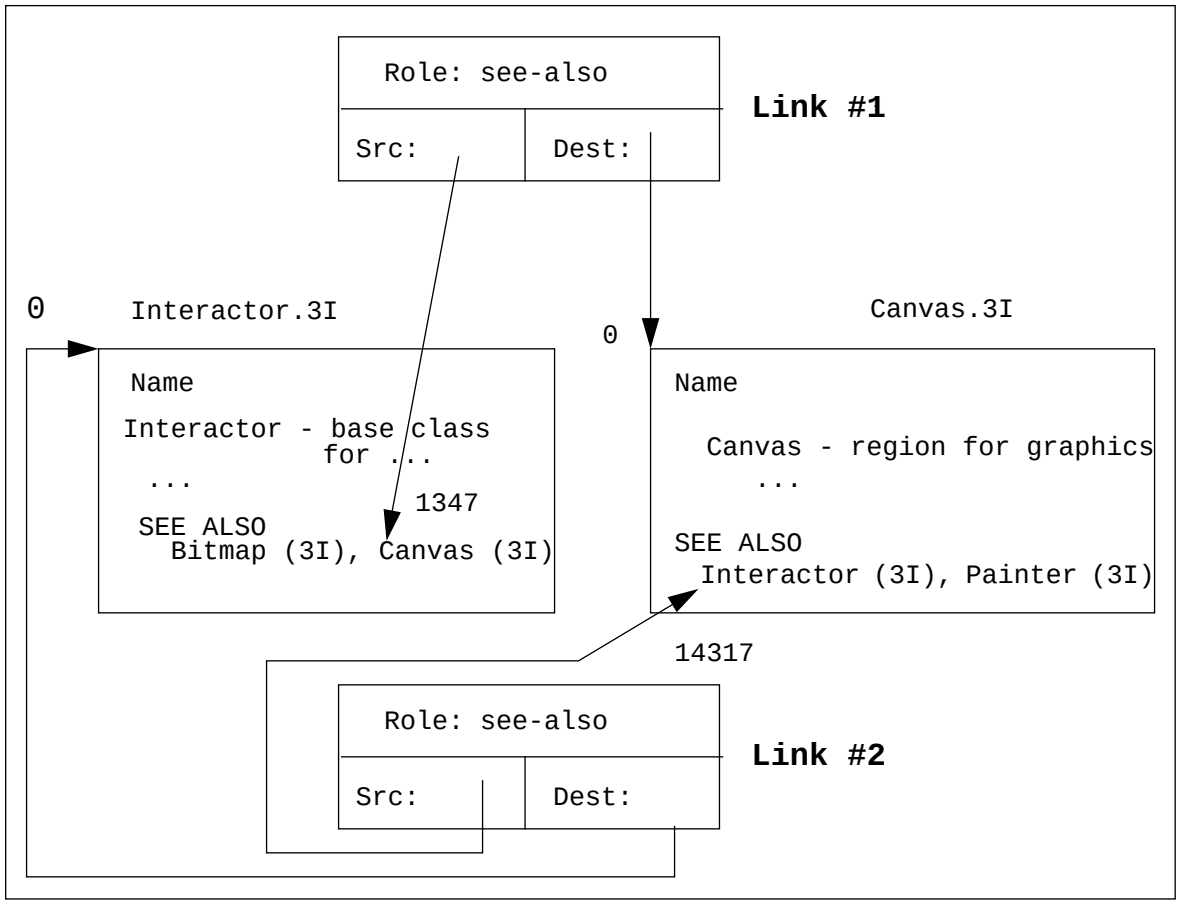


Figure 16. Canvas and Interactor Link Structure after See Also

6 Cost++ Post-processing Section

The post-processing section defines operations performed on nodes after all linking and other operations have been performed. Some examples of such operations include:

- Removing markup language terms, like `@LINK-REF` (see section 7.4.1.2 on page 56) from documents after they've been linked.
- Finding all nodes that have no links and linking them to a special node.
- Removing links between nodes.
- Content-based addition of value links.

The operations to perform are defined by *post-processors*—functions, programs, and scripts that are given nodes to manipulate. The nodes handed to post-processors can range from one specific node, to all memory resident nodes.

6.1 Syntax and Terminology

Let's start with the example of removing markup language terms:

```
(POST-PROCESS
      % FORM: remove_markup_instructions <remove-pattern>
      %                                  <remove-pattern-delimiters>
      %                                    <remove-all>
      (*ANY* FUNC remove_markup_instructions "@TERM-DEF" "True"
                                                        "False")
)
```

Figure 17. Removing Markup Language Terms

Similar to the other sections we have investigated, the post-processing section is wrapped in a set of parentheses where the opening paren must be followed by the word POST-PROCESS. In between these points are zero or more *pp-descriptors*. In our example, there is one:

```
      (*ANY* FUNC remove_markup_instructions "@TERM-DEF" "True"
                                                        "False")
```

Each node-relator describes exactly which nodes should be handed to a post-processor, along with any arguments. Very similar to feature extractors, post-processor functions, called *pp-functions,* are compiled into the Cost++ program; and post-processor programs and scripts, called *pp-programs,* are run in a separate process. Each pp-descriptor has the form:

```
  (<pp-where-look> <pp-type> <pp-name> [<pp-arg>]*)
```

where these characteristics are defined to be:

pp-where-look—which nodes will be handed to the post-processor. Legal values include:

ANY—all memory resident nodes are handed to this post-processor—this includes all nodes created or read in during the execution of Cost++.

WORKPRODUCT—hand only workproduct nodes to the post-processor.

STRUCTURE—only structure nodes (clusters and classification nodes) are handed to the post-processor.

CLUSTER—hand only cluster nodes to the post-processor.

CLASS—hand only classification nodes to the post-processor.

<node-pathname>—a string representing the full pathname of a node to hand to the post-processor. If this specific node was read in during

the execution of Cost++, it is handed to the specified post-processor. If not read in, no action takes place.

`*ANY*` is the value used in the markup language example, above. Thus, all nodes will be handed to the post-processor.

pp-type—the type of post-processor to call. It specifies whether the post-processor, pp-name (see below), is a pp-function or a pp-program. A value of `FUNC` specifies a pp-function where the function must be defined as part of the Cost++ program. A value of `PROGRAM` specifies a pp-program that is a separate program or shell script to execute.

In the markup language example, the pp-type is `FUNC`, specifying the use of a pp-function.

pp-name—the name of the actual pp-function or pp-program. When pp-type is `PROGRAM,` this field represents the full pathname of a script or program to execute. When pp-type is `FUNC`, this field is interpreted as the name of a function to call within the Cost++ program. The nodes given to these post-processors are determined by the value of pp-where-look.

In the markup language example, the pp-name of the pp-function is `remove_markup_instructions`.

pp-arg—an optional string argument. Up to 16 can be given to a post-processor and are passed as the 2nd through 17th arguments. These help the post-processor perform different behavior based on attributes found in the config file.

In the markup language example, we have three pp-args: `"@TERM-DEF"`, `"True"`, and `"False"`.

Each node is considered for handing to post-processors after all other Cost++ operations, except saving out the lattices of nodes and links (see stage 5 in section 2.2 on page 9). For each pp-descriptor whose pp-where-look matches a given node, the post-processor defined by pp-name is called, handing it the arguments specified by the pp-args. Post-processors are executed on this node in the order they are defined. Note that the ordering of pp-descriptors is significant because side effects can occur—if one post-processor removes a node the next post-processor will not see this node. Another type of side effect can take place when several config files are simultaneously given to Cost++, since pp-descriptors for one config file can affect nodes from a previous config file. This occurs because all nodes read in become memory resident and pp-descriptors with a pp-where-look of `*ANY*` will execute on nodes read previously.

Using the terminology just defined, let's revisit the markup language example. In this case, the pp-descriptor will cause the post-processing function, `remove_markup_instructions`, to be called on every node where it will also be handed the three arguments; `"@TERM-DEF"`, `"True"`, and `"False".` All occurrences

of the markup language instruction `@TERM-DEF`, along with the delimiters surrounding the instruction keyword, are removed from each node.

