MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1256 October 1990

Supporting Reuse and Evolution in Software Design

by

Yang Meng Tan

Abstract

Program design is an area of programming that can benefit significantly from machine-mediated assistance. A proposed tool, called the *Design Apprentice* (DA), can assist a programmer in the detailed design of programs. The DA supports software reuse through a library of commonly-used algorithmic fragments, or *clichés*, that codifies standard programming. The cliché library enables the programmer to describe the design of a program concisely. The DA can detect some kinds of inconsistencies and incompleteness in program descriptions. It automates detailed design by automatically selecting appropriate algorithms and data structures. It supports the evolution of program designs by keeping explicit dependencies between the design decisions made. These capabilities of the DA are underlaid by a model of programming, called *programming by successive elaboration*, which mimics the way programmers interact. Programming by successive elaboration is characterized by the use of breadth-first exposition of layered program descriptions and the successive modifications of descriptions.

A scenario is presented to illustrate the concept of the DA. Techniques for automating the detailed design process are described. A framework is given in which designs are incrementally augmented and modified by a succession of design steps. A library of clichés and a suite of design steps needed to support the scenario are presented.

Copyright © Massachusetts Institute of Technology, 1990

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the following organizations: National Science Foundation under grant IRI-8616644, Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124, IBM, MCC, NYNEX, and Siemens.

The views and conclusions contained in this document are those of the author, and should not be interpreted as representing the policies, neither expressed nor implied, of these organizations.

Contents

1 The Design Apprentice **1**
- 1.1 Overview . . . 1
- 1.2 The Design Process . . . 4
- 1.3 Automating Detailed Design . . . 5
- 1.4 Current Status and Route Map . . . 6

2 Scenario **7**
- 2.1 What the DA Knows . . . 7
- 2.2 Paragraph Justification . . . 9
- 2.3 Scenario . . . 10
- 2.4 Scene 1: Initial Program Description . . . 11
- 2.5 Scene 2: Elaboration and Interaction . . . 13
  - 2.5.1 Elaboration . . . 13
  - 2.5.2 Interaction . . . 15
  - 2.5.3 Output Code . . . 19
- 2.6 Scene 3: Explaining Design Rationale . . . 29
- 2.7 Scene 4: Adding a Guideline . . . 31
- 2.8 Scene 5: A Correction . . . 34
- 2.9 Scene 6: The Complete Description . . . 37

3 The Design Process **39**
- 3.1 What the Programmer Does . . . 39
- 3.2 What the DA Does . . . 40
- 3.3 A Framework for Automating Detailed Design . . . 43
  - 3.3.1 An Example of Automatic Detailed Design . . . 50
  - 3.3.2 Further Challenges . . . 52
- 3.4 Recording Design Dependencies . . . 53

4 Representing and Manipulating Design Artifacts **55**
- 4.1 The Plan Calculus . . . 55
- 4.2 CAKE . . . 59
- 4.3 SERIES . . . 61
- 4.4 The Cliché Library . . . 62

4.4.1 Abstract Data Types 63
4.4.2 Taxonomy of Related Clichés 63
4.4.3 Going Beyond the Plan Calculus 68
4.4.4 Families in the Cliché Library 68
4.5 Representing Design Artifacts 69
4.5.1 Translating Input Descriptions into Plans 69
4.5.2 Finer Points about Representing Designs 70
4.5.3 Representing Design Dependencies 70
4.6 Manipulating Design Artifacts 71
4.6.1 Design Steps 71
4.6.2 Propagating Constraints 83
4.6.3 When is a Design Complete? 86

5 Related Work **88**
5.1 KBEmacs 88
5.2 Deductive Synthesis 89
5.3 Program Transformation 89
5.4 Very High Level Languages 90
5.5 Program Generators 91
5.6 Algorithm Design 92
5.7 Selection of Data Structures and Algorithms 92

6 Future Work and Conclusions **94**
6.1 Conclusions 96

Acknowledgments

I am very grateful to my thesis advisor and mentor, Dick Waters, for guiding me in this work patiently. His steady moral support, advice and insights help me through thick and thin. Chuck Rich has been a constant source of ideas for me; I want to thank him for his time and patience. Many of the good ideas reported in this work originate from Dick and Chuck; the bad ones are all mine. My office-mates, Howard Reubenstein and Linda Wills are invaluable moral and technical supports for me, so are visiting Professor Rudolph Seviora and other members of the Programmer's Apprentice group: Yishai Feldman, Paul Lefelhocz, and Bob Hall.

My apartment-mates, Shail Gupta and Vishak Sankaran, generously share their Indian condiments which help spice up my life, and in return, I share my ups and downs with them. I like to thank the following for their camaraderie: Kah Kay Sung, Choon Phong Goh, Siang-Chun The, Sandiway Fong, and Beng Hong Lim.

My family is always there for me with patience and support, despite the great distance apart. My elder brothers, especially Yang Whye, have worked hard and provided for the family during many difficult years, keeping us alive and well. I am mindful of their contributions to everything I do.

Tze-Yun Leong is always there for me. I am deeply indebted to her for the caring support, enduring patience, confidence and love she has given me all along. I want to thank her for her assistance in preparing the illustrations. Thanks, Tze-Yun, for making *everything* in life looks better.

This publication is a revised version of *Supporting Reuse and Evolution in Software Design*, a report submitted to the Department of Electrical Engineering and Computer Science on August 31, 1990 in partial fulfillment of the requirements for the degree of Master of Science.

Chapter 1

The Design Apprentice

Software is difficult to construct and maintain reliably and cost-effectively. The increasing cost of reliable software needs to be countered by increases in programmer productivity. In this thesis, we explore one approach toward increasing the productivity of programmers. We put forward a new programming paradigm that is knowledge-based and assistant-based.

This paradigm is explored in the context of a proposed design support system, called the Design Apprentice (DA), which can assist the programmer in detailed design. The DA supports software reuse through a library of commonly-used algorithmic fragments, or *clichés*. The input language of the DA, together with the cliché library, enables the programmer to describe the design of a program concisely. The DA automates program design by automatically selecting appropriate algorithms and data structures. It can also detect some kinds of inconsistency and incompleteness in program descriptions. It supports the evolution of program designs by keeping explicit dependencies between the design decisions made.

In addition to the practical value of the DA, the codification of programming knowledge required in building the DA helps explore the adequacies of formalisms for representing and reasoning about programming knowledge. It can also serve as a step toward understanding how programmers approach detailed design based on their past experience.

1.1 Overview

This work is part of the Programmer's Apprentice (PA) project [27] whose goal is to design an intelligent software tool to support expert programmers in all aspects of software development. As a research strategy, most of the focus has been put on the two ends of the software development process: a *Requirements Apprentice* [24] acquires software requirements from informal descriptions, and the Design Apprentice assists the programmer in the detailed design and synthesis of programs.

Figure 1-1 summarizes our rationale for the design of the DA and the technologies

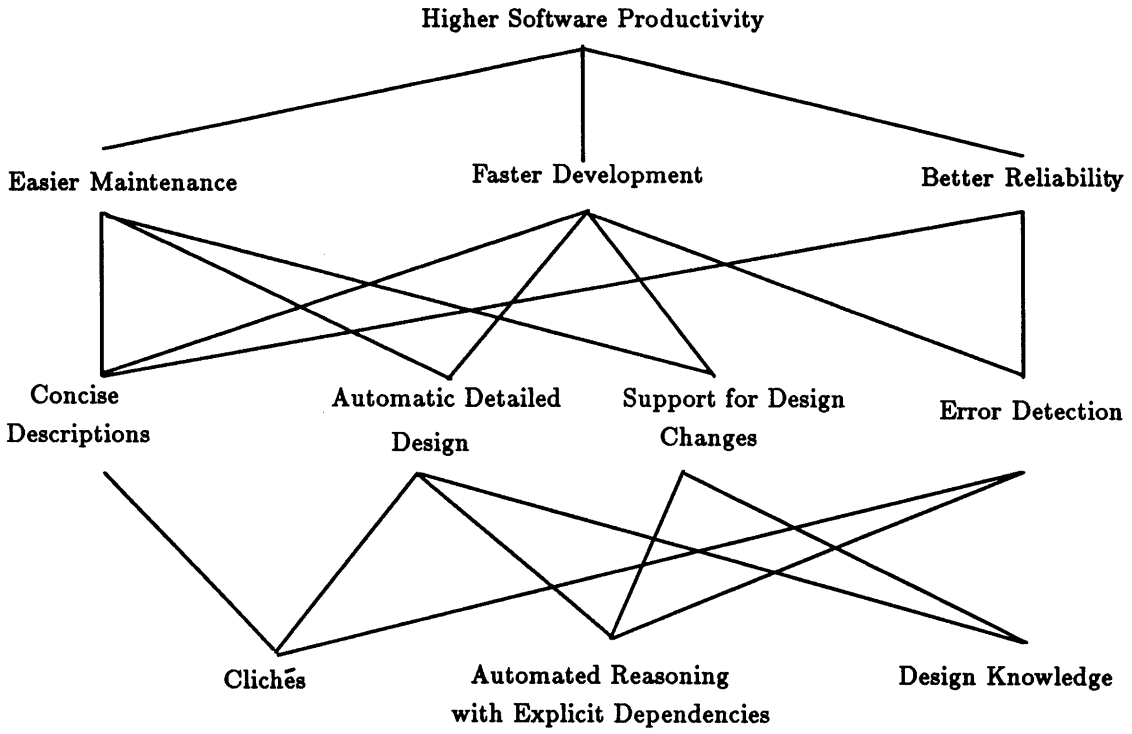


Figure 1-1: Motivation and Features of the DA.

that can be used to build the DA. The second row of the figure shows the subgoals which contribute to higher software productivity: easier software maintenance, faster software development, and more reliable software. The key capabilities of the DA that support these subgoals (shown in the third row of Figure 1-1) are discussed below.

Concise and Comprehensible Program Descriptions: The DA will allow the programmer to describe a program in a concise and comprehensible manner. The key advantage of using short and familiar descriptions to specify programs is that the descriptions are easier to write and to understand. The succinctness of the description eases the task of validation, and its familiarity helps to make specifications self-evidently correct. This support can contribute to faster software development and easier software maintenance.

Automatic Detailed Design: Much of detailed design of programs is routine, yet error-prone. With the DA, the programmer need not specify a program to the fullest details. The DA will be able to select appropriate data structures and algorithms to implement a given program description. The automation of detailed design will relieve the programmer from such mundane chores so that more attention can be paid to the high-level design of programs. This capability can contribute to more efficient software development and maintenance.

Support for Design Changes: Specifications in the real world evolve over time. To support program evolution, the DA will allow programmers to retract design decisions and make changes incrementally, and in any order. The DA provides support for changes by keeping explicit dependencies between decisions. This helps program maintenance, because past design rationale are accessible, and the impact of a design change can be studied.

Error Detection: Program descriptions are frequently incomplete and often inconsistent. The DA will be able to deal with some of this incompleteness in program descriptions by detecting them and informing the user. It will also be able to detect some kinds of inconsistencies and occasionally, offer fixes. This capability can help to improve reliability and quicken software development.

These capabilities of the DA can be supported by the technologies shown in the last row of Figure 1-1.

Clichés: A key component of the DA is a library of clichés, which codifies its knowledge of algorithms and design. A cliché contains three kinds of parts: parts that are fixed, parts that can vary from one use to another, and constraints on the parts. The variable parts are called *roles*. Many of the commonly-used programming concepts that comprise the technical vocabulary of programming can be viewed as clichés. Clichés connote frequent reuse. In software engineering frequent reuses of specifications and code are desirable, because they are more likely to be bug-free and because we are more familiar with them than with newly-written ones. The programmer will describe an intended program to the DA in terms of clichés in the library. Through the use of clichés, the programmer can describe a program in a compact and understandable way. To support detailed design, the library can contain clichés codifying different algorithms and data structures and how they can be used to implement more abstract specifications. Errors can be detected when constraints of different clichés interact to produce inconsistencies.

Automated Reasoning with Explicit Dependencies: A deductive engine provided by an automated reasoning system serves as a medium for the DA to propagate design constraints. These constraints help limit design choices in the automatic design process and allow for error checking. Explicit dependencies between the constraints, if maintained, can be used to support design changes in the DA. Dependencies between design decisions made in the DA can be encoded as constraints so that design decisions can be retracted or re-installed when their supporting decisions are removed or added.

Design Knowledge: Design knowledge guides the selection of algorithms and data structures. It also indicates how design decisions depend on each other. This is needed to determine dependencies between design decisions to support design changes. Design knowledge also includes design strategies for making progress in detailed design. Given that design is an under-constrained problem, heuristics are needed to automate detailed design.

1.2 The Design Process

Underlying the capabilities of the DA is a process model that mimics the way programmers interact. In particular, an expert programmer typically describes, using some intermediate level vocabulary, to a junior programmer what is needed. Such descriptions typically specify some input-output behavior and the high-level design of a program intended to meet this specification. The junior partner may ask the senior programmer questions to clarify doubts. They collaborate in the programming task in a co-operative manner.

Communication between programmers seldom takes place at the level of formal logical specifications. Rather, effective communication relies on a large body of shared experience or knowledge. Much of the shared knowledge between programmers can be codified as clichés and used by the DA acting as an assistant to a programmer. Clichés serve as convenient contexts for bringing related concepts into the description of programs. They serve as compact description tools.

With the DA, a programmer describes an intended program using program descriptions that refer to clichés and indicate the relationships between them. The high-level design of the program can be provided through explicit control clichés. The programmer can also indicate abstract properties of data and functions, and provide implementation guidelines. In addition, specific design steps to be carried out can also be given.

This style of program description is characterized by the use of breadth-first exposition of layered program descriptions and the successive modification of descriptions. Starting out with a few main clichés, a programmer works outward, describing each use of the clichés in more detail. This process is similar to the interaction between human programmers when a complicated computer program is being described. Frequently, to facilitate understanding, programs are described at different levels of details, in a layered fashion. We termed this process *programming by successive elaboration.*

In this process, the descriptions given by the programmer need not be strict refinements of program descriptions already said. Rather, earlier program descriptions may be retracted, modified, superseded, redefined, and re-stated. Viewed as structured natural language discourse, this process of program description elaboration encompasses the program refinement approach and goes beyond it.

The DA supports a fundamental shift of focus in the programming process. Different programming tools view programs in different lights. The DA interacts with the programmer in terms of design decisions. Adding new design decisions, changing old ones, and removing contradictory decisions are the currencies of this process. In contrast, a text editor helps a programmer edit programs as text, a syntax-directed program editor edits programs as syntax trees, and an algorithmic structure editor [33, 34] supports editing in terms of the algorithmic structure of programs. The DA maintains design dependencies among design decisions made, and helps focus the

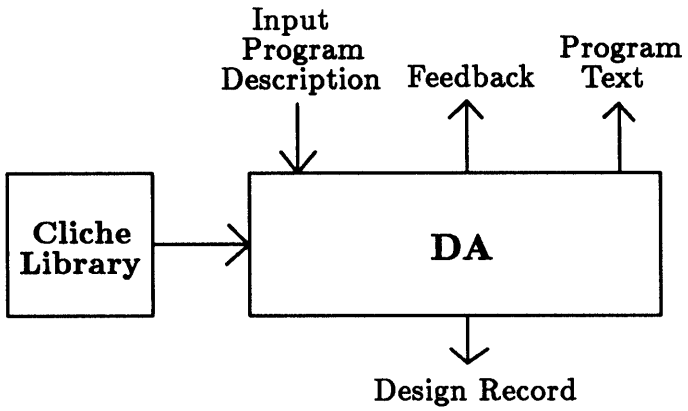


Figure 1-2: The Design Apprentice.

programming process on the decision structure of programs.

1.3 Automating Detailed Design

The DA uses the Plan Calculus formalism [25] to represent program designs as *plans*. Steps one takes in designing programs are codified as *design steps*. These design steps manipulate and transform program designs represented as plans. Given the design of a program, there may be many different well-motivated design steps the DA can take. Design heuristics can be used to choose among the alternatives. Design steps also record the decisions that support their applicability and make explicit the dependencies among the subsidiary design decisions they engender.

Figure 1-2 shows the inputs and outputs of the DA. The DA has a cliché library which codifies the shared knowledge between the programmer and the DA. The cliché library contains knowledge about commonly-used algorithms and data operations on modeling types such as set and sequence. These allow the programmer to specify the program in a familiar and concise form, and support the selection of algorithms and data structures.

The input program description given to the DA is translated into plans and serves as the initial design. The DA goes through a design cycle during which design steps are successively taken to transform the initial design into a final design. The output code is extracted from the final design that results from this design process. A design record documents the design process. Besides program text, the DA also outputs interactive warning and error messages whenever it encounters errors in the program description.

A key component of the DA is a hybrid knowledge representation and reasoning system called CAKE [9, 26] which implements the Plan Calculus and a number

of reasoning facilities including a truth maintenance system (TMS). CAKE provides the infrastructure on which the needed programming knowledge is represented and reasoned about. CAKE is used to propagate constraints and to maintain design dependencies. Constraints help to limit the design, detect contradictions, and propagate design information along data flow arcs. CAKE also supports automatic retraction of assertions and their consequents, and contradiction detections.