7 Advanced Topics

This section contains descriptions of additional features that can improve the readability of config files as well as the speed at which they are produced. It also includes information for advanced users who wish to further customize Cost++. Some of these customizations require source code modification and recompilation of Cost++. See section 8.1 on page 70 for details on source code availability and installation.

Topics are presented in terms of their corresponding config file sections. The end of each section includes a BNF that specifies the exact syntax for that section. Recall that in BNF notation, a '+' means 1 or more occurrences of, and a '*' means 0 or more occurrences of. The following low-level symbols are used throughout these BNF specifications:

```
<string-or-symbol>    ::= <string> | <symbol>
<string>              ::= <A set of characters between a pair of
                          double-quotes.>
<K-shell-wildcard-string> ::= <A string where certain symbols
                            (like '*') are interpreted as an
                            argument processed by the Unix
                            K-shell.>
<symbol>              ::= <A set of contiguous non-whitespace
                           characters, not starting with a
                           double-quote.>
<func-symbol>         ::= [a-z | A-Z | 0-9 | _]+
<full_pathname>       ::= a <symbol> that represents the relative
                          or absolute pathname of an existing file.
<yes-or-no>           ::= Yes | No <case insensitive>
```

7.1 General Features

7.1.1 :INCLUDE statement

The `:INCLUDE` statement improves readability of config files by allowing sections of config files to be stored as separate files. It also allows config file information to be shared, improving config file development when several similar files exist.

The `:INCLUDE` statement can be placed anywhere a normal config file section can be placed, thus, it can be found where any definition section, linking section, post-processing section, or structuring section is found. Additionally, `:INCLUDE` statements can be used to replace any structure statement within the structuring section. For example, we could replace the following config file:

```
(DEFINITIONS ...)
(LINK ...)
(POST-PROCESS ...)
(STRUCTURES
  (:CLASS ("Latin" ROLE:language)
    (:CLASS ("Miles" ROLE:variant)
      (:CLASS ("Myles"))
    ))
 (:CLASS ("English" ROLE:language)
   (:CLASS ("Myles"))
 ))
```

with:

```
(:INCLUDE "~/definition-section")
(:INCLUDE "~/link-section")
(:INCLUDE "~/post-process-section")
 (STRUCTURES
    (:CLASS ("Latin" ROLE:language)
      (:CLASS ("Miles" ROLE:variant)
        (:INCLUDE "~/Miles-variants")
      ))
    (:CLASS ("English" ROLE:language)
      (:INCLUDE "~/Miles-variants")
    ))
```

along with four other config files that house the definition, linking, and post-processing sections shown above, along with `(:CLASS("Myles"))` stored within the `~/Miles-variants` file.

The `:INCLUDE` statement takes one string argument that represents the full pathname of the file containing config file instructions. During execution, Cost++ acts as though the `:INCLUDE` statement were replaced with the contents of this file. These statements can themselves contain `:INCLUDE` statements, leading to their arbitrary nesting.

7.1.2 General config file BNF

The general form of config files is:

```
<config-file>    ::= [<setup-sections>] <structuring-section>
<setup-sections> ::= <setup-sections> [<definition-section>]* |
                                      [<linking-section>]* |
                                      [<pp-section>]*
```

Where the various section symbols (e.g., `<definition-section>`) are defined below.

7.2 Structuring Section Features

7.2.1 :SET statement

The :SET statement changes action-descriptor settings from within the structuring section. It is useful when two or more groups of structuring section statements use the same settings. For example, we might have a group of structuring section statements whose result is to be saved in one location and another group of statements whose result is to be saved in a different location. An alternative example is two groups of statements that build structure nodes, where each group uses a different template file. This situation occurs in the Codelibs software library of components, where one cluster of components is C++-based and is built with a C++ template file, while the other cluster is C-based and built using a C template file, as in:

```
(DEFINITIONS
  (CLUSTER_MAIN TEMPLATE: $rge/doc/C++-template)
)
(STRUCTURES
  % Here are a bunch of Codelibs components that use the
  % C++ cluster node template.
  ...
 (:CLUSTER ("String++"))
  ...
  % Now we want build some C Codelibs components that use the
  % C template.
 (:SET CLUSTER_MAIN TEMPLATE: $rge/doc/C-template)
 ...
 (:CLUSTER ("Stringx"))
 ...
)
```

The `String++` cluster is built using the `C++-template` file and the `Stringx` cluster is built using the `C-template` file. Note that instead of using the `:SET` statement, we could have added a `TEMPLATE` structure-option after each cluster listed. However, for many clusters, this would have been more cumbersome and adversely affect readability.

7.2.2 Option scoping

We have discussed three ways of setting options—using action-options, structure-options, and the :SET statement. Having these three ways to set the same option leads to the need to define their *scope*—when and how long the value of options are defined for each method.

Setting the value of an action-option using the definition section or using the :SET statement have a global scope. These values will persist until another statement changes their value, or until the end of the config file. Structure-options persist for the duration of the structure statement where they are found—including all child structure statements. After this statement, the value of the option reverts back to its previous value. Let's now consider an example showing the scoping of all three ways of setting the value of the ROLE action-option:

```
(DEFINITIONS
   (CLASS_MAIN ROLE:language)
   (PET-FORMS  ROLE:pet-form)
)
(STRUCTURES
  (:CLASS ("Languages")
    (:SET CLASS_MAIN ROLE:latin)
    (:CLASS ("Latin")
      (:CLASS ("Marcus" ROLE:variant)
        (:CLASS ("Marc")
                (PET-FORMS "Marcel"))
      )
      (:CLASS ("Miles")
        (:CLASS ("Myles"))
      )))
  (:CLASS ("Masculine_Names")
    (:CLASS ("Michael"))
  ))
```

Figure 18. Config File ROLE Scoping for First Names

This leads to the linked structure seen in Figure 19. Note that the role switches from language to latin after the :SET command. However, the explicit use of the ROLE structure-option for Marcus causes the link to Marc to have role variant. Once the Marcus structure statement is complete, the role flips back to its previous value—latin for the remainder of the config file fragment. Notice that even though the Marcel workproduct node is read under the Marcus node, its role is pet-form instead of variant. This is because the ROLE action-option from the PET-FORMS action-descriptor takes precedence and applies only to this workproduct (see section 4.1 on page 24).

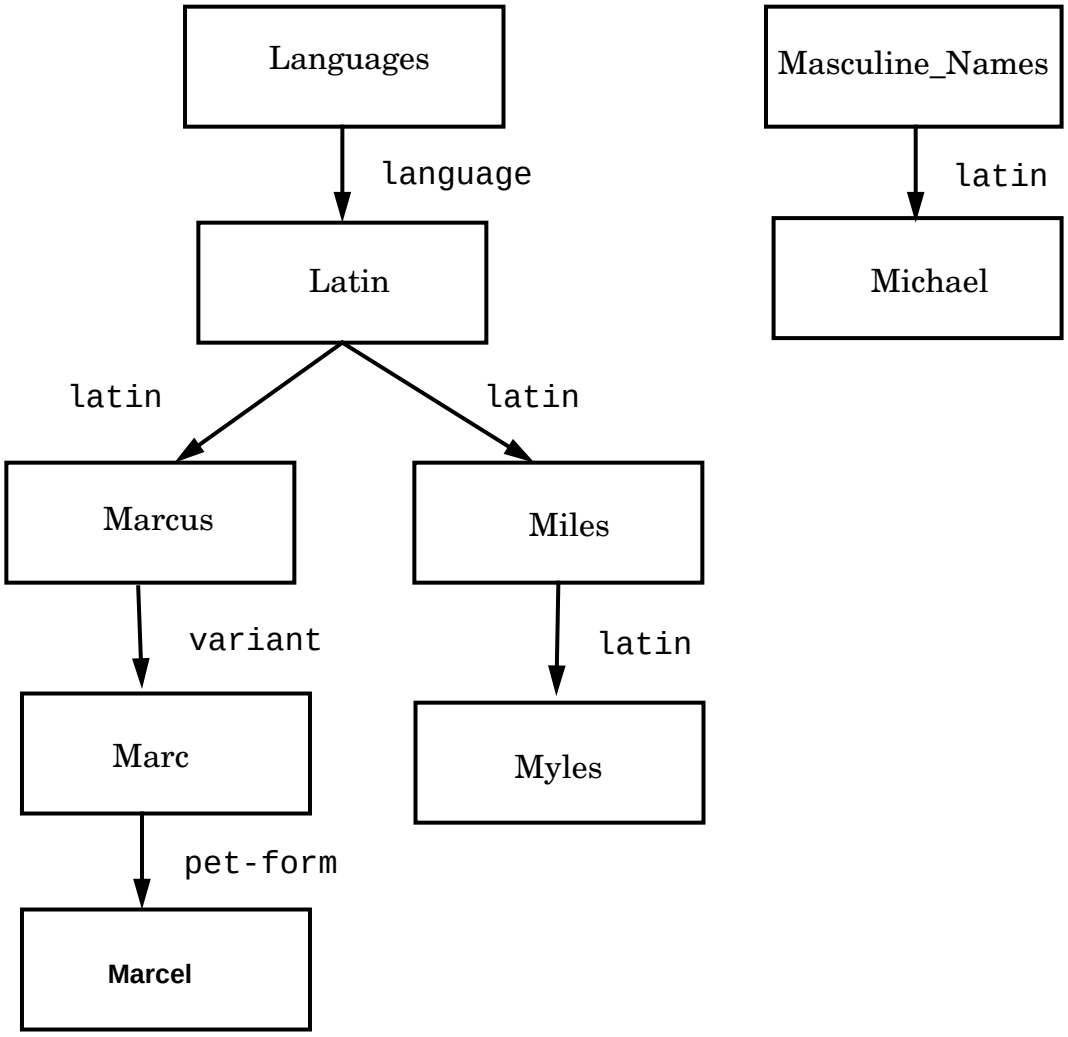


Figure 19. Scoping the `ROLE` action-option for First Names

7.2.3 Structuring Section BNF

The exact syntax of the structuring section is defined by the following BNF:

```
<structuring-section> ::= (STRUCTURES [<structure-statement>]*])
 <structure-statement> ::= <cluster-statement> |
                           <classification-statement>
 <cluster-statement> ::= (:CLUSTER <cluster-action-call>
                                   [<action-call>]*
                                   [<structure-statement>]*)
 <classification-statement> ::=
                            (:CLASS <classification-action-call>
                                   [<action-call]*
                                   [<structure-statement>]*)
 <cluster-action-call> ::= ([CLUSTER_MAIN] <cluster-name>
                                          [<structure-option>]*)
 <classification-action-call> ::= ([CLASS_MAIN]
                                           <classification-name>
                                          [<structure-option>]*)
 <action-call> ::= (<action-name> [<arg>]*)
<structure-option>  ::= <structure-opt-root> | <struct-opt-role>
 <structure-opt-root> ::= IN:<string-or-symbol> |
                          OUT:<string-or-symbol> |
                          HEADER:<yes-or-no> |
                          TEMPLATE:<string-or-symbol> |
                          INBOUND:<yes-or-no> |
                          GENERATE-WHEN-MISSING: <yes-or-no> |
                          <struct-opt-node-type>
 <struct-opt-role> ::= ROLE:<struct-opt-role-value>
 <struct-opt-role-value> ::= <string-or-symbol> | UNDEFINED
 <struct-opt-node-type> ::= NODE-TYPE:<struct-opt-n-t-value>
 <struct-opt-n-t-value> ::= <string-or-symbol> | UNDEFINED |
                            *DEFAULT*
 <cluster-name>        ::= <string>
 <classification-name> ::= <string>
 <action-name>         ::= <symbol>
 <arg>                 ::= <K-shell-wildcard-string>
```

7.3 Definition Section Features

7.3.1 Defining new parse functions

Sometimes it is necessary to extend Cost++ by building customized parse functions that interpret action-calls within the structuring section in a special way. Adding a new parse function requires modifying the Cost++ program. What follows is a brief description of how to add new parsing functions. For details, see the source code.

As Cost++ reads the definition section, it builds a C++ `Action_descriptor` object that stores all the information for each action-descriptor read. These are stored, along with all other state information, in a C++ `C_state` object. All parse functions are member functions of this `C_state` object. Thus, `dir_assist_general_node_parse_func` is really `C_state::dir_assist_general_node_parse_func.` These parse functions are found in `$rgs/C_state.C`.

When reading the structuring section and seeing an action-call, the action-name of the action-call is used to retrieve the parse function from its corresponding `Action_descriptor` object. This parse function is handed this `Action_descriptor` object and the structure node being created. The action-arguments of the action-call are read and parsed by the parse function using member functions defined for the `C_state` object. As an example, let's revisit part of the `Canvas` example from Figure 10 on page 30:

```
(DEFINITIONS
   (CLUSTER_MAIN ROLE:workproducts HEADER: Yes IN: $rge/doc
                 TEMPLATE: $rge/doc/interviews-template)
   (VALUE_LINK     FUNC:value_link_parse_func)
)
(STRUCTURES
   (:CLUSTER ("Canvas")
             (VALUE_LINK "OS" "HP-UX/7.0")
   )
)
```

When the action-call to the `VALUE_LINK` action-descriptor takes place, the parse function `C_state::value_link_parse_func` will be invoked. It will be passed the `Action_descriptor` for `VALUE_LINK` and the cluster node for `Canvas`. This parse function is then responsible for reading and interpreting the action-arguments, in this case, `"OS"` and `"HP-UX/7.0"`. Parse functions have the form:

```
   void C_state::<func-name> (Action_descriptor *descriptor ,
                              Node *structure_node ,
                              FILTER_FUNC *filter_func)
```

`<func_name>` is the name of the parse function being defined. `filter_func` is a pointer to a function that determines which links are read when new nodes are read in. See source code for details. To build a new parse function, you must:

1. Write a new parse function in `$rgs/C_state.C` and declare it in `$rgs/C_state.h`.
2. Register this function with Cost++ by adding an entry in the member function `C_state::_build_C_state`, of the form:

```
parse_func_database->register_exec_func ("<func-name>",
                                         C_state::<func-name>)
```

This will allow Cost++ to identify an action-descriptor with a `FUNC` action-option that matches your new parse function.

3. Add an action_descriptor to a config file with the `FUNC` action-option having the value that is the name of the new parse function.
4. Add action-calls within the structuring section that refer to your new action-descriptor.

For more details, see `C_state.C`.

7.3.2 Definition section BNF

```
<definition-section>    ::= (DEFINITIONS [<action-descriptor>]*)
<action-descriptor>     ::= (<action-name> [<action-option>]*)
<action-descriptor>     ::= (*GLOBAL* [<global-option>]*)
<action-name>           ::= CLASS_MAIN | CLUSTER_MAIN | <symbol>
<action-option>         ::= <structure-opt-root> |
                            <action-opt-role> |
                            ARGS:<num-args> |
                            FUNC:<func-symbol> |
                            READ-LINK-FILES:<yes-or-no>
<global-option>         ::= DEBUG:<yes-or-no> |
                            REMOVE-CLUSTER-EMPTIES:<yes-or-no> |
                            REMOVE-CLASS-EMPTIES:<yes-or-no> |
                            KEEP-LINKS:<links-to-keep>
<action-opt-role>       ::= ROLE: <action-opt-r-value>
<action-opt-r-value>    ::= <string-or-symbol> | UNDEFINED |
                            *DEFAULT*
<links-to-keep>         ::= NONE | USER-GENERATED | ALL
<num-args>              ::= -1 | [0-9]+
```

7.4 Linking Section Features

This section will help you determine when and how to write new extractors for performing content-based linking. The first subsection describes the feature extractors that are available. You should become familiar with this section whenever you are seeking to do content-based linking, since it is much more efficient to use existing extractors than to write your own. When the functionality you need doesn't exist in any of the extractor libraries, you will have to create a new extractor. The second and third subsections give high-level descriptions of when and how to write program and function extractors.