Design is an under-constrained problem. The DA uses a strategy that forges ahead by making assumptions and tries to complete the design as far as possible. Whenever a dead end in the design process is reached, the DA retracts some of the assumptions that led to the dead end. This strategy is especially effective in the presence of incomplete knowledge and incomplete specifications because many analysis techniques require either complete knowledge or complete specifications. Explicit dependencies between design decisions are essential to ease the book-keeping required by this strategy. These dependencies are installed by the design steps which transform designs. Examples of design steps include steps that select algorithms, select data structures to implement data types, select implementations for mappings, coerce types, precompute mappings, and cache mappings.

1.4 Current Status and Route Map

This thesis reports on progress made in building the DA. To illustrate the features of the DA, a detailed scenario has been created. This scenario is shown in Chapter 2. The conceptual design of the DA has been advanced based on work towards implementing the scenario. A library of clichés, mostly codified in the Plan Calculus, has been created to support the scenario. The design steps required to bridge the gap between the input description and the target code have been worked out.

In Chapter 3, we describe the design process the DA supports from the perspective of the programmer who uses it. In the same chapter, we also describe our initial idea of how the DA can automate the process of detailed design. There, we also indicate some challenges that need to be addressed. Chapter 4 shows how the knowledge needed to achieve the target scenario along with the different design artifacts can be represented and used. Work related to this thesis is discussed in Chapter 5. Chapter 6 suggests directions for the future, and summarizes the key results of our work.

Chapter 2

Scenario

The following scenario shows how a programmer can use the DA to develop a program for paragraph justification. The objective of a paragraph justification program is to break up paragraphs into lines as evenly as possible: to avoid narrow, cramped lines as well as to avoid widely-spaced, loosely-set lines. In this scenario, it is assumed that the DA does not know about paragraph justification. It knows about some general-purpose programming clichés and some graph algorithms. The example program is chosen because we think it is a realistic and non-trivial program. There is sufficient complexity in the specification of the problem to challenge both the program description language as well as the synthesis of the program. Also, the clichés that are used to motivate this example are commonly used concepts, familiar to many people with a programming background. In addition, the chosen scenario uses some clichés which are represented as program generators. This hints at how program generators can be incorporated into the DA.

Below, we first provide a brief description of the clichés and the kinds of knowledge we expect the DA to know about. We then describe the paragraph justification problem and its solution. Next, we sketch the scenario in which a description of the solution to the paragraph justification problem is given to the DA.

2.1 What the DA Knows

The DA is expected to know various pieces of knowledge about programming. Some of these general clichés are described informally and briefly below. They are described in more details where they are used.

Tokenize: *Tokenize* is a cliché that converts a sequence of characters into a sequence of tokens. It is represented as a program generator. The grammar to be parsed is specified using regular expressions. It is similar to the popular UNIX facility LEX. Some auxiliary concepts and functions defined by the tokenize cliché includes: the *content* of a token, the *type* of a token, and the set of all tokens of a specified pattern type.

Build Graph: The *build-graph* cliché captures the concept of building a graph from some given inputs. It embodies standard ways of constructing graphs. It has a number of roles. The most important ones include the following: the *input-to-node-map* role expects a function that maps the input items to the nodes of the output graph, the *arc-test* role is to be filled in by a predicate which decides whether two given input items should be connected in the output graph, and the *node-test* role is analogous to the Arc-Test role but works for nodes instead. The build-graph cliché has many other different roles because there are many ways of building a graph. Different graphs motivate different graph construction procedures which are captured in different build-graph clichés. Many roles in this cliché are optional; the user need not specify all of them in order to use it.

Single-Source Shortest Path: This cliché takes in a graph and finds a single-source shortest path. The output of this algorithm may be the length of the shortest path or the shortest path itself, or both. There are a few different shortest path algorithms, including those by Dijkstra, Bellman-Ford, and one for directed acyclic graphs [7]. For some of these algorithms, there are different data structures which can be used and they may yield different running times. The output of the algorithm may also be represented in different ways. This cliché captures some of the knowledge involved in selecting the appropriate algorithm based on the characteristics of the input graph.

Prorate: The *prorate* cliché codifies the concept of proration, and embodies the different ways of prorating an amount among a set. In the simplest case, it prorates an amount evenly among all members of the given set.

There are some subtleties involved in the seemingly simple proration cliché, e.g., when the total shares of the members does not divide the amount to be prorated, where should the "remainder" go? It could be distributed at random, or in the case of a given ordered set, it could be distributed so as to favor earlier elements or later elements, or to favor elements from both ends. This cliché abstracts from the detailed proration computation and allows the user to specify needs in terms of the abovementioned intermediate-level concepts. Prorate obeys the *Non-Shareholders get nothing* principle: it takes care to prorate only among the members of the given set which have non-zero shares in the proration process. In addition, some proration constraints are explicitly kept: there can be no negative shares. Here, we restrict the variations of Prorate to those that run in linear time. More sophisticated schemes of proration that involve sorting may take longer.

Segment: The *segment* cliché takes a sequence of items and breaks it up into a sequence of sequences of items. This cliché handles the beginning item and the end item of the sequence as boundary items when needed. This is a useful cliché in segmenting sequences, because it encapsulates some details about how to handle the end points appropriately.

Ascii-File: If a file is an ASCII file, it contains only characters belonging to the ASCII character set. The cliché encodes knowledge about the ASCII characters; for

example, that there are 128 characters in the ASCII character set, and that the space character and the return character belong to the blank-printable character subset. Familiar concepts like the non-blank printable character set, the printable character set, and the control character set are codified in this cliché.

Concatenate: The *concatenate* cliché is a program generator that generates an abstract program for concatenating arbitrary numbers of input arguments. Some input arguments may be known to be non-sequences, in which case, they are coerced into sequences statically by the program generator. The program generated is abstract in the sense that it is made up of calls to the abstract functions such as *insert* which adds an element to the front of an abstract sequence, *endcons* which adds an element at the back of the sequence, and *abstract-append* which appends two abstract sequences.

2.2 Paragraph Justification

The paragraph justification example is inspired by the way the TEX program breaks a paragraph into lines [15]. As a document typesetting program, TEX performs substantially more functions than paragraph justification. This example adopts TEX's model of paragraph typesetting but omits much of its complexity.

A paragraph is viewed as a sequence of word tokens and gap tokens. Word tokens have fixed widths, whereas the size of gap tokens may be stretched to fill up any slack. Each gap token has a width property and a stretchability property. The latter property indicates the relative amount a gap can be stretched. Each gap token in the sequence is a potential point to break a line. The word tokens and gap tokens between two adjacent break points form a line.

A graph is constructed from the sequence of tokens that represents the paragraph with the following intentions in mind: nodes of the graph are to be made from gap tokens from the input sequence, they represent places where the paragraph may be broken. A starting gap token is added to start the paragraph out, and a trailing gap token is added to end the paragraph. These two gap tokens are break points in the final typeset paragraph. An arc represents a line since the end point nodes are break points in the paragraph. The arcs of the graph are constructed in such a way that paths from the start token to the end token represent different ways of breaking the paragraph into lines.

One way to construct such a graph is to make every gap token a node and build a complete graph out of the resultant node set. Some nodes and arcs can be excluded in the construction of the graph as an optimization. We can define a metric on any two input gap tokens in the token sequence. For efficiency, we make an arc between two nodes corresponding to two gap tokens only if the metric on the two gap tokens is within some thresholds. Now, suppose we have chosen the metric such that it is high when the lines are loosely set, and that it is low when the lines are close to the normal

(optimal) settings. Since every arc has a corresponding pair of input gap tokens, we can view this metric as a measure of quality imposed on the arcs which indicates how well the lines will look in the typeset paragraph. Thus, we can view the problem of typesetting the paragraph as an instance of the shortest path problem. The path from the start token to the end token that has the smallest metric corresponds to the best way to break the paragraph into lines.

Specifically, we can define the metric to be the number of extra spaces to be distributed among the available gaps. The ideal situation is when the metric is 0 for all the lines. Since we cannot shrink the spaces in our model, this becomes the lower bound of the threshold on the metric. The upper bound represents the maximum number of extra blank spaces we will allow in each gap on a typeset line. To compute this metric, we need to compute the total number of gap tokens and the extra spaces remaining in a line.

Shortest path problems have standard efficient algorithms. In particular, the graph which arises from the paragraph justification model turns out to be rooted, directed, and acyclic.

After the shortest path has been computed, each arc in the path represents a line. For each line, we can prorate the extra spaces among the gap tokens and output the words and gaps in order.

In the scenario, we assume fixed width fonts. Hence gap tokens cannot be shrunk and they can only be stretched by fixed quanta.

2.3 Scenario

To aid in viewing the scenario, the following typographical conventions are used: keywords of the program description language are in uppercase, cliché names are in initial capitals, and changes in the input description and the output code between the steps of the scenario are underlined where appropriate.

The exact form of the description language is not important to the tasks at hand. Some means for the programmer to specify the clichés to use, some means to fill in the roles of the clichés, and some means to specify logical constraints should suffice. The major forms in the input language used in the scenario are described below:

- (`LET` ({(*Variable Value*)}*) {*Form*}*): This creates a local scope in which local variables can be named and manipulated. `LET*` is similarly defined except that the bindings of variables are performed sequentially rather than in parallel.

- (`FOR-EACH` (({*Variable*} {*Sequence*})*) {*Forms*}*): This is an explicit iteration construct. The {*Forms*} are ordered.

- (`DESIGN` {*Name*} ({*Variable*}*) {*Forms*}*): This form defines the *Name* function, which takes the given argument list, to be the forms. {*Forms*} is a sequence of statements and/or assertions.

- (REDESIGN {*Name*} ({*Variable*}*) {*Forms*}*): This is identical to DESIGN except that it indicates to the DA that *Name* has been defined before, and that the current definition supersedes the previous one.

- (WITHIN {*Name*} ({*Variable*}*) {*Forms*}*): This form is used to provide additional information about a cliché named *Name* used elsewhere in the program description. The optional argument list {*Variable*}* allows the arguments passed to the use of the *Name* cliché to be used locally within the current description. {*Forms*} is a sequence of assertions. (For the purpose of this exploration, we ignore the need to refer to different uses of the same cliché in the same program. Some conventions need to be adopted to allow the user to distinguish different uses.)

- (MODIFY-WITHIN {*Name*} ({*Variable*}*) {*Forms*}*): This form is identical to WITHIN except that it indicates to the DA that the instance of *Name* used earlier is being modified. It supersedes any previous descriptions if they prove to be contradictory.

- (DECLARE {*Assert-Specs*}*): This form can occur anywhere in the description. *Assert-Specs* are logical predicates which are asserted to be true by this form.

- (*Operator* &rest *Arguments*): This is an application of the operator to the given arguments. *Operator* may be clichés known to the DA or other new user-supplied functions.

- (IS *Role Form*): This asserts that the *Role* is filled in with *Form*. This form occurs inside WITHIN forms, the *Role* is interpreted with respect to the cliché described by the WITHIN form.

- (== *Form-1 Form-2*): This asserts that *Form-1* is logically equivalent to *Form-2*.

- (INSTANCE {*type*} {*Keyword Form*}*): This instantiates an object of the given type. There may be mappings whose domain is *type*. If F is a mappings whose domain is *type*, then :F is a valid keyword which can be used to associate the result of F on the instance with the instance itself.

2.4 Scene 1: Initial Program Description

The scenario starts with a programmer describing to the DA the high-level sketch of the paragraph justification program. This is done using specific clichés and explicit control specifications as shown in Figure 2-1.

The (incomplete) program description in Figure 2-1 mirrors some of the English description an expert programmer might provide to a junior programmer. The use of

```
(DESIGN justify (infile outfile)
  (DECLARE (Ascii-File infile) (Ascii-File outfile))
  (FOR-EACH ((paragraph (Segment (Tokenize infile) 'para-break)))
    (FOR-EACH ((line (Single-Source-Shortest-Path (Build-Graph paragraph))))
      (lineout outfile line))
    (Output outfile #\newline)))
(DESIGN lineout (outfile line)
  (Prorate (extras line) line)
  (FOR-EACH ((token line))
    (Output outfile (Content token)))
  (Output outfile #\newline))
(IMPLEMENTATION-GUIDELINES
  (PREFER time-efficiency)
  (IGNORE error-checking))
```

Figure 2-1: Initial Program Description of Justify.

commonly-known programming concepts make the program description concise and comprehensible, making it is easy to read and understand.

In the use of Tokenize in the description, **infile** is viewed as a sequence of characters. The mention of the Tokenize cliché indicates to the DA that the output from **(Tokenize infile)** is a sequence of tokens. The above design specifies that this sequence is to be segmented at para-breaks (tokens). (At this point, **para-break** is not yet defined.) The Build-Graph cliché, as used above, is given a sequence of tokens as input from which an output graph is computed. The design says that a single-source shortest path algorithm should be run on the graph. The output of the single-source shortest path algorithm may be the distance of the shortest path or the shortest path itself, or both. Since the output is used in an iteration construct in the given design, it clearly cannot be the cost of the optimal path. Hence the DA assumes that the result to be returned is an optimal path in the graph.

The **lineout** design says that the **extras** of the given line are first to be prorated among the tokens on the given line. (At this point, the function **extras** is not yet defined.) After the proration, the tokens on the line are output to **outfile** in order, and each line is ended by a newline character.

The programmer also gives the DA some implementation guidelines: the program should be efficient in its running time and error-checking should be ignored.

The DA is unable to generate any code based on the above description because it is too incomplete. When the programmer requests the DA to produce code, the DA informs the programmer accordingly as shown in Figure 2-2.

The dialog in Figure 2-2 illustrates the error detection capability of the DA. The DA issues a warning when it detects that the output of Prorate is not used in the description. The DA is following a general heuristic: an unused output is indicative of an omission in the program description. It also indicates that several functions are

```
>>> (Write-Code 'justify)
DA> WARNING: The output of Prorate is not used.
  WARNING: The following are undefined: para-break, extras.
  Tokenize: Incomplete specifications, please define the token types.
  Build-Graph: Incomplete specifications.
  Single-Source-Shortest-Path: Incomplete specifications.  Please
  provide the Start-Node, the End-Node and the Distance-Function.
```

Figure 2-2: Dialog between the DA and the Programmer.

undefined and that the specifications for a number of the clichés used are incomplete. In the case of the single-source shortest path cliché, it also indicates the mandatory unfilled roles. Here, the DA is helping to catch any incompleteness in the specification. In doing so, it is also guiding the programmer in the design, indicating places where further design is to be done.

2.5 Scene 2: Elaboration and Interaction

In this scene the programmer elaborates on the main clichés used in the design. The additional descriptions are shown in Figure 2-3. Using clichés, the programmer sketches succinctly the main operations needed for the paragraph justification task in the initial description. Together with the last scene, this scene illustrates the style of program design the DA is encouraging: present a program in a layered fashion, stating the main tasks using familiar clichés, and then developing each of the uses incrementally. In this scene, the programmer also moves to correct some of the mistakes and omissions made in the previous program description.

2.5.1 Elaboration

In `lineout`, the programmer distinguishes between two kinds of tokens: gap tokens and non-gap tokens. For gap tokens, the number of space characters to be output is equal to the width of the token. In Tokenize, the programmer defines the grammar to be scanned with the help of the `==` operator. `BOF` and `EOF` are markers for the beginning and the end of file. Para-breaks are blanks with at least two newlines, and the beginning and the end of a file are also para-breaks. (Blanks are either spaces or newlines.) Words are any non-empty sequence of non-blank printable characters. Any non-empty sequence of blanks is a gap. Once these patterns are defined, a number of auxiliary functions are automatically constructed from the tokenize cliché: the Gap-Token type is now understood to be the set of gap tokens, and Para-Break-Token and Word-Token are similarly defined. A few filter functions are also defined: Gap-Set

```
(REDESIGN lineout (outfile line)
  (LET* ((token-seq (token-sequence line))
         (prorated (Prorate (extras line) token-seq)))
  (FOR-EACH ((amount prorated) (token token-seq))
    (IF (gap-token-p token)
        (Output outfile #\space :number
                (+ amount (width token)))
        (Output outfile (Content token))))
  (Output outfile #\newline))
(WITHIN Tokenize ()
  (== para-break
      (regular-expression
        "BOF (#\space | #\newline)* | (#\space | #\newline)* EOF |
        (#\space)* #\newline (#\space)* #\newline (#\space | #\newline)*"))
  (== word (regular-expression "(Non-Blank-Printable-Characters)+"))
  (== gap (regular-expression "(#\space | #\newline)+")))
(WITHIN Build-Graph (paragraph)
  (LET ((new-paragraph (Concatenate head paragraph tail)))
    (IS (Domain Input-To-Node-Map) (gap-set new-paragraph))
    (DECLARE (Directed (Output Build-Graph))
             (Forward-Wrt-Input (Output Build-Graph)))
    (IS Root-Item head)
    (IS Arc-Test
        (Lambda ((xn token) (yn token))
          (LET ((tsequence (Sequence-In-Between xn yn new-paragraph)))
            (AND (Node (Input-To-Node-Map xn))
                 (>= (ratio tsequence) 0)
                 (<= (ratio tsequence) *Threshold*)))))))
(WITHIN Prorate ()
  (IS Proportional-To stretch) (IS Favor End))
(DESIGN ratio (seq) (/ (extras seq) (total-stretch seq)))
(DESIGN extras (seq) (- *Line-Length* (total-length seq)))
(DESIGN total-stretch (seq) (Sum (Map stretch seq)))
(DESIGN total-length (seq) (Sum (Map width seq)))
(DESIGN stretch (token) (IF (Gap-Token-P token) 1 0))
(DESIGN width (token)
  (IF (Word-Token-P token) (Number-Of-Characters token)
      (IF (Member (Last-Character (Preceding-Item token))
                  '(#\! #\? #\.)) 2 1)))
(== (Domain stretch) (Output Tokenize))
(== (Domain width) (Output Tokenize))
(DESIGN token-sequence (arc)
  (Sequence-In-Between (Node-To-Input-Map (Source-Node arc))
                       (Node-To-Input-Map (Destination-Node arc))))
(== head (INSTANCE 'gap-token :width *Para-Indent* :stretch 0))
(== tail (INSTANCE 'gap-token :width 0 :stretch 10000))
```

Figure 2-3: Additional Program Descriptions of Justify.

takes in a sequence of tokens and returns only the gap tokens, similarly for Word-Set and Para-Break-Set.

The programmer specified, in **build-graph**, that a new paragraph is to be formed from the old one by prefixing it with a leading gap, **head**, and suffixing it with a trailing gap, **tail**. **head** serves as the root of the graph to be constructed from the new paragraph. By making the width of **head** equal to the desired paragraph indentation, kept in the global variable, ***para-indent***, it can also handle paragraph indentation. To ensure that the last line of a paragraph is broken properly the stretch of **tail** is set to infinite (10000 here simulates infinity). This will force the algorithm to break the paragraph at **tail**. It should be noted that the function that is used to fill the input-to-node-map role is by default partial. This is so because in general we do not expect all inputs to be mapped onto nodes. From the paragraph justification domain, it is known that the resulting graph is a forward graph with respect to the input sequence from which the graph is constructed: all arcs point forward with respect to the order of the input sequence. The programmer also specifies that the domain of Input-To-Node-Map is the gap set in the new paragraph. Finally the programmer gave the DA an explicit test for arcs given two input tokens. The test says that if the first argument token, **xn**, has a corresponding node (**Input-To-Node-Map xn**) and if the ratio of the subsequence between the the two argument tokens is between 0 and ***threshold***, then there exists an arc from (**Input-To-Node-Map xn**) to the node corresponding to the second argument token. Note that this implies that the second argument token, under Input-To-Node-Map, is mapped to a node.

The programmer indicated in the descriptions that the proration is to be done proportional to the stretch of the tokens. The programmer also instructed the DA to favor the end of the line when prorating the line. This means that the DA should first try to allocate "extras" to the end tokens of the given line. This is the desired behavior since we want **tail** (the last token on the last line) to take up all the excess spaces on the last line. By default, the Prorate cliché favors the beginning elements of the input sequence.

The programmer also specifies how the various functions should be computed. The total length of a sequence is simply the sum of the widths of the sequence. Other functions are similarly defined.

A subtle point in the high-level design of the program should be noted here: the last token on a segment is given a stretch of 10000 so as to force the algorithm to make it a break point. Prorate will give such a gap token all the slack.

2.5.2 Interaction

With the additional information, the DA proceeds to analyze the descriptions. At this point, **para-break** has been defined in the Tokenize cliché as a particular type of token, and the extras function has been explicitly defined. The interaction that follows is given in Figure 2-4 to Figure 2-7.

```
DA> Constraint Violation: The #\newline character is not an Ascii
  character.  Note: The following are standard ways of defining the
  #\newline character in the Ascii character set:
  (1) Define #\newline to be #\linefeed followed by #\return.
  (2) Define #\newline to be #\return.
>>> Use(2).
DA> Constraint Violation: The characters possible in the input to
  Tokenize must be spanned by the characters in the definitions of
  token types.  How should the remaining Ascii characters be handled?
  Note: The following are standard ways of handling the remaining Ascii
  characters:
  (1) Ignore them.
  (2) Treat them as #\Space.
  (3) Flag them as errors.
>>> Explain.
DA> The characters possible in the input to Tokenize must be spanned by
  the characters in the definitions of token types because
  (a) The characters possible in the input to Tokenize is the
  Ascii character set.
  (b) The characters in the definitions of token types in Tokenize consists
  of the Non-Blank-Printable-Characters plus #\return plus #\space.
  (c) "Infile" is an Ascii file.
  (d) An Ascii file contains the Ascii character set.
  (e) The Ascii character set is not spanned by the
  Non-Blank-Printable-Characters plus #\return plus #\space.
>>> Use(2)
```

Figure 2-4: A Constraint Violation and an Explanation.

The dialog in this scene illustrates the interactive nature of the design process that the DA supports through its active error-checking capability. In Figure 2-4 the DA notes that the newline character found in the regular expressions of Tokenize is not an ASCII character. The newline character is a Common Lisp character which is frequently used to denote the sequence of ASCII characters: Linefeed Return, or the ASCII Return character. These methods of recovering from the constraint violation are codified in the Ascii-File cliché. The programmer chooses to define the newline character to be synonymous with the return character.

Next, the DA notes that the given description of Tokenize is incomplete because it does not handle all ASCII characters. This violation arose from the interaction of constraints which came from two sources. First, the Ascii-File cliché has a constraint that defines the ASCII character set. Second, the Tokenize cliché imposes a constraint that the character set has to be fully accounted for, and it contains knowledge about how the remaining characters may be handled. The programmer then asked the DA to explain the question posed. Here, the DA shows its rudimentary explanation capability: it pulls out the dependencies supporting the different assertions that led to the constraint violation at hand.

```
DA> WARNING: ratio is undefined when total-stretch = 0.
>>> (REDESIGN ratio (seq)
      (LET ((extras (extras seq)))
        (IF (= extras 0) 0
            (LET ((total-stretch (total-stretch seq)))
              (IF (= total-stretch 0) 10000
                  (/ extras total-stretch))))))
```

Figure 2-5: A Constraint Violation: Partial Function.

The DA next notices that `ratio` as currently given is undefined if the total-stretch of the sequence is 0, as shown in Figure 2-5. If it could prove to itself that total-stretch of the sequences passed to it is never 0, then it would not have bothered the programmer with such a warning. Here, since it is unable to do so, it reminded the programmer of a commonly-known potential pitfall. The programmer resolved the warning by defining the function to have a value of 0 when extras is 0, and to have a large value of 10000 when extras is not 0 and total-stretch is 0. The first case allows for the situation when a long word takes up the entire line, and the second case prevents the build-graph algorithm from building arcs with non-zero extra spaces and no gaps to which to distribute the spaces.

In Figure 2-6 the DA complains that the default value for the `preceding-item` computation is `nil`. The output of this function, however, is passed to the last-character function which expects its input to be a string. This is a possible runtime error. The programmer decides to change the design accordingly to specifically check for this case before last-character is invoked.

```
DA> WARNING: By default Preceding-Item return Nil for the first
    element in the implicit sequence.  Last-Character expects a string.
    There is a possibility of runtime error.
>>> (REDESIGN width (token)
      (IF (word-token-p token) (Number-Of-Characters token)
          (LET ((previous-token (Preceding-Item token)))
            (IF previous-token
                (IF (Member (Last-Character previous-token)
                            '(#\! #\? #\.))
                    2 1)
                0))))
```

Figure 2-6: A Constraint Violation: A Possible Run-Time Error.

```
DA> In Single-Source-Shortest-Path, what should the start node, the
end node and the distance function be?
>>> (WITHIN Single-Source-Shortest-Path (graph)
      (IS Start-Node (Root graph)) (IS End-Node (Sink graph))
      (IS Distance-Function ratio))
DA> Type Conflict: The domain of Ratio is of type Sequence but the
domain of Distance-Function in Single-Source-Shortest-Path is of type
Arc.
>>> (WITHIN Single-Source-Shortest-Path (graph)
      (IS Distance-Function arc-ratio))
DA> WARNING: arc-ratio is undefined.
>>> (DESIGN arc-ratio (arc) (ratio (token-sequence arc)))
```

Figure 2-7: An Omission in Program Descriptions and a Type Conflict.

The DA notes in Figure 2-7 that the single-source shortest path description is incomplete since the start node, the end node and the distance function to be minimized were not given. The programmer then provides the DA with the needed information.

2.5.3 Output Code

Based on the additional descriptions given, the DA generates the code shown in Figure 2-8 to Figure 2-13. For the sake of brevity, the code for `Internal-Get-Token` is not shown. Note that this could be built by the popular UNIX utility, LEX [17]. Some important salient features of the output code are noteworthy here: the size of the output code is some four times larger than the input program descriptions. The output code is quite complicated, at least when compared to most toy programs used to exercise automatic programming techniques. There are also multiple modules in the output program. Some mappings are pre-computed in the output code even though they were not so indicated by the user: `token-width`, `token-next` and `token-stretch`. The `arc-ratio` mapping is cached while `ratio` is computed so that it is efficiently implemented by a look-up operation rather than re-computed.

Given the program description, there are many detailed design choices left to the DA. Some of the main choices available to the DA are described and the reasons why the DA picked a particular implementation are briefly explained.

Tokenize: The Tokenize cliché allows the programmer to define the patterns to look for in the input sequence of characters. It has knowledge about how to implement a tokenize procedure. For simplicity, we are not showing the various ways tokenization can be achieved in the current scenario. We assume a non-deterministic finite state automata implementation of tokenization.

Independent of the tokenization algorithm used, there are lower level design choices that need to be made before a program can be constructed. For example, the physical representation of a token may vary. The sequence of tokens can also be represented in many ways: it could be implemented as a list, a vector, a doubly-linked list, a chain-shelf or kept implicit in some iteration structure. In [14] Knuth calls an output-restricted deque a shelf. A deque is a linear list for which all insertions and deletions are made at the ends of the list. Deletions are restricted from one end of an output-restricted deque. Here we call a chain of structures which is output-restricted a chain-shelf.

In the scenario, the DA chooses to represent a token, by default, as a Lisp structure. A chain-shelf is used to represent the sequence of tokens since such a data structure efficiently supports the adding of tokens from both the front and the back of the sequence, and scanning subsequences of a token sequence. All these operations are needed in the program.

Build-Graph: The Build-Graph cliché allows the programmer to describe the abstract properties of the graph desired and of the input, in addition to the specific roles needed to construct such a graph. The DA is told that the output graph is

forward-wrt-input. This special condition on the output graph implies that the output graph is acyclic. This lemmas is recorded in the build-graph cliché. The user also specified that the output graph is directed. Thus the DA deduces that the output is a DAG. Based on the property of the given arc-test, and the special properties of the graph to be built, the *build-rooted-forward-dag-with-weak-left-arc-test* algorithm was picked. This algorithm, however, leaves open the implementation of the graph. The DA then used some codified knowledge about graph construction: the implementation of graphs is influenced by the specific processing to be performed on them. In the description, the DA is told that a single-source shortest path algorithm is to be run on the output graph from Build-Graph. Shortest path algorithms typically relax the arcs of the graph in some order or use the adjacency list information of nodes. At this point, the DA left the implementation choice of the graph open: it could be an arc set or an adjacency list since they are equally efficient.

Single-Source-Shortest-Path: As described in an earlier section, there are many different shortest path algorithms applicable here. In the scenario, the DA selects an algorithm, named DAG-SSSP, that is suitable for directed acyclic graphs (DAGS), and which is the fastest applicable algorithm. This selection relies on the fact that the input graph is a DAG which, in turn relies on the user assertion that the output graph of build-graph is forward-wrt-input. The DAG-SSSP algorithm requires that the arcs of the graph be topologically sorted first before the relaxation step. Here the best choice the DA found is one where the graph is implemented as an ordered arc set. In such a case, the arc set is already topologically sorted by the Build-Graph algorithm chosen, and hence the topological sorting step in the shortest path algorithm for DAGS can be omitted.

Prorate: The DA selects the *prorate-favor-end* algorithm and computes the proration proportional to the given stretch function.

Segment: The DA selects the *segment-chain-into-chain-shelves* algorithm and computes the proration proportional to the given stretch function.

Output: The DA chooses the Lisp primitive `write-char` function to implement the use of the output cliché in the justify design. The other uses of the output cliché are turned into `write-n-chars`, `write-string` and `write-char` in `lineout`, in order of their uses in the lineout design.

Mappings: From the program description, the DA deduces that width and stretch are mappings on tokens. They may be explicitly stored or computed on demand. If we decide to store them, we can store them in various ways, in vectors, hash tables, or fields on the structure of the token. Here, the DA pre-computes these mappings and stores them in the fields of the token structure.

Similarly, the DA finds the following mappings on arcs: *source*, *destination*, and *arc-ratio*. The DA chooses to represent an arc as a Lisp structure and the functions on arcs as fields of the structure, similar to the implementation of the token. Note that since `total-length` and `total-stretch` of an arc is not needed in later computation,

they are computed on demand.

Two local mappings in the shortest path algorithm, *min-distance* and *best-previous-arc* are represented as local vectors by the DA.

Preceding-Item: The computation underlying this cliché expects a current item and an underlying sequence of items in which to search for the item just before the given item. In this use, the DA takes the underlying sequence to be the domain sequence associated with the top-level function that uses it, by default. In general, the only way to find the item preceding a given one in a sequence is to scan the whole sequence, matching the given item with every item of the sequence. In the output code, the DA chooses to pre-compute the width mapping, and since the pre-computation requires scanning the entire domain sequence, the DA uses a series function **previous** to pick out the previous item during the scanning. **previous** does this by remembering the previous item during the scanning process so it is done efficiently.

A Brief Guide for Reading the Output Code

The output code to be generated by the DA uses the Common Lisp SERIES macro package [35]. The series macro package is used because it contains a rich set of powerful iteration abstractions that allow the DA to avoid reasoning about loops directly. A few notes about the series functions used in the output code are also sketched here to help the reader understand the output code. Detail specifications of the series functions can be obtained from [35].

The beginning forms in Figure 2-8 show the main data structures used to represent a number of types implicitly specified in the given program descriptions: tokens, nodes, arcs, and graphs. Their fields are needed to implement several mappings defined on their respective types.

The **Justify** Lisp function is similar to the program description given except that the DA inserts code to coerce the input ASCII file into an output stream. **Iterate** is a series iteration construct, similar to **Dolist** in Lisp, but it works on series instead of lists. **Scan** is a series function that scans a list into a series. Like **Justify**, the structure of **Lineout** mirrors its program description.

The next function shown in Figure 2-8 is **Build-Graph**. It shows the extra tokens being made and appended to the input paragraph being passed on an auxiliary function, **Internal-Build-Graph**. **Choose-If** is a series function that selects out of a series those elements that passed the predicate test provided. **Scan-Chain-Shelf** scans a chain-shelf into a series; it needs the successor function of the chain-shelf by which the successor of an element in the chain-shelf can be obtained. Token-Chain-Shelf-Endcons and Token-Chain-Shelf-Cons are operations to add tokens to a token sequence represented as a chain shelf, from the front and from the back, respectively. In **Token-Sequence**, **Scan-Chain** is a macro that scans subsequences of a sequence represented as a chain when two elements of the sequence are given. The keywords indicate whether the end elements themselves should be included. In

Extras, **Collect-Sum** is a series function that sums up a series of numbers. **#M** is a series reader macro that expands into its **Mapping** macro which maps a function over a series, similar to the **Mapcar** Lisp function but instead, it returns a series.

The first function in Figure 2-9 shows **Tokenize**. It is a series function as indicated by an explicit series declaration. It relies on the **Internal-Get-Token** abstraction to return a series of tokens as a result of tokenizing the input file. It is a non-deterministic finite state automata implementation of tokenization, which is elided for brevity.

Tokenize precomputes three token mappings: width, stretch, and the successor function of the output token sequence of tokenization. The second function in the figure shows the code for **Segment**. This function is written in terms of **series** primitives which are explained in [35]. It segments a sequence represented as a chain into a series of chain-shelves according to the given predicate.

Figure 2-10 shows the actual build graph algorithm chosen by the DA, rendered as **Internal-Build-Graph**. The main structure of the function is derived from the *build-rooted-forward-dag-with-weak-left-arc-test* cliché codified in the cliché library. **Collect-Fn** is a series function that supports the concept of reducing a series of data to some non-series data, similar to the Lisp **Reduce** function. It is a collector with internal state. It takes four arguments. The initial state is provided by the second argument in the form of a function, and how the states are to be transformed from one state to another is specified as the third argument. The first argument indicates the type of values returned by the third argument. The fourth argument is a series of values. For example, for the inner **Collect-Fn** in **Internal-Build-Graph**, its internal state has five data that are passed from one iteration to another, its second argument is a function with six inputs, the first five are the data in the internal state from the last iteration, and the sixth is the corresponding element in its input series.

Figure 2-11 shows the **Single-Source-Shortest-Path** function which comes from the *path-of-dag-sssp* cliché. In that function, **Scan-Fn-Inclusive** is used. It is a series function that produces a series starting with some non-series seed datum. Its last argument, an end-test predicate, indicates when the scanning should stop. The syntax of **Scan-Fn-Inclusive** syntax is similar to that of **Collect-Fn**. In the figure, it is used to extract the optimal path at the end of the relaxation step in the shortest path algorithm.

Figure 2-11 also shows the *prorate-favor-end* algorithm chosen by the DA. The **Collecting-Fn** used in **Prorate** is a series function that is similar to **Collect-Fn** but it returns the intermediate data values in the state as series.