7.4.1 Feature extractor libraries

Currently, only libraries of function extractors are available—no libraries of extractor programs have been built. Existing function extractors have been separated into conditionally compilable libraries, where the default version of Cost++ comes with all libraries. The following description of function extractors has been separated into these different libraries. A brief description of each library is given, followed by details about the function extractors available. Each function description starts with the name of the extractor, followed by any required arguments. Each description is then followed by a ⇒ symbol and the word *General*, *Medium*, *Specific*, or *Very Specific;* to inform you of the generality of this extractor.

7.4.1.1 Function extractors found in `$rgs/general_feature_extractors.C`

These functions are all general extractors used by many different applications.

`FUNC positioned_item_list_search <search-pattern>[<dependency>] [<link-to>]⇒General`

This extractor is very useful for setting up links to comma-separated items that follow defined fields within nodes. For example, we might link a mail message with given keywords to other nodes these keywords refer to.

DETAILS

This extractor finds the first occurrence of `<search-pattern>` that begins at the beginning of a line. The entries following it are then analyzed as a set of comma separated items that may be found on one or more following lines. The end of comma-separated items is determined by a blank line (only whitespace). It is legal to have only one item, and the last item of the set does not require a trailing comma. Linkspots are returned for each item found, where the position returned is the start of the item. The identifier returned is the lower-cased content of the item—disregarding leading and trailing whitespace. This extractor also maps underscores in items to spaces.

If no occurrence of `<search-pattern>` is found, no linkspots are returned.

Sometimes linking a node is dependent on whether it does or does not contain another field. For example, we might have a bug report node that contains a `BUG FIXED BY` field. When this field is empty, the bug has not been fixed, and we would like to link its keywords with an `outstanding-bugs` link role. When the field is complete, the bug has been fixed, and we would like to link its keywords with a `fixed-bugs` link role. To perform this, we must be able to tell whether the `BUG FIXED BY` field has been filled in or not. The `<dependency>` option exists to handle this situation. It has the form:

```
"EXCEPT" <field-pattern> | "WITH" <field-pattern>
```

<field-pattern> is a string that represents the name of a field, similar to <search-pattern>. The "EXCEPT" form only allows linkspots to be returned when <field-pattern> does not exist, or represents a blank field. The "WITH" form performs the opposite—it will only allow linkspots to be returned when <field-pattern> exists and is followed by one or more items.

If you need to link at a different location within the node then at the beginning of each item, you may want to use the <link-to> option. For example, we might have a set of book bibliography nodes that contain a Title field and a Reviews field. We would like to link book review nodes to their corresponding bibliography nodes when their Title fields match. However, the position we would like to link to in the bibliography node is the Reviews field. <link-to> has the form: "LINK-TO" <link-to-pattern>. It causes the location following the first occurrence of <link-to-pattern> to be the position returned for each linkspot. If <link-to-pattern> is not found within the node, a warning message is issued, and no linkspots returned.

EXAMPLES

We might have a mail message containing the fragment:

```
...
KEYWORDS: Jumping Fish, Moosefish
...
```

If positioned_item_list_search "KEYWORDS:", were handed this mail message, it would generate linkspots with identifier jumping fish and moosefish with positions located at the J in Jumping Fish and the M in Moosefish, respectively. The same result would be achieved with:

```
KEYWORDS: Jumping_Fish, Moosefish
```

As another example, we might have a bug report containing the fragment:

```
...
KEYWORDS: Jumping Fish, Moosefish
BUG FIXED BY: Mike
...
```

If positioned_item_list_search "KEYWORDS:" "EXCEPT" "BUG FIXED BY:", were handed the above bug report, it would return no linkspots. Whereas, positioned_item_list_search "KEYWORDS:" "WITH" "BUG FIXED BY" would return the linkspots specified in the first example.

```
FUNC smart_node_name_linkspots [<dumb-mode>][<raw-mode>]
                               [<link-to>]⇒General
```

This extractor is useful for globally referencing other nodes, such as linking the see also section of manual pages to the actual manual pages they reference.

DETAILS

A linkspot is returned that has position `0` and an identifier that is the root name of this node. This root name consists of the filename of the file (with no path information) with suffix information stripped off. Thus, a node like `$rlwi/doc/Interactor.3I` would have an identifier of `Interactor`.

When `<dumb-mode>` is not specified or is `False`, the linkspot identifier returned will be down-cased and underscores will be replaced with spaces. If `<dumb-mode>` is `True`, no special processing of the root name is performed to determine the identifier. This mode can be useful for similarly named nodes that differ in case.

When `<raw-mode>` is not specified, the linkspot identifier will *not* contain the dot ('.') or suffix information. When `RAW` is specified for `<raw-mode>`, the linkspot identifier will contain the suffix information and the dot. This is useful when suffix information is important, like for nodes that are the names of authors.

`<link-to>` is used to specify a different linkspot position. See, `positioned_item_list_search`, for details.

EXAMPLES

For the following examples, calls to this extractor with the given arguments, when handed nodes with the given pathname, will return a linkspot with the specified identifier:

```
FUNC smart_node_name_linkspots
     → $rlwi/hubs/Stringx.3x → stringx
FUNC smart_node_name_linkspots "False"
     → $rlwi/hubs/Andreas_Paepcke → andreas paepcke
FUNC smart_node_name_linkspots "False" "RAW"
     → $rlwi/hubs/Stringx.3x → stringx.3x
FUNC smart_node_name_linkspots "True"
     → $rlwi/hubs/Stringx.3x → Stringx
FUNC smart_node_name_linkspots "True"
     → $rlwi/hubs/Andreas_Paepcke → Andreas_Paepcke
FUNC smart_node_name_linkspots "True" "RAW"
     → $rlwi/authors/Cox_B. → Cox B.
```

`FUNC cluster_link <id>⇒General`

Used to link from field-placeholders within cluster nodes to workproducts. This extractor is usually used in conjunction with `workproduct_link`.

DETAILS

A cluster node is searched for the field-placeholder <id> in a cluster node and a linkspot is returned with identifier <id>, and a position that is the first position following the field-placeholder within the cluster.

EXAMPLE

If a cluster node that contains a field-placeholder of the form:

```
DOCUMENTATION:<+>
```

is handed to this extractor through a call of the form:

```
FUNC cluster_link "DOCUMENTATION:<"
```

This extractor would return a linkspot with identifier DOCUMENTATION:< and position that is the location of the + sign within the field-placeholder of the cluster.

`FUNC workproduct_link <id>⇒General`

This is used to globally link workproduct nodes from cluster node field-placeholders. It is often used in conjunction with the `cluster_link` extractor.

DETAILS

This extractor simply returns a linkspot with position `0`, identifier <id>, and a node that is the node handed to this extractor.

EXAMPLE

A call to this extractor of the form:

```
FUNC workproduct_link "DOCUMENTATION:<"
```

when handed the node `~/workproduct1`, will return a `LinkSpot` with identifier DOCUMENTATION:<, position `0`, and node `~/workproduct1`.

7.4.1.2 Function extractors found in `$rgs/doc_feature_extractors.C`

Extractors in this library are used for building hypertext documentation—cross-document linking of terms, table of content files, etc.

`FUNC markup_linkspots <markup-pattern> <remove-keyword>⇒Medium`

This extractor is used to link keywords wrapped within markup language commands. Thus, it can help with linking references to terms in a document to their definitions in another document. It is usually used in conjunction with the `remove_markup_instructions` post-processing function (see section 7.5.1.2 on page 67).

DETAILS

All markup commands have the form:

```
<markup-pattern>‘<keyword>’
```

Linkspots are returned for each markup language command matching `<markup-pattern>` found within a node. The identifier used for each linkspot is the quoted keyword following `<markup-pattern>`. If `<remove-keyword>` is `False`, the position of each linkspot is the position of the first character of the keyword within the node. If `<remove-keyword>` is `True`, the position will be the first character of the markup instruction—with the assumption the markup instruction and keyword will be removed (see section 7.5.1.2 on page 67)—making the effective position the next character following the markup command.