All the auxiliary functions used in the above discussion are shown in the last two figures, Figure 2-12 and Figure 2-13. The **Scan-Chain** macro in Figure 2-13 uses the series function **Scan-Fn**. This is identical to **Scan-Fn-Inclusive** discussed earlier except that unlike the latter, which returns the last datum that passed its end-test predicate, **Scan-Fn** does not. **Subseries** is a series function that returns a sub-series of the input series, similar to the **Subseq** function in Lisp.

```
; main data structures used by Justify
(Defstruct (Token-Struct (:Conc-Name Nil))
  Token-Type Token-Next Token-Content Token-Width Token-Stretch)
(Defstruct (Node-Struct (:Conc-Name Nil))
  Node-Index Node-Token Node-Next)
(Defstruct (Arc-Struct (:Conc-Name Nil))
  Arc-Source Arc-Destination Arc-Next Arc-Ratio)
(Defstruct (Graph-Struct (:Conc-Name Nil))
  Graph-Root Graph-Arc-Sequence Graph-Node-Set-Size Graph-Sink)
(Defun Justify (Input-File Output-File)
  ; Justify: String X String -> Nil
  (With-Open-File (Outstream Output-File :Direction :Output)
    (Iterate ((Paragraph (Segment (Tokenize Input-File)
                                  #'Para-Break-Token-P)))
      (Iterate ((Line (Scan (Single-Source-Shortest-Path
                             (Build-Graph Paragraph)))))
        (Lineout Outstream Line))
      (Write-Char #\Return Outstream))))
(Defun Lineout (Outstream Line-Arc)
  ; Lineout: Stream X Arc -> Nil
  (Let* ((Token-Seq (Token-Sequence Line-Arc))
         (Prorated (Prorate (Extras Token-Seq) (Collect Token-Seq))))
    (Iterate ((Amount Prorated) (Token (Token-Sequence Line-Arc)))
      (If (Gap-Token-P Token)
          (Write-N-Chars #\Space (+ (Token-Width Token) Amount) Outstream)
          (Write-String (Token-Content Token) Outstream))))
  (Write-Char #\Return Outstream))
(Defun Build-Graph (Paragraph)
  (Let* ((Head (Make-Token-Struct :Token-Type ':Gap :Token-Width
                                  *Para-Indent* :Token-Stretch 0))
         (Tail (Make-Token-Struct :Token-Type ':Gap :Token-Width 0
                                  :Token-Stretch 10000)))
    (Internal-Build-Graph Head
     (Choose-If
      #'Gap-Token-P
      (Scan-Chain-Shelf
       #'Token-Next
       (Token-Chain-Shelf-Endcons
        Tail (Token-Chain-Shelf-Cons Head Paragraph)))))))
(Defun Token-Sequence (Arc)
  (Declare (Optimizable-Series-Function))
  (Scan-Chain #'Token-Next (Node-Token (Arc-Source Arc)) :Include-Start Nil
              :Before (Node-Token (Arc-Destination Arc))))
(Defun Extras (Tokens)
  (Declare (Optimizable-Series-Function) (Series Tokens))
  (- *Line-Length* (Collect-Sum (#Mtoken-Width Tokens))))
```

Figure 2-8: Initial code created by the DA (part 1).

```
(Defun Tokenize (Input-File)
  (Declare (Optimizable-Series-Function))
  ; Justify: String -> (Series Token)
  (Let* ((Tokens (Internal-Get-Tokens Input-File))
         (Tokens-1
           (Mapping ((Current-Token Tokens)
                     (Previous-Token (Previous Tokens Nil)))
             (When Previous-Token
               (Setf (Token-Next Previous-Token) Current-Token))
             Current-Token))
         (Tokens-2
           (Mapping ((Current-Token Tokens-1))
             (Setf (Token-Stretch Current-Token)
                   (If (Gap-Token-P Current-Token) 1 0))
             Current-Token)))
    (Mapping ((Current-Token Tokens-2)
              (Previous-Token (Previous Tokens-2 Nil)))
      (Setf (Token-Width Current-Token)
            (If (Word-Token-P Current-Token)
                (Length (Token-Content Current-Token))
                (If Previous-Token
                    (If (Member (Last-Character
                                  (Token-Content Previous-Token))
                                '(#\! #\? #\.))
                        2 1) 0)))
      Current-Token)))
(Defun Segment (Tseries Predicate)
  ; Specific plan used: Segment-Chain-Into-Chain-Shelves
  ; Segment: (Series T) X (T -> Bool) -> (Series (Pair T T))
  ; This segments the input series into segments separated by input
  ; elements that pass the predicate test.  Each segment is represented
  ; by a pair of the first element and the last element of that segment.
  ; Input should be an explicit series of chain.  Output is a series of
  ; chain-shelves (having the same implicit  successor function as
  ; the input chain).  Returns empty series if input series is empty.
  (Declare (Optimizable-Series-Function) (Series Tseries)
           (Off-Line-Port Tseries))
  (Producing (Outs) ((Ins Tseries) Item Previous
                     Node (Result Nil) (Done-Flag Nil))
    (Loop
      (Tagbody
        L (When Done-Flag (Terminate-Producing))
          (Setq Previous Item)
          (Setq Item (Next-In Ins (Setq Done-Flag T Node T) (Go S)))
          (Setq Node (Funcall Predicate Item))
        S (When (Null Result)
            (If Node (Go L))
            (Setq Result (Cons Item Nil)))
          (When (Not Node) (Go L))
          (Next-Out Outs (Prog1 (Rplacd Result Previous)
                                (Setq Result Nil)))))))
```

Figure 2-9: Initial code created by the DA (Part 2).

```
(Defun Internal-Build-Graph (Head Paragraph)
  ; Specific plan used: Build-Rooted-Forward-Dag-With-Weak-Left-Arc-Test
  ; Build-Graph: (Pair Token Token) -> Rooted-Dag
  (Declare (Optimizable-Series-Function) (Series Paragraph))
  (Multiple-Value-Bind (Arcs Queue Index Current-Node)
    (Collect-Fn
      '(Values T T T T)
      #'(Lambda ()
          (Let ((Root (Make-Node-Struct :Node-Token Head :Node-Index 0)))
            (Values Nil (Make-Chain-Shelf Root Root) 1 Nil)))
      #'(Lambda (Arcs Queue Index Current-Node Current-Token)
          (Multiple-Value-Bind (Arcs Queue Index Current-Token
                                Current-Node)
            (Collect-Fn '(Values T T T T T)
              #'(Lambda () (Values Arcs Queue Index Current-Token Nil))
              #'(Lambda (Arcs Queue Index Current-Token
                         Current-Node Queue-Node)
                  (Let* ((Queue-Token (Node-Token Queue-Node))
                         (Tokens (Scan-Chain #'Token-Next
                                             Queue-Token :Include-Start
                                             Nil :Before Current-Token))
                         (Total-Stretch
                          (Collect-Sum (#Mtoken-Stretch Tokens)))
                         (Extras (Extras Tokens))
                         (Ratio (If (= Extras 0) 0
                                    (If (= Total-Stretch 0) 10000
                                        (/ Extras Total-Stretch)))))
                    (When (And (>= Ratio 0) (<= Ratio *Threshold*))
                      (When (Null Current-Node)
                        (Setq Current-Node
                              (Make-Node-Struct
                                :Node-Token Current-Token
                                :Node-Index Index))
                        (Setq Index (+ Index 1))
                        (Setq Queue (Node-Chain-Shelf-Endcons
                                     Current-Node Queue)))
                      (Let ((Arc (Make-Arc-Struct
                                  :Arc-Source Queue-Node
                                  :Arc-Destination Current-Node)))
                        (Setf (Arc-Ratio Arc) Ratio)
                        (Setq Arcs (Arc-Chain-Shelf-Endcons Arc Arcs)))))
                  (Values Arcs Queue Index Current-Token Current-Node))
                (Scan-Chain-Shelf #'Node-Next Queue))
              (Values Arcs Queue Index Current-Node)))
      Paragraph)
    (Make-Graph-Struct :Graph-Sink Current-Node
                       :Graph-Root (Chain-Shelf-Head Queue)
                       :Graph-Node-Set-Size Index
                       :Graph-Arc-Sequence Arcs)))
```

Figure 2-10: Initial code created by the DA (part 3).

```
(Defun Single-Source-Shortest-Path (Graph)
  ; Specific plan used: Path-Of-Dag-Sssp
  ; Single-Source-Shortest-Path: Graph -> (Series Arc)
  (Let* ((Root (Graph-Root Graph)) (Sink (Graph-Sink Graph))
         (Size (Graph-Node-Set-Size Graph))
         (Min-Distance (Make-Array Size))
         (Best-Previous-Arc (Make-Array Size)))
    (Iterate ((Index (Scan-Range :Below (Graph-Node-Set-Size Graph))))
      (Setf (Svref Min-Distance Index) *Infinity*))
    (Setf (Svref Min-Distance (Node-Index Root)) 0)
    (Iterate ((Arc (Scan-Chain-Shelf #'Arc-Next
                    (Graph-Arc-Sequence Graph))))
      (Let* ((Destination-Index (Node-Index (Arc-Destination Arc)))
             (Distance-Via-Node (+ (Svref Min-Distance
                                          (Node-Index (Arc-Source Arc)))
                                   (Arc-Ratio Arc))))
        (If (> (Svref Min-Distance Destination-Index) Distance-Via-Node)
            (Setf (Svref Min-Distance Destination-Index) Distance-Via-Node
                  (Svref Best-Previous-Arc Destination-Index) Arc))))
    (Nreverse
      (Collect
        (Scan-Fn-Inclusive T
          #'(Lambda () (Svref Best-Previous-Arc (Node-Index Sink)))
          #'(Lambda (Arc) (Svref Best-Previous-Arc
                                 (Node-Index (Arc-Source Arc))))
          #'(Lambda (Arc) (Eq (Arc-Source Arc) Root)))))))
(Defun Prorate (Amount Items)
  ; Specific plan used: Prorate-Favor-End
  ; Prorate: Number X (List Items) -> (Series Number)
  ; takes in a list of shares and returns a series of the amount
  ; prorated across the shares favoring the end.  if all shares are
  ; zero the whole amount goes on the first share.  the only time the
  ; sum of the output is not equal to amount is when there are no shares
  ; and therefore no output.
  (Declare (Optimizable-Series-Function) (Series Shares))
  (Let ((Total-Shares (Collect-Sum (#MToken-Stretch (Scan Items)))))
    (Collecting-Fn '(Values Integer Realnum Realnum)
      #'(Lambda () (Values 0 Amount Total-Shares))
      #'(Lambda (Last-Share Unallocated-Amount Remaining-Shares Share)
          (Declare (Ignore Last-Share))
          (Let ((Allocation
                  (If (= Remaining-Shares Share) Unallocated-Amount
                      (Floor (* Unallocated-Amount Share)
                             Remaining-Shares))))
            (Values Allocation (- Unallocated-Amount Allocation)
                    (- Remaining-Shares Share))))
      (#MToken-Stretch (Scan Items)))))
```

Figure 2-11: Initial code created by the DA (part 4).

```
; these are pre-defined functions available to the DA:
(Defun Last-Character (String)
  ; Last-Character: String -> (*Or Character Nil)
  (Let ((Index (1- (Length String))))
    (If (> Index -1) (Char String Index))))
(Defun Write-N-Chars (Char N Stream)
  ; Write-N-Chars: Stream -> Integer
  ; works by side-effects.
  (Iterate ((I (Scan-Range :Below N)))
    (Declare (Ignore I))
    (Write-Char Char Stream)))
; the next 3 functions are generated by the Tokenize cliche
;  with the given input specs.
(Defun Word-Token-P (Token)
  ; Word-Token-P: Token -> Bool
  (And (Typep Token 'Token-Struct) (Equal (Token-Type Token) :Word)))
(Defun Gap-Token-P (Token)
  ; Gap-Token-P: Token -> Bool
  (And (Typep Token 'Token-Struct) (Equal (Token-Type Token) :Gap)))
(Defun Para-Break-Token-P (Token)
  ; Para-Break-Token-P: Token -> Bool
  (And (Typep Token 'Token-Struct) (Equal (Token-Type Token) :Para-Break)))
(Defun Make-Chain-Shelf (Head Tail) (Cons Head Tail))
(Defun Chain-Shelf-Head (Chain-Shelf) (Car Chain-Shelf))
(Defun Chain-Shelf-Tail (Chain-Shelf) (Cdr Chain-Shelf))
(Defun Scan-Chain-Shelf (Next-Function Chain-Shelf)
  ; requires chain-shelf to be non-empty
  (Declare (Optimizable-Series-Function))
  (Scan-Chain Next-Function (Chain-Shelf-Head Chain-Shelf)
              :To (Chain-Shelf-Tail Chain-Shelf)))
```

Figure 2-12: Initial code created by the DA (part 5).

```
; convenient type declarations that Common Lisp lacks (useful names)
(Deftype Realnum () '(Or Float Integer))
(Deftype Bool () '(Member T Nil))
(Deftype Predicate () '(Function T (Member T Nil)))
(Defmacro Scan-Chain (Next-Function Start &Key (Include-Start T)
                      (Before Nil Before-P) (To Nil To-P))
  ; check to make sure that :before and :to are not both given.
  (If (And Before-P To-P)
      (Error "~% Scan-Chain: :before and :to cannot be both given."))
  (If Include-Start
     (If Before-P
        `(Scan-Fn T #'(Lambda () ,Start)
                  ,Next-Function #'(Lambda (X) (Eq X ,Before)))
       (If To-P
          `(Scan-Fn-Inclusive T #'(Lambda () ,Start)
                              ,Next-Function #'(Lambda (X) (Eq X ,To)))
         `(Scan-Fn T #'(Lambda () ,Start) ,Next-Function #'Null)))
    (If Before-P
       `(Subseries
         (Scan-Fn T #'(Lambda () ,Start) ,Next-Function
                  #'(Lambda (X) (Eq X ,Before))) 1)
      (If To-P
         `(Subseries
           (Scan-Fn-Inclusive T #'(Lambda () ,Start) ,Next-Function
                              #'(Lambda (X) (Eq X ,To))) 1)
        `(Subseries (Scan-Fn t #'(lambda () ,start)
                             ,next-function #'null) 1)))))
(Defun Node-Chain-Shelf-Endcons (Node Node-Chain-Shelf)
  (If (Null Node-Chain-Shelf) (Make-Chain-Shelf Node Node)
      (Progn (Setf (Node-Next (Chain-Shelf-Tail Node-Chain-Shelf)) Node)
             (Make-Chain-Shelf (Chain-Shelf-Head Node-Chain-Shelf) Node))))
(Defun Arc-Chain-Shelf-Endcons (Arc Arc-Chain-Shelf)
  (If (Null Arc-Chain-Shelf) (Make-Chain-Shelf Arc Arc)
      (Progn (Setf (Arc-Next (Chain-Shelf-Tail Arc-Chain-Shelf)) Arc)
             (Make-Chain-Shelf (Chain-Shelf-Head Arc-Chain-Shelf) Arc))))
(Defun Token-Chain-Shelf-Endcons (Token Token-Chain-Shelf)
  (If (Null Token-Chain-Shelf) (Make-Chain-Shelf Token Token)
      (Progn (Setf (Token-Next (Chain-Shelf-Tail Token-Chain-Shelf)) Token)
             (Make-Chain-Shelf (Chain-Shelf-Head Token-Chain-Shelf)
                               Token))))
(Defun Token-Chain-Shelf-Cons (Token Token-Chain-Shelf)
  (If (Null Token-Chain-Shelf) (Make-Chain-Shelf Token Token)
      (Progn (Setf (Token-Next Token) (Chain-Shelf-Head Token-Chain-Shelf))
             (Make-Chain-Shelf Token
                               (Chain-Shelf-Tail Token-Chain-Shelf)))))
```

Figure 2-13: Initial code created by the DA (part 6).

2.6 Scene 3: Explaining Design Rationale

In this scene, the programmer queries the DA about the detailed design of the program. The tags on the DA's output facilitate the interaction between the DA and the programmer; they allow the programmer to refer to the keys for further explanation. This scene serves to illustrate the design rationale used by the DA to arrive at the chosen implementations (see Figure 2-14). There are many explicit design decisions the DA has to make in order to arrive at the implementation shown earlier; only some of the more interesting ones are shown here. Since the interaction reproduces some of the reasons discussed at length in the last section, we shall not elaborate here.

```
>>> Explain-Design.
DA> The following subsidiary design choices have been made:
 1. A structure named Token-Struct is used to represent a Token in Tokenize.
 2. A Chain-Shelf is used to represent the token sequence.
 3. The Build-Rooted-Forward-Dag-With-Weak-Left-Arc-Test cliche has been
 chosen to implement Build-Graph.
 4. A structure named Node-Struct is used to represent a Node in the Graph.
 5. An ordered arc set has been chosen to represent Graph.
 6. The Dag-Sssp cliche has been chosen to implement
 Single-Source-Shortest-Path.
 7. The Prorate-Favor-End cliche has been chosen to implement Prorate.
>>> Why(3)
DA> The Build-Rooted-Forward-Dag-With-Weak-Left-Arc-Test cliche is chosen
 because
 (3.1) It is a Build-Graph cliche.
 (3.2) It is rooted.
 (3.3) It is forward-wrt-input.
 (3.4) Its Arc-Test is weak-left.
 (3.5) Favor time efficiency.
>>> Why(5)
DA> Graph is represented as an ordered arc-set because:
 (5.1) An ordered arc-set can be used to represent the graph.
 (5.2) The graph is to be used by Single-Source-Shortest-Path
 in which arcs are enumerated.
 (5.3) The topological sorting step in the Dag-Sssp algorithm can be
 eliminated.
>>> Why(5.3)
DA> The topological-sorting step in the Dag-Sssp algorithm can be
 eliminated because:
 (5.3.1) The first step in Dag-Sssp enumerates the input arc-set of
 Graph in topologically-sorted order.
 (5.3.2) The output arc-set of Build-Rooted-Forward-Dag-With-Weak-Left-Arc-Test
 is topologically-sorted.
 (5.3.3) The output arc-set of Build-Rooted-Forward-Dag-With-Weak-Left-Arc-Test
 is the input arc-set of the Dag-Sssp algorithm.
>>> Why(6)
DA> The Dag-Sssp cliche was chosen because:
 (6.1) Favor time efficiency.
 (6.2) Dag-Sssp is a Single-Source-Shortest-Path cliche.
 (6.3) Input graph to Single-Source-Shortest-Path is directed.
 (6.4) Input graph to Single-Source-Shortest-Path is acyclic.
 (6.5) Most-Efficient(Possible-Implementations(Sssp)) = Dag-Sssp.
>>> Value-Of(Time-Efficiency(Rooted-Dag-Sssp))
DA> DAG-SSSP has time complexity O(m+n) where m is the size of the
 arc-set, and n is the size of the node-set.
>>> Why(2)
DA> A Chain-Shelf is used to represent the token sequence because
 (2.1) A Chain-Shelf supports the following operations well:
 EndCons, Cons and Sequence-In-Between.
```

Figure 2-14: The Rationale of Automated Design Decisions.

2.7 Scene 4: Adding a Guideline

The programmer observes that local vectors are used to implement the **min-distance** and **best-previous-arc** functions in the shortest path algorithm. The DA tries to keep information useful to local computations close to those computations. In this case, the programmer prefers the storage to be distributed among the domain elements.

```
(IMPLEMENTATION-GUIDELINES
  (PREFER Distributed-Mapping Single-Source-Shortest-Path))
```

Figure 2-15: Adding a New Implementation Guideline.

Note that in Figure 2-16 and Figure 2-17, the **node-index** field of **node-struct** is no longer used. The only reason why the **node-index** field was needed in the earlier scene was so that the **min-distance** array could be accessed when given a node. All the relevant information is kept on the **node-struct** structure itself, and so the **node-index** is no longer needed. Similarly, the **graph-node-set-size** is no longer needed and is replaced by **graph-node-set**. The latter is needed in the new implementation of the two mappings. In the output code, the basic shortest path algorithm is not changed, only the way the mappings are accessed and modified are changed.

This scene shows how the user's implementation guidelines can influence the choices made by the DA. It also illustrates the use of explicit design dependencies kept by the DA in order to remove the **node-index** and **graph-node-set-size** once they are no longer useful. It shows the kinds of services the DA can provide in supporting design changes.

```
(Defstruct (Node-Struct (:Conc-Name Nil))
  Node-Token Node-Next Node-Min-Distance Node-Previous-Arc)
(Defstruct (Graph-Struct (:Conc-Name Nil))
  Graph-Root Graph-Arc-Sequence Graph-Node-Set Graph-Sink)
                                ; was Graph-Node-Set-Size
(Defun Single-Source-Shortest-Path (Graph)
  ; Specific plan used: Path-Of-Dag-Sssp
  (Let ((Root (Graph-Root Graph))
        (Sink (Graph-Sink Graph)))
    (Iterate ((Node (Scan-Chain-Shelf #'Node-Next (Graph-Node-Set Graph))))
      (Setf (Node-Min-Distance Node) *Infinity*))
    (Setf (Node-Min-Distance Root) 0)
    (Iterate ((Arc (Scan-Chain-Shelf #'Arc-Next
                                     (Graph-Arc-Sequence Graph))))
      (Let* ((Destination-Node (Arc-Destination Arc))
        (Distance-Via-Node (+ (Node-Min-Distance (Arc-Source Arc))
                              (Arc-Ratio Arc))))
        (If (> (Node-Min-Distance Destination-Node) Distance-Via-Node)
            (Setf (Node-Min-Distance Destination-Node) Distance-Via-Node
                  (Node-Previous-Arc Destination-Node) Arc))))
    (Nreverse
      (Collect (Scan-Fn-Inclusive T
                #'(Lambda () (Node-Previous-Arc Sink))
                #'(Lambda (Arc) (Node-Previous-Arc (Arc-Source Arc)))
                #'(Lambda (Arc) (Eq (Arc-Source Arc) Root)))))))
```

Figure 2-16: Scene 4: Modified Code (Part 1).

```
; note that node-index is no longer used.
(Defun Internal-Build-Graph (Head Paragraph)
  ; Specific plan used: Build-Rooted-Forward-Dag-With-Weak-Left-Arc-Test
  (Declare (Optimizable-Series-Function) (Series Paragraph))
  (Multiple-Value-Bind (Arcs Queue Index Current-Node)
    (Collect-Fn
      '(Values T T T T)
      #'(Lambda ()
          (Let ((Root (Make-Node-Struct :Node-Token Head)))
                    ; no setf of node-index
            (Values Nil (Make-Chain-Shelf Root Root) 1 Nil)))
      #'(Lambda (Arcs Queue Index Current-Node Current-Token)
          (Multiple-Value-Bind (Arcs Queue Index Current-Token
                                Current-Node)
            (Collect-Fn
              '(Values T T T T T)
              #'(Lambda () (Values Arcs Queue Index Current-Token Nil))
              #'(Lambda (Arcs Queue Index Current-Token
                         Current-Node Queue-Node)
                  (Let* ((Queue-Token (Node-Token Queue-Node))
                         (Tokens (Scan-Chain #'Token-Next
                                             Queue-Token :Include-Start
                                             Nil :Before Current-Token))
                         (Total-Stretch
                          (Collect-Sum (#Mtoken-Stretch Tokens)))
                         (Extras (Extras Tokens))
                         (Ratio (If (= Extras 0) 0
                                    (If (= Total-Stretch 0) 10000
                                        (/ Extras Total-Stretch)))))
                    (When (And (>= Ratio 0) (<= Ratio *Threshold*))
                          (When (Null Current-Node)
                                (Setq Current-Node
                                      (Make-Node-Struct
                                       :Node-Token Current-Token))
                                      ; no setf of node-index
                                (Setq Index (+ Index 1))
                                (Setq Queue (Node-Chain-Shelf-Endcons
                                             Current-Node Queue)))
                          (Let ((Arc (Make-Arc-Struct
                                      :Arc-Source Queue-Node
                                      :Arc-Destination Current-Node)))
                               (Setf (Arc-Ratio Arc) Ratio)
                               (Setq Arcs
                                     (Arc-Chain-Shelf-Endcons Arc Arcs)))))
                  (Values Arcs Queue Index Current-Token Current-Node))
              (Scan-Chain-Shelf #'Node-Next Queue))
            (Values Arcs Queue Index Current-Node)))
        Paragraph)
    (Make-Graph-Struct :Graph-Sink Current-Node
                       :Graph-Root (Chain-Shelf-Head Queue)
                       :Graph-Node-Set Queue
                       ; was :Graph-Node-Set-Size Index
                       :Graph-Arc-Sequence Arcs)))
```

Figure 2-17: Scene 4: Modified Code (Part 2).

2.8 Scene 5: A Correction

Now that the programmer is satisfied with the description, the output program is run on a test file. The programmer notices that the beginning of the line is not indented correctly. Looking at the descriptions once more, the programmer realizes that both uses of *sequence-in-between* by default do not include the end points they are given, and hence the **head** token is not in the output. The description for Sequence-In-Between can be changed to include both end points. However, this change will cause the end gap tokens of internal lines to be output twice: once as the end token of the previous line and the second time as the begin token of the current line. After some thinking, the programmer comes up with an idea: four tokens will be added instead of just **head** and **tail**. The programmer renames **head** and **tail** to **para-head** and **para-tail** respectively. A new **head** is placed before **para-head** and a new **tail** is placed after **para-tail**. Both **head** and **tail** will have 0 width and 0 stretch. The resulting modified code is given in Figure 2-20. The programmer runs the same test file with the modified program; and this time, the expected result is obtained. The new result file is shown in Figure 2-21.

```
Our approach is influenced by our view on the nature of
programming. Hence, before delving into the depths of our
approach, we briefly characterize our view of programming.

Programming is Knowledge-Intensive: Different sources of
knowledge are required. Knowledge of data structures and
algorithms are key components of programming, so are program
structuring techniques, program specification, and knowledge
about the application domain. We believe that much of
the knowledge needed in programming can be codified so
that a computer program can make use of it mechanically.

Requirements Constantly Change: Many changes are constantly
being made to programs due to constantly changing needs.
This is not just evident in program maintenance but
also during the development process. Program maintenance,
especially for large systems, is difficult, because much
of the design rationale is not written down; and where
it is written down, it is usually not kept up-to-date.
```

Figure 2-18: The Result File from Justify.

```
(MODIFY-WITHIN Build-Graph (paragraph)
  (LET ((new-paragraph (Concatenate head para-head paragraph
                                    para-tail tail)))
    (IS (Domain Input-To-Node-Map) (Gap-Set new-paragraph))
    (DECLARE (Directed (Output Build-Graph))
             (Forward-Wrt-Input (Output Build-Graph)))
    (IS Root-Item Head)
    (IS Arc-Test
        (Lambda ((xn token) (yn token))
          (LET ((tsequence (Sequence-In-Between xn yn new-paragraph)))
            (And (Node (Input-To-Node-Map xn))
                 (>= (Ratio tsequence) 0)
                 (<= (Ratio tsequence) *Threshold*)))))))
(== para-head (INSTANCE 'gap-token :width *Para-Indent* :stretch 0))
(== para-tail (INSTANCE 'gap-token :width 0 :stretch 10000))
(== head (INSTANCE 'gap-token :width 0 :stretch 0))
(== tail (INSTANCE 'gap-token :width 0 :stretch 0))
```

Figure 2-19: Adding Additional New Gap Tokens.

```
(Defun Build-Graph (Paragraph)
  (Let* ((Head (Make-Token-Struct :Token-Type ':Gap :Token-Width 0
                                  :Token-Stretch 0))
         (Para-Head (Make-Token-Struct :Token-Type ':Gap :Token-Width
                                       *Para-Indent* :Token-Stretch 0))
         (Para-Tail (Make-Token-Struct :Token-Type ':Gap :Token-Width 0
                                       :Token-Stretch 10000))
         (Tail (Make-Token-Struct :Token-Type ':Gap :Token-Width 0
                                  :Token-Stretch 0)))
    (Internal-Build-Graph
     Head
     (Choose-If
      #'Gap-Token-P
      (Scan-Chain-Shelf
       #'Token-Next
       (Token-Chain-Shelf-Endcons
        Tail
        (Token-Chain-Shelf-Endcons
         Para-Tail
         (Token-Chain-Shelf-Cons
          Head (Token-Chain-Shelf-Cons Para-Head Paragraph))))))))))
```

Figure 2-20: Scene 5: Modified Code.

```
Our approach is influenced by our view on the nature of
programming.  Hence, before delving into the depths  of  our
approach, we briefly characterize our view of programming.

Programming is Knowledge-Intensive:  Different  sources
of knowledge are required.  Knowledge of data structures and
algorithms are key components of programming, so are program
structuring techniques, program specification, and knowledge
about the application domain.  We believe that much  of  the
knowledge needed in programming can be codified  so  that  a
computer program can make use of it mechanically.

Requirements Constantly Change:  Many  changes  are
constantly being made to programs due to constantly changing
needs.  This is not just evident in program maintenance  but
also during the development process.  Program  maintenance,
especially for large systems, is difficult, because much  of
the design rationale is not written down; and  where  it  is
written down, it is usually not kept up-to-date.
```

Figure 2-21: New Result File from Modified Justify.

2.9 Scene 6: The Complete Description

The DA performs its tasks based on its interpretation of the given program description. The net cumulative description is maintained by the DA and is given in Figure 2-22 and Figure 2-23. If the programmer had given the DA the final program descriptions, it would have output the same code shown in the last scene.

```
(DESIGN justify (infile outfile)
  (Ascii-File infile) (Ascii-File outfile)
  (FOR-EACH ((paragraph (Segment (Tokenize infile) 'para-break)))
    (FOR-EACH ((line (Single-Source-Shortest-Path
                      (Build-Graph paragraph))))
      (lineout outfile line))
    (Output outfile #\return)))
(DESIGN lineout (line output-file)
  (LET* ((token-seq (token-sequence line))
         (prorated (Prorate (extras token-seq) token-seq)))
    (FOR-EACH ((amount prorated) (token token-seq))
     (IF (Gap-Token-P token)
         (Output output-file #\space :number
                 (+ (Width token) amount))
         (Output output-file (Content token))))
    (Output output-file #\return)))
(WITHIN Tokenize ()
  (== para-break
      (regular-expression
        "BOF (#\space | #\return)* | (#\space | #\return)* EOF |
        (#\space)* #\return (#\space)* #\return (#\space | #\return)*"))
  (== word (regular-expression "(Non-Blank-Printable-Characters)+"))
  (== gap (regular-expression "(#\space | #\return)+"))
  (Treat others #\space))
(WITHIN Build-Graph (paragraph)
  (LET ((new-paragraph (Concatenate head para-head paragraph
                                    para-tail tail)))
    (IS (Domain Input-To-Node-Map) (Gap-Set new-paragraph))
    (DECLARE (Directed (Output Build-Graph))
             (Forward-Wrt-Input (Output Build-Graph)))
    (IS Root-Item Head)
    (IS Arc-Test
        (Lambda ((xn token) (yn token))
          (LET ((tsequence (Sequence-In-Between xn yn new-paragraph)))
            (And (Node (Input-To-Node-Map xn))
                 (>= (Ratio tsequence) 0)
                 (<= (Ratio tsequence) *Threshold*)))))))
```

Figure 2-22: Program Description of Justify (Part 1).

```
(IMPLEMENTATION-GUIDELINES
  (PREFER Time-Efficiency)
  (IGNORE Error-Checking)
  (PREFER Distributed-Mapping Single-Source-Shortest-Path))
(WRITE-CODE 'justify)
(WITHIN Prorate ()
  (IS Favor End)
  (IS Proportional-To stretch))
(WITHIN Single-Source-Shortest-Path (graph)
  (IS Start-Node (Root graph))
  (IS End-Node (Sink graph))
  (IS Distance-Function arc-ratio))
(DESIGN arc-ratio (arc) (ratio (token-sequence arc)))
(DESIGN extras (seq) (- *Line-Length* (total-length seq)))
(DESIGN ratio (seq)
  (LET ((extras (extras seq)))
    (IF (= extras 0) 0
        (LET ((total-stretch (total-stretch seq)))
          (IF (= total-stretch 0) 10000
              (/ extras total-stretch))))))
(DESIGN total-length (seq) (Sum (Map width seq)))
(DESIGN total-stretch (seq) (Sum (Map stretch seq)))
(== para-head (INSTANCE 'gap-token :width *Para-Indent* :stretch 0))
(== para-tail (INSTANCE 'gap-token :width 0 :stretch 10000))
(== head (INSTANCE 'gap-token :width 0 :stretch 0))
(== tail (INSTANCE 'gap-token :width 0 :stretch 0))
(DESIGN stretch (token) (IF (Gap-Token-P token) 1 0))
(DESIGN width (token)
  (IF (Word-Token-P token) (Number-Of-Characters (Content token))
      (LET ((previous-token (Preceding-Item token)))
        (IF previous-token
            (IF (Member (Last-Character (Content previous-token))
                        '(#\! #\? #\.)) 2 1)
            0))))
(== (Domain stretch) (Output Tokenize))
(== (Domain width) (Output Tokenize))
(DESIGN token-sequence (arc)
  (Sequence-In-Between (Node-To-Input-Map (Source-Node arc))
                       (Node-To-Input-Map (Destination-Node arc))))
```

Figure 2-23: Program Description of Justify (Part 2).

Chapter 3

The Design Process

In this chapter we describe the design process the DA supports. In Section 3.1 we describe the design process as seen by a user of the DA, explaining and motivating the style of interaction between the user and the DA. In Section 3.2 we describe a framework which can support the capabilities of the DA. In Section 3.3 we describe our current idea of how the DA can automate detailed design. In Section 3.4 we discuss the high-level issues we face in recording dependencies among design decisions.

3.1 What the Programmer Does

From the programmer's perspective, the DA supports programming in two broad aspects. First, it offers a process model for describing programs that is familiar, natural, and useful. Second, this process underlies a number of capabilities the DA offers: error detection, support for design changes, and detailed design automation.

The model for interacting with the user adopted by the DA is inspired by our observation of the interaction between human programmers. This exchange typically takes place using some shared vocabulary that makes the process efficient. In communicating a complex program to another programmer, an expert programmer typically describes the program using some programming concepts that are mutually shared and understood. The listener may ask the speaker questions to clarify doubts.

The DA mimics the communication process between programmers by acting as an assistant to the programmer. Using some intermediate level vocabulary, the programmer will describe to the DA what is needed. The shared vocabulary is codified in a body of clichés, kept in the DA's cliché library. The program description given by the programmer specifies some input-output behavior and may include the high-level design of a program intended to meet a specification. The DA will complete the detailed design of the given program. It may ask the programmer questions to clarify doubts, manifested as inconsistencies and incompleteness in the program description. They collaborate in the programming task in a co-operative manner.