EXAMPLES

Given the text:

```
...use @LINK-REF‘Component Browser’ to find...
```

The following calls to `markup_linkspots` lead to the linkspot shown after the → symbol.

`FUNC markup_linkspots "@LINK-REF" "False"` → linkspot with identifier `Component Browser` and position at the `C` in `Component Browser`

`FUNC markup_linkspots "@LINK-REF" "True"` → linkspot with identifier `Component Browser` and position at the `@` in `@LINK-REF.`

`FUNC help_file_linkspots⇒Specific`

Generates linkspots for the Kiosk help file section, subsection, subsubsection, and detail entries. It is used to link to the corresponding table of contents sections.

DETAILS

Entries have the form:

```
[num.num.num.num] <text>
```

as in:

```
[1.1] Introduction
```

See source file documentation for more details.

`FUNC toc_linkspots⇒Specific`

Generates linkspots for the Kiosk table of contents files that contain a set of section, subsection, subsubsection, and detail entries. It is used to link to the corresponding sections within help files.

DETAILS

Entries have the form:

```
[num.num.num.num] <text>
```

as in:

```
[1.1] Introduction
```

See source file documentation for more details.

`FUNC author_item_search <search-pattern>⇒Specific`

This extractor is used to link author references to author nodes. It generates linkspots for author reference nodes that have author names in last name first format; it can be used to link to author nodes by using it in conjunction with the `smart_node_name_linkspots` (with the `raw_mode` option) extractor.

DETAILS

This extractor analyzes the author items after `<search-pattern>` and parses them as author names of the form:

```
<last_name>,<first_and_other_initials>[, | \n]
```

A second comma, or the newline, act as a separator between author names. Authors may be on more than one line. A blank line is needed to separate the items of a `<search-pattern>`. Linkspots are returned with identifiers that are the author's names—excluding the comma separating last and first names—with a position that is the first character of the last name.

EXAMPLE

Given a node with contents:

```
...
AUTHOR: Griss, M., Cox, B.
...
```

If this node were handed to; `FUNC author_item_search "AUTHOR:"`, it would return two linkspots with identifiers, `"Griss M."` and `"Cox B."`, and with positions at the `G` in `Griss` and the `C` in `Cox`.

7.4.1.3 Function extractors found in `$rgs/software_feature_extractors.C`

Extractors found in this library link features in software, such as linking parent to child class definitions in C++ code. For more detailed descriptions of behavior, see the source code.

`FUNC get_class⇒Specific`

Returns linkspots that are the names of classes in class declarations within C++ header files. Each linkspot has a position that is the first character of the class name within the declaration and identifier that is the class name.

`FUNC get_parent_classes⇒Specific`

This returns linkspots that are the names of parent classes referenced through inheritance in class declarations within C++ header files. Each linkspot has a position that is the first character of a parent class name within the declaration and identifier that is the class name. This extractor is used in conjunction with the `get_class` extractor.

`FUNC get_friend⇒Specific`

This extractor is similar to `get_class`. It returns linkspots to all C++ friend declarations found within C++ header files.

`FUNC get_memfuncs⇒Specific`

This extractor is used to specify that a C++ source code file has a member function within it. Returns a linkspot with position `0` and identifier that is the class name of this member function.

`FUNC cluster_to_config_linkspots⇒Very Specific`

This extractor is used to link a cluster node's `FILES` field-placeholder to the `FILES` part of source code configuration files. It returns a linkspot with position at the placeholder and an identifier of `Config_files`. This extractor is used in conjunction with `config_to_cluster_linkspots`.

`FUNC config_to_cluster_linkspots⇒Very Specific`

This is used to link a cluster node's `FILES` field-placeholder to the `FILES` part of source code configuration files. It is used in conjunction with `cluster_to_config_linkspots`. It returns a linkspot with position at the `FILES` part of the configuration file with an identifier of `Config_files`.

`FUNC cluster_to_manpage_linkspots⇒Very Specific`

This extractor is used to link the different sections of the manual pages that describe software libraries to their corresponding field-placeholders within cluster nodes (see the example in section 3.1.2.2 on page 19). This extractor returns up to six linkspots that specify the position of the subsections of the `DOCUMENTATION` field-placeholder within software cluster nodes. Included are the locations of the `SYNOPSIS`, `DESCRIPTION`, `EXAMPLES`, `SEE ALSO`, `AUTHOR`, and `NOTES` field-placeholders. This extractor is used in conjunction with `manpage_to_cluster_linkspots`.

`FUNC manpage_to_cluster_linkspots⇒Very Specific`

This is similar to `cluster_to_manpage_linkspots` but returns linkspots to the actual start of each of the six listed subsections within the manual pages.

`FUNC see_also_linkspots⇒Specific`

Returns linkspots to see also references within a manual page, with identifiers that are the name of the reference and position that is the location of the first character of the reference name within the manual page. It works similar to the `positioned_item_list_search` extractor, with `<search-pattern>` equal to `SEE ALSO`. More of a discussion of this extractor can be found in section 5.4 on page 39.

7.4.1.4 Function extractors found in `$rgs/GMM_feature_extractors.C`

These extractors are very specific to the Group Memory Manager (GMM) application and to the Kiosk feedback mechanism. There is not enough space here to describe these extractors. See source code for details.

7.4.1.5 Function extractors found in `$rgs/SMI_feature_extractors.C`

Extractors in this library are very specific to the network management application. There is not enough space here to describe these extractors. See source code for details.

7.4.2 Writing new program extractors

This section describes how to write new program extractors when the functionality you need is not found in the extractor libraries. Such extractors are written when good performance is not critical, since they require no recompilation of the Cost++ program.

Cost++ executes a program extractor, by calling it in the form:

```
<extractor-name> <node-path-name> [<extractor-arg>]*
```

`<extractor-name>` is the script or program to execute. It is passed the full pathname of the node to analyze (`<node-path-name>`), along with up to 16 optional string arguments (`<extractor-arg>`). `<extractor-name>` passes linkspot information back to Cost++ by writing this information to standard output, where Cost++ reads it until the extractor terminates. Program extractors can return an arbitrary amount of linkspot information. The only restrictions on them are that they return *identifier-position* pairs, where first, an identifier (string) must be output on a separate line, followed by an integer position on a separate line. Once each pair is read, Cost++ will generate the appropriate `LinkSpot` object for the given node.

As an example, let's revisit the use of the program extractor `keyword-list-linkspots.ksh` discussed in Figure 13 on page 38. Assume we have the following definition, linking, and structuring sections:

```
(DEFINTIONS
  (KEYWORD_LIST IN:$gmm/lists OUT:$gmm/lists)
)
(LINK
    % Link sorted keyword list entries to equivalent keywords in
    % CONTRIBUTIONS.
   (RELATE *VARIABLE* *GLOBAL*
           (SRC_ITEM      KEYWORD_LIST      1
                   PROGRAM $gmm/Admin/keyword-list-linkspots.ksh)
           (DEST_ITEM     CONTRIBUTIONS     m
                    FUNC positioned_item_list_search "KEYWORDS:")
   ))
(STRUCTURES
   (:CLASS ("FOO")
           (KEYWORD_LIST "the_list")
   ))
```

along with the following contents for file `$gmm/Admin/keyword-list-linkspots.ksh`:

```
#!/bin/ksh
#
# Form: keyword-list-linkspots.ksh <keyword-list>
#
# This script is invoked through a Cost++ node-relator to return
# linkspots at the beginning of each word in a sorted list of
# words passed in the first argument. The identifiers returned
# consist of each line (lower-cased) with a position at the
# beginning of each line.


exec < $1 # Open the keyword list for reading
integer link_loc=0
while read -r line  # Read in each line of $1 into ‘line’.
do
  typeset -l line  # Lower-case line
 print "$line\n$link_loc"  # print line and first char position.
  link_loc=$link_loc+${#line}+1 # Add length of line to get
                                # position.
done
```

During stages 2a and 3a of the linking process (section 5.1 on page 32), Cost++ will call this program extractor as:

```
$gmm/Admin/keyword-list-linkspots.ksh "$gmm/lists/the_list"
```

For each line found within `$gmm/lists/the_list`, this extractor will print the lower-cased version of this line as a linkspot identifier and print the beginning of this line as a linkspot position. This information is then read by Cost++ to build the actual `LinkSpot` objects. If the contents of `$gmm/lists/the_list` were:

```
Browsers
CollectionPresenter
ComponentPresenter
Core Dump
Cost++
Dennis F. Freeze
Dumped Core
FileBrowser
```

The script would print:

```
browsers
0
```

for the first line and:

```
collectionpresenter
8
```

for the second line. Cost++ would then build appropriate linkspots from this information.