The descriptions provided by the programmer need not be strict refinements of

program descriptions already said. Program descriptions may change during the programming process. Later program descriptions can refine, modify, supersede, and redefine earlier ones. Earlier descriptions may also be retracted or re-stated.

This interaction between with the programmer and the DA is characterized by three features.

Use of Clichés as a Description Tool: Communication between programmers seldom takes place at the level of first principles. Rather, effective communication relies on a large body of shared experience or knowledge. We believe that much of the shared knowledge between programmers can be codified and used by the DA acting as an assistant to a programmer. In this work we use clichés to describe the intent and the design of the program in a concise and comprehensible manner.

Clichés also serve as convenient contexts for bringing related concepts into the description of programs. For example, if the specification involves the *Graph* cliché, it conjures up a context in which many other ideas are meaningful because they are closely related to the graph cliché: concepts like the node set of a graph, the arc set of a graph, properties of a graph such as its cyclicity, its directedness, and its rootedness, and operations on graphs such as finding a shortest path through a graph, finding a minimum spanning tree of a graph, and searching a graph.

Using clichés as a tool for describing programs promotes the clarity of the specification. The succinctness of the description eases the task of validation: it helps to make *specifications self-evidently correct.*

Programming by Successive Elaboration: A feature of using clichés to describe programs in the DA is the incremental breadth-first exposition of the program description. A programmer starts with a few clichés that when combined gives the needed program. With these main clichés, the programmer works outward, indicating each use of the clichés in more detail. This layering of description helps the programmer focus on the sub-parts of a program after the main skeleton of the program has been sketched out. The intentional controlling of details in this process, being important for understanding a complex artifact, contributes to the ease of understanding the program. We term this process *programming by successive elaboration.*

Focus on Design Decisions: As a programming tool, the DA supports a fundamental shift in the attitude of the programmer. It interacts with the programmer in terms of design decisions. The decisions that lead to an implementation of a program are seldom made independently. The DA helps focus the programming process on the decision structure of the program being constructed by maintaining the dependencies of the design decisions made. The focus of the programming process is on design decisions: adding new ones, and retracting old ones that lead to contradictions.

3.2 What the DA Does

The key components in the architecture of the DA are shown in Figure 3-1. An

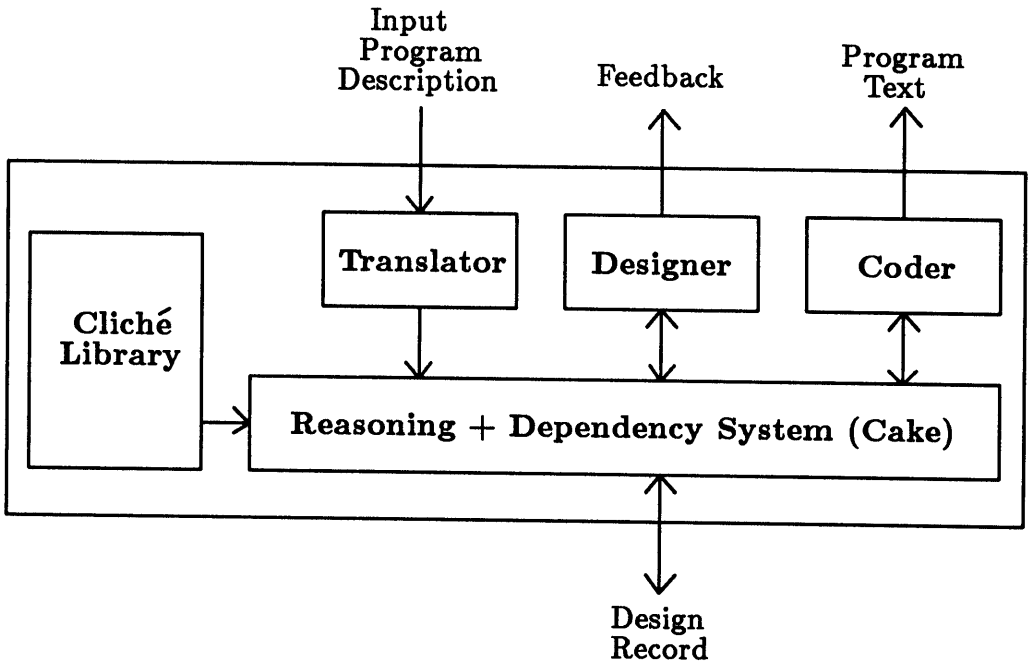


Figure 3-1: The Architecture of the DA.

important part of the DA is a library of clichés. This contains the major part of the knowledge needed to automate the design process. The translator module in the DA translates the input program description into the representation used during the detailed design process. The main module of the DA, the *designer*, selects design steps that transform the representations of designs. These changes in the designs are kept in a design record. The design record may be used by other software tools to provide other kinds of assistance. For example, a design printer may summarize the key design steps in a design record to provide design documentation. The DA requires an automated reasoning system with explicit dependencies to provide a deductive framework for reasoning and maintaining design dependencies. The final design is rendered as program text by the coder.

The representation of program designs and clichés in the library, and the specific reasoning system used are discussed in the next chapter. The discussion in the rest of this chapter focuses on the techniques that motivate the architecture of the DA and is independent of the specific representations used.

Cliché Library for Reuse: To support the selection of algorithms in the DA, we need to design related algorithmic clichés in tandem and organize them in a way that can facilitate this selection. The features of each cliché in the family should be made explicit so that the distinctions between members can be used as a basis for selection.

To support implementations of data abstractions, a library of commonly-used data

operations based on some modeling types such as set and sequence, and the various ways to implement them can be constructed. These can be organized according to several dimensions. For example, they can be organized according to the applicability conditions of an implementation, e.g.,: *this implementation is for finite sets.* They can also be ranked by the efficiency they effect.

Given a data abstraction, which is a set of operations for a given abstract type, we can find an *implementation cover* for the set from the library. An implementation cover is a set of concrete implementations (based on one or more concrete types) for the set of operations in the data abstraction. In this work, for simplicity, we consider only those implementation covers that involve one concrete type. Hence, finding a concrete implementation cover for a set of operations on an abstract type is the same as finding a data structure for the abstract type. For example, consider a set of operations consisting of `set-add` and `set-member`. An implementation cover for these two operations is the set containing the executable operations `Lisp:Cons` and `Lisp:Member`. A good implementation cover for a given *use* of a data abstraction is one that optimizes its constituent operations according to some criteria (usually run-time efficiency). This requires some frequency of use information to be available for the selection to be effective. To help provide these, we can annotate the algorithmic clichés with the frequency of each abstract operation used whenever appropriate. For example, the Build-Graph algorithm used in the scenario can have annotations to the effect that the number of times the `EndCons` operation on the intermediate node set is called in the algorithm is at most linear in the size of the input sequence.

Deductive Framework: A deductive framework for manipulating constraints in the design process is important in several respects: First, we need the expressiveness of the underlying logical language to facilitate encoding different kinds of constraints that different clichés motivate. A logical framework is extendable. Second, it provides a medium for propagating the encoded constraints which allows for error checking. Third, sophisticated reasoning procedures can be built on top of such a framework to perform deductions.

General first-order deduction is semi-decidable and propositional deduction takes exponential time. Nevertheless, we envision that simple deductions will serve as the *glue* towards bridging the gaps between the logical conclusions that are needed for achieving design and the knowledge that resides in the cliché library.

We expect clichés to have local information which can help constrain the selection process. Constraints from different clichés can interact to mutually constrain the selection of data structures and algorithms.

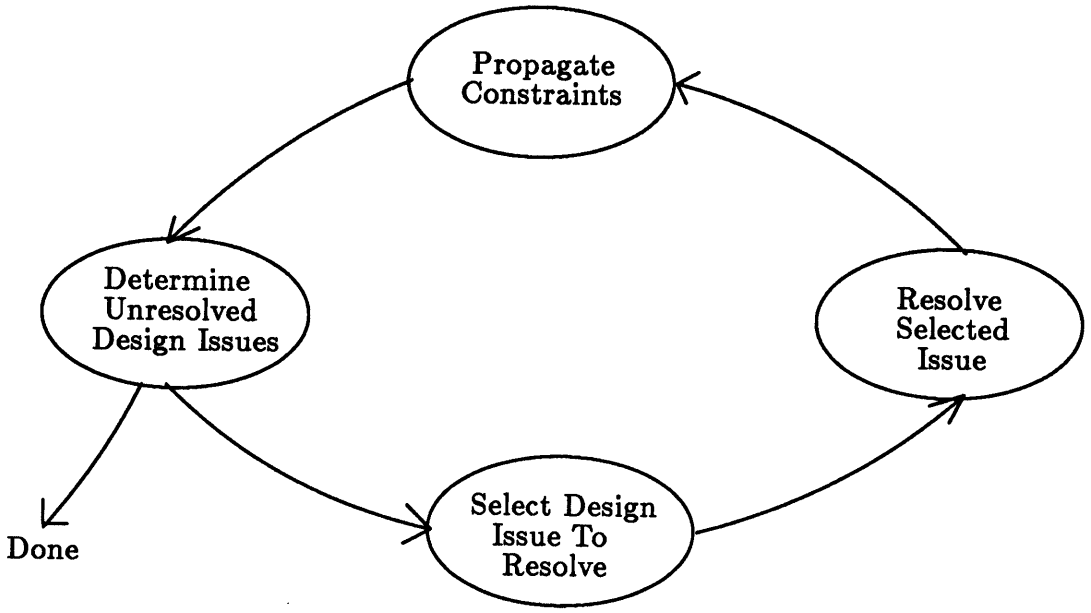


Figure 3-2: The Design Cycle in the DA.

3.3 A Framework for Automating Detailed Design

In detailed design, we envision specific design steps modifying the current design of a program to yield a new design. The output code can be extracted from the final design that results from the design process. Given the design of a program, there may be many different well-motivated design steps we can take. We therefore need some design heuristics to help us choose among the alternatives. Some design steps make implementation selections. We also need selection heuristics to guide the chosen design steps when making specific choices. Error checking and maintenance of design dependencies are also performed during this detailed design process.

Program designs, starting with the initial input program description, are represented explicitly so that *design steps* can manipulate them directly. The DA carries out a design cycle during which a design step out of its repertoire of design steps is chosen and applied as shown in Figure 3-2.

The initial design provides the starting design constraints which are propagated throughout the design by the DA. The next phase in the design cycle determines issues which must be addressed before the design is done. If there are no issues pending, then the design is complete. If there are pending issues, the DA selects one of them to resolve by the use of some design step. This design step is then applied to the current design to yield a new design. The new design may add or remove some constraints in

the design which must again be propagated throughout the design. The new design may also add new issues to be considered.

A design is complete only when all abstract operations specified directly or indirectly by the user have been implemented by concrete executable operations. In the DA, the concrete operations are Lisp primitive functions and generic library functions. Any un-implementable operation is a design issue. Most of the design steps discussed in this section address design issues by choosing implementations for abstract operations.

When a dead end is reached in the current state of a design where no feasible implementations are possible for some abstract operation, the DA resolves such conflicts by retracting some previous design decision, and selecting an alternative choice. This happens when some choice is found to be incompatible with other design choices in the constraint propagation phase of the design cycle.

The design cycle terminates when there are no pending design issues to address. This simplified notion of termination suffices for the purpose of the discussion in this chapter. A more precise notion of termination is given in the next chapter. The constraint propagation step indicated in the design cycle is difficult to discuss without a specific reasoning system. This is postponed until the next chapter where it is discussed in the context of the reasoning system used.

Based on the representations described in the next chapter, we have an implemented procedure to determine unresolved design issues. Part of the constraint propagation procedures have also been implemented. The different kinds of design steps needed to resolve design issues have been explored in detail but they have not yet been implemented. How best to select a design issue to address next in the design cycle remains to be worked out.

The following are the major kinds of design steps that arise in the detailed design of the program in the scenario presented in Chapter 2. They are described below in the abstract as acting on some representation of a program design. We will return to these design steps in the next chapter to show how they manipulate a specific representation of a program design. For each of the design steps, we first describe why they are needed, what they are and when they are applicable. For the relevant design steps, we also discuss some selection heuristics that are associated with the step.

Select View: There can be many different ways an algorithmic cliché (in the abstract) can be invoked as an operator. The order of expected input arguments and the number of arguments may differ from one use to another. What matters more is the semantic associations the input and output arguments have with respect to the cliché. We call this association a *view* of the cliché. We employ a notion of a *typical call* of a cliché, denoting the typical way a cliché may be used as an operator in the program description. We associate an ordered list of the most frequent ways a cliché may be invoked with the cliché. For example, the most typical call of a prorate cliché expects arguments in the following order: (a) amount to be prorated; (b) the

sequence of shareholders among which to prorate the given amount; and optionally, (c) the total shares, if available.

With this information, the select view design step makes a guess of what the correspondences might be. This step is useful for relating the input and output arguments of the particular use of the cliché with constraints that may be associated with the cliché. These constraints may be type constraints on the arguments or other logical constraints. When there is a contradiction, the DA may choose to backtrack and try another guess.

This step is applied to a cliché instance in the program description when it does not have an associated view. This happens when the program description is first given or when previous views are retracted due to contradictions in the design.

Select Algorithmic Cliché: The program description provided by the user may indicate the use of specific algorithms and operations. Detailed design involves choosing the most desirable algorithm with respect to the given criteria. For example, the user may specify the use of the single-source shortest path algorithm, but there are several algorithms that compute the shortest path in a graph. If the DA can deduce from the program description that the input graph is a directed acyclic graph (DAG), then a very efficient algorithm for DAGS can be used.

Select Data Structure: The DA knows about a number of abstract data types such as sets and sequences. For those clichés which are standard operations on an abstract type codified in the library, they can also be implemented by the previous select algorithmic cliché step. However, it is better to consider a whole cluster of operations in parallel: Frequently, we can cluster many of the operations whose inputs and outputs are closely related by analyzing the data flow constraints given in the program description. For example, the output of a **set-union** operation may go to a **set-member** operation. There are four sets involved here: the two input sets and output set of **set-union** and the input set of **set-member**. The implementations of all these sets can frequently be made identical. If the operations acting on the same abstract type can be clustered together to form a data abstraction, then we can decide on the implementations of the whole cluster jointly by finding an implementation cover for the operations. This is a rather important heuristic because we are unlikely to be able to choose the right data structures unless we know all the operations needed, and the frequencies of use of each of them. Frequency information must either come from the programmer or be implicit in the higher-level algorithmic clichés used.

The clustering of operations can proceed with different granularity. We could choose to implement all abstract sets in a design the same way. This, however, may be too constraining and may force independent uses of sets to be co-implemented at the expense of efficiency. Alternatively, if there is a basis for distinguishing between the sets, we can partition them into smaller independent subsets. This is frequently possible. For example, in a graph algorithm, we frequently do not require the node set and the arc set of the graph to be implemented the same way.

The select data structure design step is given a set of operations that act on an

abstract data type, and chooses a data structure that implements the operations efficiently from the cliché library.

To find an optimal solution in general is difficult. It is frequently not possible because frequency information about the use of the operations is lacking. The DA relies on heuristics characterizing when a data structure is better than another. For example, a heuristic in the implementation of the abstract sequence type states that (a) if the EndCons operation is needed, the following data structures are good: Vector, Chain-Shelf, List-Shelf, Doubly-Linked-Chain, and Doubly-Linked-List. (A list-shelf is just like a chain-shelf except that the underlying list is a linked list instead of a chain. A doubly-linked-chain is a chain that is linked both forward and backward.) We can order the suggestions using whatever distinguishing features they have. For example, doubly-linked structures are put last because they consume more space.

Consider the set of operations on the token sequence in the scenario, they are: Sequence-In-Between, EndCons and Insert. The following two selection heuristics are applicable in addition to those above: (b) if the Insert operation is needed, the following data structures are good: Linked-List, Chain, Chain-Shelf, List-Shelf, Doubly-Linked-Chain, and Doubly-Linked-List. (c) If the Sequence-In-Between operation is needed, the following data structures are good: Chain, Chain-Shelf, and Doubly-Linked-Chain. The intersection of the suggestions of the three heuristics gives us two choices: chain-shelf or doubly-linked-chain. A further heuristic suggests that a chain-shelf is in general more space-efficient than a doubly-linked-chain. In the data abstraction involving the Arc type and the Node type where the only operation invoked on both types is EndCons, a vector is not chosen because there is a precondition for using a vector to implement a sequence: the length of the sequence must be known a priori. Both the sizes of the node set and the arc set are unknown when the vector has to be created.

Implement Mapping: Many algorithmic clichés contain abstract mappings without constraining their implementations. Some of these mappings can typically be viewed as computing intermediate results. For example, in the single-source shortest path algorithm, there are two such mappings: Min-Distance of a node holds the current minimum distance from the start node to the given node, and the Best-Previous-Arc of a node keeps the arc on the current optimal path that ends with the current node. Some standard ways of implementing mappings in Lisp include: (a) as a vector; (b) as an association list; (c) as a hash table; and (d) distributed among the domain elements. For example, if the domain elements are structures, mappings can be kept in fields of the structures. Which representation is better depends on the use and the nature of the domain of these mappings.

Pre-Compute Mapping: Some mappings are most appropriately pre-computed before any access, while others are best computed when needed. For example, a mapping whose domain elements are ordered and whose definition depends on this implicit order may be better pre-computed since we can frequently compute all elements in one iteration through the entire domain. In the scenario, the domain of `width` is the

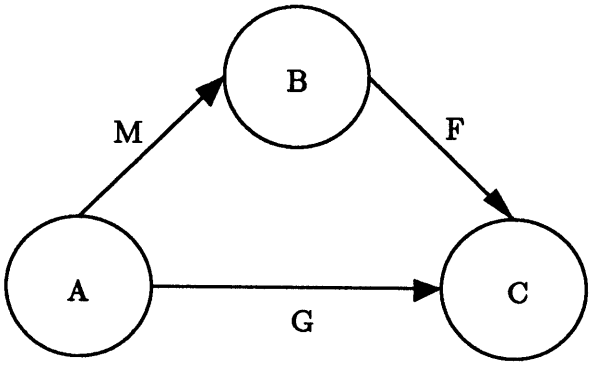


Figure 3-3: An Opportunity for Caching: Two Related Functions.

output sequence of `tokenize`, and `width` depends on both the given token and the token prior to the given one. Single access of the mapping can take linear time to search for the correct previous token (unless the previous element of every token is stored).

The *pre-compute mapping* design step generates a plan to compute and explicitly store the given mapping before every use of the mapping in the current design. It prefers to place the pre-computation close to the creation of the domain elements of the mapping. It also implements all uses of the mapping in the design by the retrieval function corresponding to the chosen explicit storage method. For example, if the domain elements are Lisp structures, a field may be used to store the value of the function, and the accessor function of the field is used. A design constraint is made to implement future accesses to this mapping by the retrieval.

Cache Mapping: If a mapping is not to be pre-computed and is used in several places, it may be efficient to memoize the computed results for future uses rather than recomputing each time it is needed. An important subsidiary issue is finding an appropriate place to store the memoized result efficiently.

The *cache mapping* design step is intended to be invoked on two mappings, one of which is defined in terms of the other. The design step is useful if it is desirable to memoize the values of one mapping so that the other mapping can be more cheaply computed. To make this more concrete, consider the design step acting on the example from the scenario: we want to cache the mapping `ratio` for a second mapping `arc-ratio`. The domain of `ratio` is a set of sequences of tokens. The general method for storing a mapping whose domain elements are sequences is to use a hash table. However, if the current design represented the sequences as a series, hashing will not

work. In our above example, we can store the mapping on the domain elements of `arc-ratio`. An arc is implemented as a Lisp structure. This design step adds a new field `arc-ratio` and stores the computed `ratio` values directly in the field when `ratio` is being computed.

More generally, consider the two mappings illustrated in Figure 3-3, `F: B` $\rightarrow$ `C`, and `G: A` $\rightarrow$ `C`, related as follows: `(G a) = (F (M a))`. Suppose we intend to cache computed values of `F` so that `G` can look up the cached values. An immediate question is: where can we store cached values of `F`? We could implement `F` as an explicitly stored mapping so that after `F` has been computed, future accesses of `F` can be done by simple retrievals. However, if `F` cannot be represented as an explicitly stored mapping directly for some reason, then we can consider storing computed values of `F` on some data structure that makes it easy for `G` to access them. To be more specific, consider the case when `A` is a Lisp structure where we can store the cached values in a field on `A` so future invocations of `G` can be performed by simple retrievals. What are the conditions sufficient for carrying out this optimization step? It is easiest to work backwards as follows:

The intention of the optimization is so that we get the value of `(G a)` for some `a` in `A` by performing a look-up operation on `a`. That is, for all `a` in `A`, we must find an `s` such that (1) `s = (M a)`; (2) `(F s)` must be computed before `(G a)` is needed; (3) when `(F s)` is being computed, the corresponding `a` is accessible so that we can store `(F s)` on it.

This design step also operationalizes some checking that is required to ensure that the optimization step can be carried out. This is discussed in more detail in the next chapter in the context of a specific representation for program designs.

This design step is triggered by specific design heuristics that notice that such optimization opportunities exist.

Coerce Type: This design step tries to resolve a mismatch in the expected types of some data flow that passes between two operations in the design. The step is triggered by the DA when the output end of the data flow has an incompatible type from the input end of the arc. It looks for a way to coerce the output data into an instance of the expected input type. This is typically done by finding a unary function that can coerce the output type into a type acceptable to the receiving operator. If no coercions can be found, the step complains to the programmer about the type constraint violation.

This step can also be considered as a selection of an appropriate view for a given data object. Data objects may be viewed in a number of ways. For example, we can view a path as a sequence of arcs or a sequence of nodes, and an ASCII file as a sequence of characters or a stream. In the scenario, the output ASCII file given is viewed as an output stream by opening the file at the start of the computation, and closing it at the end. Part of the Ascii-File cliché is some knowledge about how we can view an ASCII file as an output stream or an input stream. This design step, with the help of specific knowledge in the Ascii-File cliché, adds two new operations

to open and close the output file. The exact placement of these operations and the attendant data flow can be determined from the first use and the last use of the ASCII file in the program description.

Examples of other uses of coerce type design steps in the scenario are: adding the Content operation to coerce tokens into strings before they are fed to the Last-Character operation and the Number-Of-Characters operation in the width design.

Omit Operation: This design step is triggered whenever an operation is being implemented. It checks to make sure that the postconditions guaranteed by this operation are not already satisfied by the current design. If they are already satisfied, then the operation can be omitted. The scenario contains an example where the topological sorting step in the single-source shortest algorithm chosen was omitted because the input graph it was given was already topologically sorted by the chosen build-graph algorithm preceding it.

Exchange Operation for Input: This design step removes an operation in a design, adding an input to the design in place of the removed operation, thereby changing the input-output specification that the design implements. For example, if this step is applied to the design of the `width` function to remove the `Preceding-Item` operation, then the new `width` function will now have two input arguments instead of one. This design step is used primarily as a setup step for other design steps that make use of the new designs created.

Make Assumption: Design is an under-constrained problem. A strategy many programmers do in design is to forge ahead by making assumptions and complete the design as far as possible. Whenever a dead end in the design process is reached, we retract some of the assumptions that have led to the dead end. This strategy is especially effective in the presence of incomplete knowledge and incomplete specifications.

The DA posts new constraints in a design as assumptions in order to advance the design. Such assumptions may stay on to become part of the design or may later be retracted if they are found to be inappropriate. There are several kinds of assumptions the DA makes automatically. One kind is to assume that an operator has certain types based on some particular applications involving the operator. For example, if the DA deduces that `width` is applied to a token and that it returns an integer, then it may assume that the signature of `width` is `token` $\rightarrow$ `integer`. This may be wrong because `width` may also be applied to non-tokens or it may return non-integers. This closed-world assumption may be retracted later if contradictions arise from it.

Another kind of default assumption the DA makes is in the user-specified functions. The DA assumes that these functions are subroutine boundaries, we call these *single subroutine assumptions.* However, it may be well-motivated to implement different uses of the same user-defined function differently. Single subroutine assumptions help to reduce the number of design decisions the DA has to consider. This assumption can be violated if the programmer or other design heuristics indicate so.

User Says: During detailed design, the user may specify some specific action

to be taken. The *user-says* design step is used to explicitly represent this action in the design record. This explicit recording also helps support retraction of user-given descriptions. The input program descriptions given by the programmer are considered to be user-says design steps. When the programmer retracts some earlier program description, the user-says design step that added the program description to the design is retracted.

3.3.1 An Example of Automatic Detailed Design

The design step types described in the last subsection are motivated by an analysis of the program in the scenario. The following is a list of the major design steps required to transform the input program description in the last scene of the scenario into the final output program.

- Select-View: A select-view step views the input argument of the Tokenize specification as the input file on which the tokenization is run and its output as the output sequence of tokens from tokenization. Another views the input argument of the single-source shortest path specification as the input graph and the output as the output shortest path. Another select-view step views the first input argument of the prorate specification as the amount to be prorated, its second argument as the sequence of items over which the proration is done, and its output as the sequence of prorated amounts.

- Select-Algorithm on Tokenize: This step chooses the *NFA-tokenizer* algorithm to implement the Tokenize specification.

- Select-Algorithm on Build-Graph: This step chooses the *build-rooted-forward-dag-with-weak-left-arc-test* algorithm to implement the Build-Graph specification.

- Select-Algorithm on Single-Source-Shortest-Path: This step chooses the *dag-sssp* algorithm to implement the Single-Source-Shortest-Path specification.

- Omit-Operation on Topological-Sort operation: The topological sorting operation of the *dag-sssp* algorithm is omitted because the graph it is given is already topologically-sorted. This follows from the build graph algorithm chosen.

- Select-Algorithm on Output: There are four uses of the output specification in the paragraph justification program. This design step is applied four times, once on each use. For the output specification in the justify design, a select-algorithm step chooses the *write-char* function. In the lineout design, the *write-n-chars* function is chosen for the first use, the *write-string* function for the second use, and the *write-char* function for the third use.

- Select-Algorithm on Prorate: This step chooses the *prorate-favor-end* algorithm to implement the prorate specification.
- Select-Data-Structure on the sequence of tokens from the output of Tokenize: This step represents the sequence as a series.
- Select-Data-Structure on the sequence of tokens from the output of Segment: This step represents the sequence as a chain-shelf.
- Select-Data-Structure on the sequence of nodes in the build graph algorithm: This step represents the node sequence as a chain-shelf.
- Select-Data-Structure on the sequence of arcs in the build graph algorithm: This step represents the arc sequence as a chain-shelf.
- Select-Data-Structure on the path in the shortest path algorithm: This step represents the path as a Lisp list.
- Select-Data-Structure on graph in the build graph algorithm: This step represents the graph as a sequence of arcs kept in a field in a Lisp structure.
- Select-Data-Structure: A select-data-structure step represents tokens as Lisp structures, **token-struct**. Similar steps when applied to nodes and arcs also chooses to represent them as Lisp structures, as **node-struct** and **arc-struct** respectively.
- Coerce-Type: A coerce-type step adds new operations to the justify design to open the input file at the start of the justify operation and close the output stream at the end. Another coerce-type step adds a content operation to coerce the input token of the Last-Character function into a string. A similar step is applied to the input of the Number-Of-Characters function.
- Implement-Mapping: A number of mappings are implemented by the implement mapping design steps. The following mappings on tokens are implemented as fields in the Lisp structure representing the token type: content, pattern, successor-token, width and stretch. The fields are respectively named token-content, token-type, token-next, token-width and token-stretch. Similarly, the following mappings on nodes are implemented as fields in the Lisp structure representing the node type: node-to-token-map, successor-node, min-distance, and best-previous-arc. The fields are respectively named: node-token, node-next, node-min-distance, and node-best-previous-arc. Similar implement mapping design steps implement the mappings on arcs and graphs as fields on the Lisp structures representing their respective types.

- Exchange-Operation-For-Input: One of these steps is applied to the Preceding-Item specification in the width design. This creates a new function that is like width but takes in two arguments instead of one. This step is to prepare for the efficient pre-computation of the width mapping.

- Precompute-Mapping: A precompute-mapping step moves the computation of the width function to the beginning of the justify design, right after the tokenize specification. A similar step pre-computes the successor-token mapping and the stretch mapping.

- Cache-Mapping: A cache-mapping design step caches the values of the ratio computation in the build-graph algorithm to be used for the arc-ratio mapping later in the shortest path algorithm.

- Make-Assumption: The single occurrence of the `lineout` operation in the `justify` design suggests that its first input is an output stream and its second input is an arc. (This information is obtained from constraints propagated in the design after the select-view design step has run.) The output is not constrained by this use. A make-assumption design step assumes that the `lineout` function takes in an output stream and an arc and returns some data. Single subroutine assumptions are made for all user-supplied functions.

3.3.2 Further Challenges

There are a number of challenges that must be met before the DA program can automate detailed design effectively:

Need for Program Metrics: The currently contemplated selection heuristics for helping the DA choose implementations are rather weak because they are too local. The clustering of operations is a step in the direction of considering the selections globally. However, to make effective tradeoff decisions in choosing an implementation cover for a data abstraction, we need a formal language for characterizing the efficiency of programs and program skeletons. Each cliché in the cliché library can be annotated with various program metrics, and methods are needed to estimate new metric values when clichés are combined. Automatic means of classifying the various program metrics, when available, can aid in the codification of programming clichés since they can reduce the analysis effort of the knowledge engineer.

Search Control: Our framework can be viewed as establishing a search space in which an executable program can be derived from some program description. This work has not adequately explored the control issue in this search problem. We have indicated in the earlier section one technique that can help control the search process: data operation clustering. Much more, however, remains to be explored. The existence of a good program metric may serve as a good guide in exploring the various alternatives effectively.

3.4 Recording Design Dependencies

There are several reasons a design decision may have to be modified. First, the user may want to change it. We have already discussed the attendant benefits that accrue from this service. Second, when a dead end is reached in the automatic detailed design process, some design decision may have to be retracted. Third, the DA may want to make some design decision only if some conditions remain true, and may want to reconsider the decision when the conditions are no longer true. It is possible to support all these operations by re-deriving the design from the beginning. Another way is to chronologically backtrack to the decision that needs to be modified. A better way, however, is to retract only those decisions that matter. There are three distinct advantages to dependency-directed backtracking. It is more efficient, the explicit dependencies can be used to provide some explanations, and a more sophisticated reasoner may use the explicit reasoning record for introspection and reasoning. In our context we are primarily concerned with supporting backtracking efficiently and generating explanations from the dependencies.

When a program design is viewed as a set of logical constraints, each design step is seen to be adding new constraints to a design and/or deleting old constraints from a design. One kind of design step only adds new constraints to the design, we term them *monotonic* design steps. The second kind, the *non-monotonic* design steps delete some constraints from the old design, and they may add new constraints to the design.

Out of the ten design steps described in the last section, five of them are monotonic design steps. They include the steps for selecting views, algorithms, data structures, and mapping implementations and the make-assumption step. The rest are all potentially non-monotonic steps.

The maintenance of design dependencies in the DA requires the following functions:

- Explicit Recording: A design step may add and delete many different constraints at one time. It is convenient to have a single decision node associated with the design step so that if we want to retract this step, we can simply retract the associated node.

- Automatic Retraction: If a design step depends on some conditions which support its application, then the design step should be automatically retracted when any of the supporting conditions are no longer true. By retracting a design decision, we mean retracting the new constraints added to the current design and adding back those constraints that were removed by the design step. This may trigger other design decisions to be retracted too if the latter depend on the former.

It is useful to note here that whether we want to re-apply the design step automatically when the supporting conditions later become true again is a separate issue.

Given that the DA already has a mechanism for automatic detailed design, i.e., the design cycle, this issue is not considered here. Retraction of some design decisions may make the design incomplete, thus triggering the design cycle to try to complete the design.

Recording Requirements: To support automatic retraction, an explicit *decision record* can be kept with each design step taken, and the following kinds of information should be kept for each decision record: the new constraints added by the step, the old constraints that were removed by it, and the conditions supporting the step. The latter is to help support the automatic retraction of the design step based on some changing conditions.

The main difficulty in maintaining design dependencies in the DA lies in recording the conditions supporting the application of design steps in the decision record. For each design step, we want to record *only* those conditions that should trigger the retraction of the step. It is, however, not always clear how this can be done efficiently. In the worst case, we can always make a non-monotonic design step depend on every design step that took place before it, thus degenerating into strict chronological dependency. For non-monotonic design steps, should we make the prior existence of a constraint that is to be removed by this design step one of its supporting conditions? This can be done in two ways. First, we can copy the entire old design to a new design with the exception of those constraints to be removed by the design step. The immense cost involved seems unacceptable. Second, we can make the current design step depend on some other design decision, called a *sponsor*, that led to the prior existence of the constraint (to be removed). There can be multiple sponsors for a given design constraint. In this way, if the sponsoring design decision is retracted, the non-monotonic design step can also be retracted automatically. The exact implication of this scheme, however, has not yet been worked out.

The recording of design dependencies is carried out by individual design steps. How this is carried out is best explained in terms of some specific representations for program designs. This is covered in the next chapter when the design steps are described.

Chapter 4

Representing and Manipulating Design Artifacts

In this chapter we discuss how the various design artifacts needed by the DA can be represented. Section 4.1 describes the Plan Calculus which is used to represent algorithmic clichés and program designs. It shows how the various kinds of knowledge present in the cliché library are represented. Section 4.2 describes CAKE, a knowledge representation and reasoning system that contains an implementation of the Plan Calculus. Section 4.3 describes the Common Lisp macro package SERIES used to codify loop computation clichés. Section 4.4 outlines the cliché library as it exists now, and the principles used to organize the clichés. Section 4.5 describes how program designs and design dependencies are represented. Section 4.6 illustrates how the various design steps described in the last chapter manipulate design representations and how these design steps record design dependencies in the evolving design record. It also describes how some design constraints are propagated in CAKE. In the context of using plans to represent program designs, Section 4.6 also discusses the various notions of when a design is considered complete.

4.1 The Plan Calculus

The Plan Calculus [25] is used to represent program designs and algorithmic clichés in the cliché library. It models programs and designs as *plans*. A plan consists of a structural part and a logical part. The structural part of a plan captures the data flow and control flow between the computation steps that make up the plan, and is illustrated via a *plan diagram*, see Figure 4-1. The logical part consists of constraints on the parts of the plan and is typically left out of plan diagrams.