7.4.3 Writing new function extractors

This section describes how to write new function extractors when the functionality you need is not found in the extractor libraries. Such extractors are written when good performance is essential.

Similar to how a program extractor is called, a function extractor is passed a pointer to the node to analyze, followed by up to 16 optional string arguments. This gives function extractor calls the form:

```
LinkInfoList*  <extractor-name> (Node *the-node ,const char* arg,
                                 ...);
```

`the-node` is a pointer to the C++ `Node` object that houses the node to search for linkspots. `arg,` and any remaining arguments, are optional strings that the extractor can use.

Function extractors gather and return linkspots in `LinkInfoList` objects, which are arrays of `LinkInfo` objects. As mentioned in section 5.1 on page 32, the `LinkSpot` objects we've discussed are really simplifications of their true underlying data representation. `LinkInfo`s contain an identifier and an array of objects (true `LinkSpots`) that contain all the positions and nodes where linkspots have been found. In creating new function extractors, you need not worry about directly creating `LinkSpot` and `LinkInfo` objects. A set of higher-level functions in `$rgs/internal_link_sup.C` are used by most function extractors for this task.

Following is a high-level description of the steps in writing a new function extractor:

1. Ensure you have everything necessary to compile Cost++ (see section 8.3 on page 71 for details).
2. Write the body of your extractor, adding to an existing library of extractors, or creating a new library. Instructions for creating a new library are found at the top of all extractor library sources (e.g., `$rgs/general_feature_extractors.C`). Many operations an extractor performs (e.g., searching) are implemented through calls to the `Node` member functions of `the_node`. For a list of these functions, see `$rshs/Node.C` and `$rshs/Node.h`. Adding links to a `LinkInfoList` is done using the function `add_linkspot_to_list` in `$rgs/internal_link_sup.C`. When you wish to read more arguments passed to the function, the `varargs` facility must be used. See examples in the library sources and see the *Unix* manual page on `varargs`.
3. Register the new function extractor in the initialization routine for the library in which it belongs. For example, for `$rgs/general_feature_extractors.C`, this is the function `initialize_for_general_feature_extracting`. You will need to add a line of the form:

   ```
   sitable->register_exec_func ("<extractor-name>" ,
                                <extractor-name>)
   ```

   where `<extractor-name>` is the name of the new extractor. This will allow Cost++ to correctly identify a node-relator with an extractor-name that is the name of your new extractor.
4. After writing a new function extractor and registering it, recompile Cost++ (run `nmake` in directory `$rgs`).
5. Add the linking section node-relators to the config files where this function extractor will be used.

As an example of writing a new function extractor, consider the `workproduct_link` extractor from section 7.4.1.1 on page 53. This extractor takes one argument, `id`, and simply returns a linkspot with identifier `id` and position `0`. The code for this function, found in `$rgs/sofware_feature_extractors.C`, looks like:

```
LinkInfoList*  workproduct_link (Node *workproduct_node ,
                                 const char* id, ...)

 // This is used to globally link to workproduct nodes from a
 // cluster node template position. Simply return a global
 // link (position 0) with id.  Example: ‘id’ =
 // "HEADER:<" We would then return location 0 within
 // ‘workproduct_node’ along with the keyword “HEADER:<".


{
  if (DEBUG)
    {
      cout <<
      "  **DBG--Executing feature extractor‘workproduct_link’";
      cout << "on node ‘" << workproduct_node->file_name () <<
              "‘.\n" << flush;
    };
  LinkInfoList *return_lil = new LinkInfoList;
  add_linkspot_to_list (return_lil, workproduct_node, 0,
                        strdup (id));
  return (return_lil);
};
```

The `if (DEBUG)` section is used to print debugging information when the `-d` option is used in running Cost++ (see section 8.2 on page 70). A new `LinkInfoList` is created by the line containing `new LinkInfoList`. Linkspots are added to a `LinkInfoList` using `add_linkspot_to_list`, which takes an existing `LinkInfoList`, a `Node`, position, and identifier.

The interface description for this extractor is found in `$rgs/software-feature-extractors.h`, and looks like:

```
LinkInfoList* workproduct_link (Node *workproduct_node ,
                                const char *opt_attribute, ...);
```

The extractor is registered in the function `initialize_for_software_feature_extracting`, found in `$rgs/software_feature_extractors.C`. This is done with the line:

```
sitable->register_exec_func ("workproduct_link" ,
                             workproduct_link);
```

The use of this extractor in a node-relator can be seen in section 5.3 on page 37.

7.4.4 Linking section BNF

```
<linking-section>        ::= (LINK [<node-relator>]*)
<node-relator>           ::= (RELATE <link-role> <link-time>
                                     <src-item>
                                     <dest-item>)
<link-role>              ::= <string> | *VARIABLE*
<link-time>              ::= *GLOBAL* | *LATTICE* | *STRUCTURE* |
                             *CLUSTER* | *CLASS*
<src-item>               ::= (SRC_ITEM <where-look> <one-or-many>
                                       <extractor-type>
                                       <extractor-name>
                                       [<extractor-arg>]*)
<dest-item>              ::= (DEST_ITEM <where-look> <one-or-many>
                                        <extractor-type>
                                        <extractor-name>
                                        [<extractor-arg>]*)
<where-look>             ::= *STRUCTURE* | *CLUSTER* | *CLASS* |
                             *WORKPRODUCT* | <action-name>
<one-or-many>            ::= 1 | m
<extractor-type>         ::= PROGRAM | FUNC
<extractor-name>         ::= <func-symbol> | <full-pathname>
<extractor-arg>          ::= <string>
```

7.5 Post-Processing Section Features

This section will help you determine when and how to write new post-processors. The first subsection describes the post-processors that are available. When the functionality you need doesn't exist in any of the post-processor libraries, you will have to create a new post-processing program or function. Creating such pp-programs and pp-functions is very similar to creating function and program extractors for linking (see section 7.4 on page 52). The main difference is that post-processors are less complicated—they have no return value. The final two subsections give a high-level description of how to write such pp-programs and pp-functions in terms of the salient differences between writing feature extractors and post-processors.

7.5.1 Pp-function libraries

Pp-functions follow the same form as feature function extractors in that they exist in separate, conditionally compilable libraries. Their discussion will also follow the same format as function extractors (see section 7.4.1 on page 53).

7.5.1.1 Pp-functions found in $rgs/general_pps.C

These functions are all general post-processing functions used by many different applications.

```
FUNC add_value_links_based_on_search <role> <search-field-name>
                                        [<beginning-of-line>]⇒General
```

This pp-function adds value links to a node based on the node's contents.[14]

DETAILS

If the regular expression `<search-field-name>` is found within the node, a value link is created with role `<role>`, position `0`, and value that is the contents of the rest of the line following the match to `<search-field-name>`. Leading blanks will be stripped from the value along with the ending newline. If `<beginning-of-line>`[15] is `True`, `<search-field-name>` will only match at the beginning of a line.[16] If `<beginning-of-line>` is `False` or is missing, the pattern `<search-field-name>` can occur anywhere in the node.

EXAMPLE

In Kiosk, the symbolic name of a node corresponds to the value of a value link with role `DisplayName`. This is very useful for nodes that have non-descriptive file names. For example, *Unix mh* mail messages correspond to numbered files. To see more descriptive information about a mail message, we could have Cost++ generate symbolic names for mail messages that are the content of the subject field of each mail message with the pp-descriptor:

```
(*WORKPRODUCTS* FUNC add_value_links_base_on_search
                                    "DisplayName" " *Subject:"
                                                  "True")
```

```
FUNC remove_link <node-to-delete-links-to> [<role>]
                                             [<owner>]⇒General
```

Remove a binary link that connect this node to `<node-to-delete-links-to>`.

DETAILS

`<node-to-delete-links-to>` is the full pathname of the node. If `<role>` is given, the link must have the given role. If `<owner>` is given, the link must have the given owner.

EXAMPLE

14. In the future, adding value links in this way will be performed by the linking section.

15. The newline character cannot currently be passed into a Cost++ string, thus this extra parameter is needed.

16. Note that the way this is currently implemented, a match cannot occur on the first line of the node.

If we wanted to remove a link between node ~/tuna and node ~/fish that has the role fishy, we could add the pp-descriptor:

```
("~/tuna" FUNC remove_link "~/fish" "fishy")
```

FUNC remove_node⇒General

Remove this node from memory.

7.5.1.2 Pp-functions found in $rgs/doc_pps.C

This library contains post-processors specific to hypertext documentation markup language removal.

```
FUNC remove_markup_instructions <markup-pattern>
                                <delete-delimiters>
                                <delete-all>⇒Medium
```

This removes markup language instructions from nodes as discussed earlier in this section. For details on markup language instructions, see section 7.4.1.2 on page 56.

DETAILS

Given markup instructions that have the form:

```
<markup-pattern>'<key-phrase>'
```

This pp-function finds all markup instructions that match <markup-pattern> and deletes them. If <delete-delimiter> is True, the single quote delimiters surrounding the <key-phrase> are also removed. If <delete-all> is True, the delimiters and <key-phrase>s are removed.

EXAMPLES

The pp-descriptor:

```
(*WORKPRODUCT* remove_markup_instructions "@LINK-REF" "True"
                                                      "False")
```

would remove all occurrences of @LINK-REF within all workproducts read by Cost++. It would also remove the delimiters surrounding the markup language keywords. Thus:

```
...use @LINK-REF'Component Browser' to find...
```

would change to:

```
      ...use Component Browser to find...

FUNC add_help_menu_value_links <node> <role>⇒Very Specific
```

This function is useful for building hypertext on-line documentation help and is used in generating the on-line documentation for Kiosk. It generates value links for specific section headings found within a document. These value links are then used by Kiosk to “jump” to these sections when a user wants help.

DETAILS

This function searches a node for all document section headings of the form:

`\n[<n1>[.<n2>[.<n3>[.<n4>]]]] *<menu-item> menu for <menu-for>\n`

For each such heading found, it will make a value link with role `<role>`, a value of `<menu-for>:<menu-item>`, and an offset that is of the beginning of this section heading.

For example, if `node1` contains:

```
...
[4.2] Links menu for Node Presenters
...
```

Calling `FUNC add_help_menu_value_links node1 "Help_Menu"`, would produce a value link with role `"Help_Menu"` and value `"Node Presenters:Links Menu"` with an offset that is at the beginning of this section. Kiosk can then use this value link to jump to this section when a user wants help concerning the Links menu of a Kiosk node presenter.

7.5.1.3 Pp-functions found in $rgs/GMM_pps.C

The pp-functions of this section are very specific to the Group Memory Manager application and to the Kiosk feedback mechanism. There is not enough space here to describe these functions. See source code for details.

7.5.2 Writing new pp-programs

The invocation of pp-programs follows the same form as calls to program extractors, namely:

```
<pp-name> <node-path-name> [<pp-arg>]*
```

Cost++ runs the program or script in `<pp-name>`, but doesn’t read any information printed by the pp-program.

7.5.3 Writing new pp-functions

Pp-functions have the form:

```
void <pp-name> (Node *the-node, const char *arg, ...);
```

Follow these steps in writing a new pp-function:

1. Ensure you have everything necessary to compile Cost++ (see section 8.3 on page 71 for details).
2. Write the body of your function, adding to an existing library of post-processors, or creating a new library. Instructions for creating a new library are found at the top of all post-processor library sources (e.g., `$rgs/general_pps.C`). Common operations used in extractors include the various member functions available on the `Node` object passed into your extractor. See function extractor notes for details (section 7.4.3 on page 62).
3. Register the new pp-function in the initialization routine for the library in which it belongs. For `$rgs/general_pps.C`, this is the function `initialize_for_general_pp`. You will need to add a line of the form:

   ```
   pptable->register_PP_func ("<pp-name>" , <pp-name>)
   ```

   This will allow Cost++ to correctly identify a pp-descriptor with a pp-name that is the name of your new pp-function.
4. After writing the new pp-function and registering it, recompile Cost++ (run `nmake` in directory `$rgs`).
5. Add the post-processing section pp-descriptors to the config files where this pp-function will be used.

7.5.4 Post-processing BNF

```
<pp-section>    ::= (POST-PROCESS [<pp-descriptor>]*)
<pp-descriptor> ::= (<pp-where-look> <pp-type> <pp-name>
                                              [<pp-arg>]*)
<pp-where-look> ::= *ANY* | *STRUCTURE* | *CLUSTER* | *CLASS* |
                    <full-pathname>
<pp-name>       ::= <func-symbol> | <full-pathname>
<pp-type>       ::= PROGRAM | FUNC
<pp-arg         ::= <string>
```

8 Using Cost++

8.1 Installing Cost++

In order to install Cost++, you must have access to the *HP ninstall* facility and have obtained permission to install Cost++ by sending e-mail to kiosk@hplkiosk.hpl.hp.com. Once complete, you can install Cost++ by following the instructions listed after performing:

```
ninstall -dh hplkiosk.hpl.hp.com kiosk-dev
```

After installing, make sure you setup the correct environment variables according to the instructions.

8.2 Executing Cost++

The executable for Cost++ is located on `$rb`. To execute Cost++, enter:

```
$rb/Cost++ [-d][-m] [<config-filename>]*
```

`<config-filename>` is the pathname of each config file to be read by Cost++. If none are specified, the file `$rgd/import.cost` is used. As Cost++ is executing, it attempts to give you feedback about the different stages of its execution. This includes what section of the config file is being read, names of structure nodes created, internal links created, and when the lattices built are saved out. For even more information, run Cost++ with the debug option (`-d`). This is very useful when you are debugging feature extractors and post processing functions.

The `-m` option tells Cost++ to run in *multi-user* mode. Its purpose is to avoid multi-user contention problems (i.e., two or more users modifying the same node at the same time). When in this mode, Cost++ attempts to communicate with other Kiosk and Cost++ programs that are also running in multi-user mode. This communication is performed through the Bart software bus[1][2]. If Bart is not running, Cost++ will immediately terminate with a message to this effect. Otherwise, Cost++ will post all nodes that it modifies over Bart. This results in all other multi-user Kiosk and Cost++

programs being informed of nodes this Cost++ is modifying. Furthermore, just before Cost++ attempts to save out all the lattices it has built (stage 6), it will check if any of the nodes it has modified are also being modified by other Kiosk or Cost++ programs. If any such nodes are found, Cost++ will terminate with a message to this effect.

Examples:

```
$rb/Cost++
$rb/Cost++ $rge/simple1/simple1.cost
$rb/Cost++ -m /tmp/banana.cost
$rb/Cost++ -d /tmp/foo1.cost /tmp/foo2.cost
```

8.3 Compiling Cost++

Cost++ has been successfully compiled under HP-UX 7.03 and HP-UX 8.00 on the 300 series machines and under HP-UX 8.07 on the 700 series machines. There is no guarantee that Cost++ will compile under other operating systems or on other machines. The following guidelines should help you in recompiling Cost++:

1. Cost++ uses several different software libraries and packages which you will need to install before compiling. This includes:
   1. C++ version 3.0—you will need a C++ licence before you can install. Code is available, via update, from `hplego.cup.hp.com`.
   2. Codelibs—available via ninstall from `hp-gjd.sde.hp.com` as the package `codelibs`. Always install Codelibs after you install C++.
   3. lsdmem—use anonymous ftp to `hplsdln`. Change directory to `pub/liblsdmem`, and read the `README` (“`get README -`”) file. Pickup the package appropriate for your system. Then unpack it according to the (simple) instructions in the `README`.
   4. nmake—available via ninstall from `hp-gjd.sde.hp.com` as the package `nmake`.

   You need not worry about where to install C++, Codelibs, lsdmem, and nmake—they all have hard-wired places where they are installed.

2. Install the Cost++ sources through:

   ```
   ninstall -vh hplkiosk.hpl.hp.com kiosk-dev.Cost++.src
   ```

   Also make sure you have setup the necessary environment variables for compiling, according to `$ra/needed_env_vars`.

3. Modify the `$rgs/Makefileops` file to include or exclude the machine and libraries you wish to have in your version of Cost++.

4. Change directory to `$rgs`, and type `nmake`. The new version of Cost++ will be placed on `$rgs` and on `$rb`.

9 Limitations and Future Work

An important part of learning a system is understanding its limitations. In this section, Cost++ limitations will be considered through examining potential enhancements to Cost++. These enhancements are placed into three categories that are discussed below—feature extractor and post-processor improvements, language consistency and simplification, and design and implementation improvements.

9.1 Feature Extractor and Post-processor Improvements

This section considers some potential improvements to feature extractors and post-processors.

9.1.1 Linking the results of separate config files

Although we have successfully performed global linking between separately generated structures using the `external_link_parse_func`, there is no current way to link across the lattices generated by different config files. Such linking is needed for situations like "see also" manual page references from workproducts of one config file to a manual page generated within another config file. For example, the InterViews library might be generated by the config file `IV.config` and the Codelibs library generated using the config file `Codelibs.config`. Within each of these libraries, all manual pages defined and referenced are linked. However, the cross-library definitions and references will not be linked, as in a manual page reference from an InterViews component to a Codelibs component. In this case, Cost++ needs the manual page reference and definition information for each library for use when other libraries are generated.

9.1.2 Richer node distinguishing ability

Extractors can only distinguish nodes for examination based on their class or through being read by a specific action-descriptor. This is too limiting for certain applications. For example, we might have similarly named nodes stored in different locations, as in a `Mark_Gisi` classification node under a `Bugs_Filed_By` node and a `Mark_Gisi` node under a `Bugs_Fixed_By` node. It is currently difficult to have a different feature extractor that handles each of these different `Mark_Gisi` nodes.

Another related ability needed is to allow feature extractors to run over all nodes. This could be accomplished by extending feature extractor's where-look to include the `*ANY*` option, like post-processors.

9.1.3 Feature extraction based on information from other nodes

Feature extractors and post-processors cannot easily create links based on information that spans nodes. Their simplicity, in that they work on each node in

isolation, presents a problem when linking requires information about other nodes. For example, we cannot currently sort a set of e-mail messages according to their submission dates and link them in ascending order. No data structures or support currently exist for storing information about previous nodes considered by feature extractors and post-processors.

9.1.4 More flexibility for determining the link role generated

At present, the link roles specified in node-relators are a hardcoded name (in the configuration file), or the name of the keyword found by a feature extractor. There are cases where we would like to compute the link role in an independent manner. For example, we might want to link an e-mail message with topic `Meeting` to a `Meetings` classification node and have the link role be the contents of the `From:` field of the mail message (e.g., Michael L. Creech).

9.1.5 Value link generation

Feature extractors should be able to build value links. Currently, post-processors must be used to perform this linking.

9.1.6 Link owner specification

Currently, all links generated by Cost++ have an owner of `Cost++`. It is sometimes desirable to generate links with a different owner so that these links will not go away when a config file uses a `KEEP-LINKS` of `USER-GENERATED`. This could be done by adding an optional `OWNER` field to node-relators.

9.2 Language Consistency and Simplification

Enhancements discussed here concern changing overly complicated and inconsistent structures in the Cost++ language.

9.2.1 Node-relator where-look consistency

At present, a node-relator where-look can be the name of an action-descriptor or one of a set of special symbols (e.g., `*CLUSTER*`, `*CLASS*`, `*WORKPRODUCT*`). To minimize the number of these special symbols, the use of `*CLUSTER*` and `*CLASS*` should be replaced with `CLUSTER_MAIN` and `CLASS_MAIN`, respectively.

9.2.2 Make workproducts more like structure nodes

The use of workproducts could be simplified and generalized by making them work more like structure nodes. In this case, three major extensions to workproducts are needed:

- Allow workproducts to have child nodes.

  It would sometimes be useful to have workproducts that have other workproducts or structure nodes as children.
- Allow workproducts to be read in at the top level.

At present, workproducts cannot be read in at the top level—they must be read in underneath a structure node. For example, if we want to read a set of workproducts that will be linked together by feature extractors, but we do not need any structure nodes, we have to read the workproducts under a "bogus" structure node and then remove this node using a pp-function.

- Add action-options to workproducts.

  There are times when we need to change the value of equivalent action-options for workproducts. For example, read a specific workproduct from a different location then the standard location, as in: (`NROFF-DOC ("foo.3I") IN:$rd`).

9.2.3 Simplification of field-placeholder removal

Unused field-placeholders are currently removed through the `*GLOBAL*` action-options; `REMOVE-CLUSTER-EMPTIES` and `REMOVE-CLASS-EMPTIES`. To reduce the number of special action-options, these can be easily performed by post-processing operations.

9.2.4 Rename the `ROLE` action-option

As we have discussed in section 4.1 on page 24, there is an ambiguity in the use of the `ROLE` action-option for workproducts versus structure nodes. For clarification, we might change `ROLE` into `ROLE-TO-DECENDENT` for structure nodes, and `ROLE-TO-PARENT` for workproducts.

9.2.5 Newline characters in strings

Various feature extractors and post-processors take strings that represent regular expressions for searching purposes. The newline character cannot currently be passed into a Cost++ string, causing "kludge" extra parameters to be added to pass newline information to these functions.

9.3 Design and Implementation Improvements

This section considers features that require improvements to the way Cost++ is designed and implemented.

9.3.1 Extensible parse functions

The current parse functions are difficult to extend. They should be restructured so that they follow the same form as feature extractors—libraries of separately compilable parse functions can be created along with script-based parse functions.

9.3.2 Better syntax error handling and recovery

Error reporting and recovery for syntax errors is very weak within the current Cost++. The existing recursive decent parser should be replaced with a Unix *YACC*-based parser with better error handling.

9.3.3 Greater user interaction

Cost++ currently works as a batch processing tool. There is a strong desire to make some of its features interactively available from Kiosk. For example, users might wish to command feature extractors to link over lists of specified nodes in an interactive manner. This requires restructuring Cost++ so that feature extraction is programmatically callable and physically separable from the rest of Cost++.

10 References

1.Brian W. Beach. *Software Glue: Connecting Reusable Software Components*. PhD Thesis, University of California, Santa Cruz. 1992.

2.Brian W. Beach. Connecting Software Components with Declarative Glue. In *Proceedings of 14th International Conference on Software Engineering*, pages 120-137, Melbourne, Australia, May 1992.

3.Michael L. Creech, Dennis F. Freeze and Martin L. Griss. Kiosk: A Hypertext-based Software Reuse Tool. Internal *HP Laboratories Technical Report* SSL-TM-91-03, March 1991.

4.Michael L. Creech, Dennis F. Freeze and Martin L. Griss. Using Hypertext in Selecting Reusable Software Components. *Hypertext '91*, pages 25-38, December 1991.

5.Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing users interfaces with InterViews. *IEEE Computer*, pages 8-22, February 1989.

6.Parag Patel. Codelibs - About a C++ Code Re-use Library. In *Proceedings of the 3rd International TOOLS Conference*, page 79, 1990.

7.Christine L. Tsien. Automated Link Creation in a Hypertext-based Software Reuse Library. *HP Laboratories Technical Report* HPL-91-131, September 1991.

11 Index

R

S