The plan calculus formalism has several desirable properties essential to a representation for algorithmic clichés. First, by representing data flow and control flow directly and explicitly, it abstracts away from the syntactic variations in specifying clichés. This provides us with a unique representation for each cliché. Second, we

can codify knowledge that is basic to programming, independent of the target programming language. For example, the cliché to view an abstract list as a set is not language-dependent, neither is the single-source shortest path algorithm language-dependent. The plan calculus can also codify knowledge that is language-specific too. For example, we can specify how the Lisp linked list type can be used to implement an abstract list. In the cliché library, we expect to have both kinds of knowledge. We view programming knowledge that is language-independent to be in the *modeling level.* Moving from one programming language to another does not involve changing knowledge in the modeling world, it only involves adding language-specific clichés codifying the various ways the new language supports the basic abstract types and operations in the modeling level. Third, the plan calculus is very expressive. It can be used to codify implementation relationships, data abstractions and optimizations.

Figure 4-1 illustrates several features of the Plan Calculus. The left of the figure is a plan; it is the plan for the `ratio` design given in the scenario. The input description for `ratio` (in the last scene) is shown in Figure 4-2. Plain solid arcs denote data flow and hatched arcs represent control flow. The topmost arc is the input to the plan, and the bottommost arc is the output of the plan. Boxes, or *roles*, in a plan denote computation steps. There are three kinds of boxes:

(1) Input-output specifications such as the `extras1` box, the `total-stretch1` box and the `div1` can be viewed as function calls. These boxes have input ports and output ports which are typed. Logical constraints can be associated with the inputs and outputs in the form of preconditions and postconditions. For example, the precondition of `div1` states that the divisor must not be zero. Each input-output specification also has an entry point, called an `in` situation and an exit point called an `out` situation. These are useful for modeling constraints on the order of execution of the boxes. Input-output specifications may have annotations to indicate that they have Lisp implementations. For example, the `div1` box has an annotation to the effect that its Lisp implementation is `/`. We call such input-output specifications *implementable.*

(2) Test specifications such as `zerop1` and `zerop2` correspond to predicate applications. They are distinguished by two exit points labeled `S` (succeed situation) and `F` (fail situation) corresponding to the result of the predicate.

(3) `J1` and `J2` are `Join` boxes which do not correspond to any computation, but they are needed for merging control flows after a test specification. Join specifications have two input ports, `succeed-input` and `fail-input`, each has a corresponding entry point, a succeed situation (labeled `S`), and a fail situation (labeled `F`) respectively. It has a single output port labeled `output` and a single exit point labeled `out`.

Figure 4-1 also illustrates an example of an *overlay.* The hooked lines linking the two plans specify the correspondences between the two plans, indicating how the left plan can be used to implement the `ratio` box on the right. Unlabeled correspondences indicate equalities.

A key part of the cliché library is knowledge about data abstractions and their

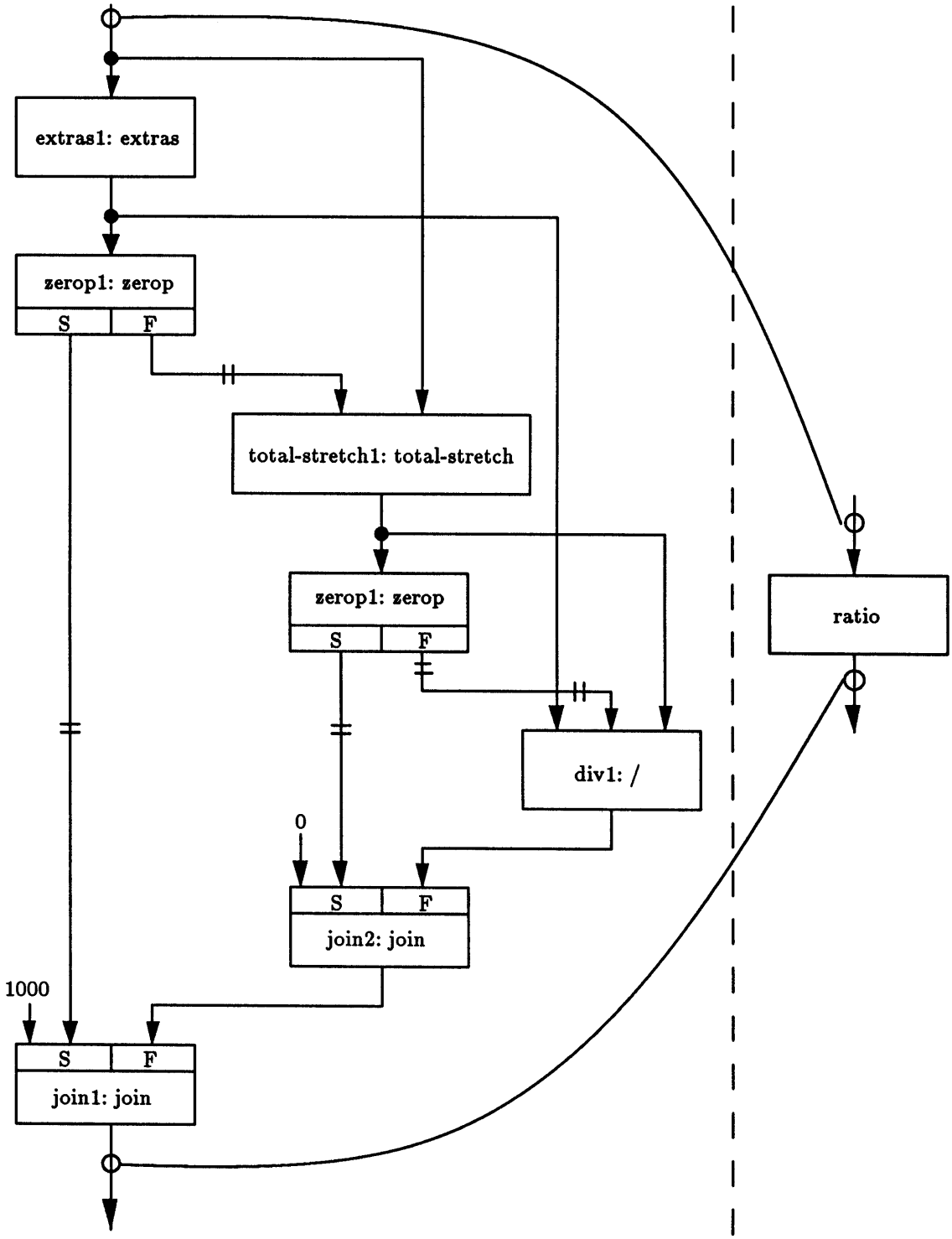


Figure 4-1: An Example of a Plan: The Ratio plan.

```
(DESIGN ratio (seq)
  (LET ((extras (extras seq)))
    (IF (= extras 0) 0
        (LET ((total-stretch (total-stretch seq)))
          (IF (= total-stretch 0) 10000
              (/ extras total-stretch))))))
```

Figure 4-2: Input Program Description for the Ratio Function.

implementations. This knowledge is codified via *data plans* and *data overlays*. Figure 4-3 shows an example of a data plan which represents a standard aggregation of data. The data plan **arc** is a pair of nodes labeled **source** and **destination**. The constructor function, (**!make arc**) and the selector functions (**!select arc source**) and (**!select arc destination**), automatically defined by the data plan, are also shown in the figure.

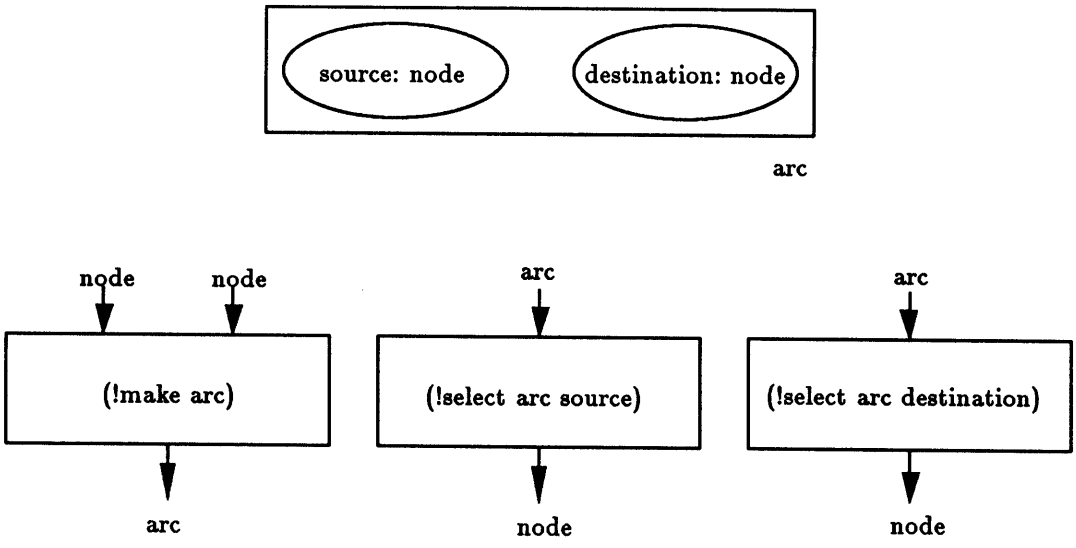


Figure 4-3: An Example of a Data Plan: Arc.

Figure 4-4 shows an overlay which is defined with the help of a data overlay. The left box is a test specification which checks to see if a given data item is in a given abstract list. The right box is a set membership test specification. The implementation overlay shows how we can view the **member-of-list?** specification as implementing a set membership test via the data overlay (**!as list set**). The latter overlay specifies how we can view a list as a set. It is a function that takes a list into a set. In the DA a list is modeled as the union type of the empty list and a data plan that contains two parts: **head** is a data item, and **tail** is a list (recursively).

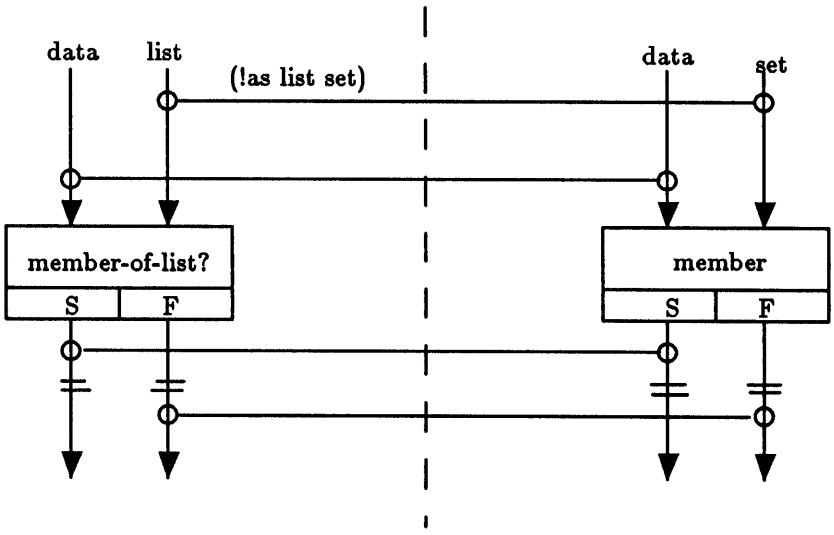


Figure 4-4: Implementation Overlay using a Data Overlay.

The (`!as list set`) overlay is defined as: if the list is empty, the result is the empty set, else the result is the union of the singleton set made up of the head of the list and the result of recursively invoking the data overlay on the tail of the list. Given that most of the definitions involving data plans and overlays are non-structural, they cannot be well-illustrated in a plan diagram.

4.2 CAKE

CAKE [9, 26] is a knowledge representation and reasoning system which implements the Plan Calculus and a number of other reasoning facilities, including a truth maintenance system (TMS). CAKE provides the infrastructure on which the needed programming knowledge is represented and reasoned about.

CAKE has a layered architecture: a TMS with equality and pattern-directed procedure invocation mechanisms forms the foundation of the system. On top of this is an implementation of a typed logic within which the following special-purpose decision procedures reside: algebraic properties of operators such as commutativity and associativity, sets, partial functions, and frames. The top layer is an implementation of the Plan Calculus discussed in the last section.

The TMS in CAKE can be used to propagate constraints and to maintain design dependencies. It also supports automatic retraction of assertions and their consequents, and contradiction detections. Constraints can be expressed in full first order predicate calculus though most propagation of constraints is carried out only at the propositional level. As complete propositional reasoning is computationally intractable,

CAKE sacrifices completeness for efficiency. It automatically performs simple one-step deductions equivalent to unit propositional resolution. As unit propositional resolution is fast but incomplete, CAKE supports quick but shallow deductions. It also supports a pattern-directed procedure invocation mechanism which can be used to implement some controlled reasoning of quantified terms.

CAKE implements the formal semantics of plans by providing a number of definitional forms that are translated into the underlying logic. Plans become structured types and overlays become functions from plan types to plan types. Individual plan instances can be made, and their semantics are implemented as logical constraints in the CAKE knowledge base.

For example, the definitions for the plans shown in Figure 4-1 in the last section are shown in Figure 4-5. `Defio` in CAKE defines an input-output specification plan type (box type). The `:inputs` keyword is used to provide a list of input ports of the input-output specification. The elements of this list are pairs, the first member of a pair is the name of the parameter, and the second is the type of the parameter. Thus, in Figure 4-5, the input-output specification named `ratio` has an input parameter named `arg1` whose type is `data`. Similarly, the `:output` keyword is used to indicate output ports of the box. `Defplan` defines a non-atomic plan type. The `:roles` keyword specifies the list of boxes making up the plan. Boxes are also named and typed. The `ratio-plan` has seven boxes. There are two instances of the `zerop` test specification type, two instances of the `join` specification type, and three input-output specifications: one named `div1` is an instance of the plan for division of numbers, the one named `extras1` is an instance of the extras plan type, and a third named `total-stretch1` is an instance of the total-stretch plan type. The prefix `!sometimes` refers to a box which need not be defined at all times because control flow may not be passed to the box. The `:dflow` keyword indicates the data flow constraints between ports of the boxes in the plan. In `ratio-plan`, there are data flow constraints from the output port named `output` of the `extras1` box to the input port named `arg1` of the `zerop1` box, and the same datum also goes to `arg1` of `div1`. Other data flow constraints are similarly specified. The keyword can also be used to indicate *tied inputs*, e.g., `arg1` of `extras1` and `arg1` of `total-stretch1` are tied in the figure. It can also be used to indicate constant inputs to some input ports, e.g., `succeed-input` of the `j1` join box is always 0 in the figure. Similarly, control flow constraints are specified using the `:cflow` keyword. The first control flow specification in the figure indicates that control flows from the succeed situation of `zerop1` to that of `j1`.

The last CAKE form shown in Figure 4-5 is `defoverlay`. The form shown defines an overlay named `(!as ratio-plan ratio)` whose domain is ratio-plan and whose range is the ratio input-output specification. The correspondences that define this overlay is indicated with the use of the `:corrs` keyword. The overlay in Figure 4-5 specifies that we can view an instance of a ratio-plan as an instance of a ratio input-output specification as follows: the `arg1` of the ratio input-output specification is the `arg1` of the `extras1` box in the ratio-plan, its output is the output of the `j1` join,

its in situation is the in situation of the **extras1** box, and its out situation is either the succeed situation or fail situation of the join box **j1**.

```
(Defio Ratio
  (:Inputs (Arg1 Data))
  (:Outputs (Output Data)))
(Defplan Ratio-Plan
  (:Roles
    (Extras1 Extras) (Total-Stretch1 (!Sometimes Total-Stretch))
    (Zerop1 Zerop) (Zerop2 (!Sometimes Zerop))
    (Join1 Join) (Join2 Join) (Div1 (!Sometimes /)))
  (:Dflow
    ((Output ?Extras1) (Arg1 ?Zerop1) (Arg1 ?Div1))
    ((Output ?Total-Stretch1) (Arg1 ?Zerop2) (Arg2 ?Div1))
    ((Arg1 ?Extras1) (Arg1 ?Total-Stretch1))
    (0 (Succeed-Input ?J1))
    ((Output ?J2) (Fail-Input ?J1))
    (10000 (Succeed-Input ?J2))
    ((Output ?Div1) (Fail-Input ?J2)))
  (:CFlow
    ((Succeed ?Zerop1) (Succeed ?J1))
    ((Fail ?Zerop1) (In ?Total-Stretch1))
    ((Succeed ?ZeroP2) (Succeed ?J2))
    ((Fail ?ZeroP2) (In ?Div1))))
(Defoverlay (!As Ratio-Plan Ratio)
  (:Corrs :Arg1 (Arg1 ?Extras1)
          :Output (Output ?j1)
          :In (In ?Extras1)
          :Out (If (Defined (Succeed ?J1)) (Succeed ?J1) (Fail ?J1))))
```

Figure 4-5: The CAKE Definition for the Ratio Plan.

4.3 SERIES

SERIES [35] is a Common Lisp macro package that supports a new data type called *series* which is similar to the Lisp sequence type. Computations on sequences of data items are easier to understand and modify when written as compositions of functions than as equivalent loops. SERIES allows programs to be written as series expressions, or compositions of series functions, without incurring any run-time overhead.

In this work, algorithmic clichés involving computations of sequences of data are expressed as series expressions as far as possible. As a result, much of the output code needed in the scenario are rendered as series computations. This has three key benefits. First, the DA does not have to manipulate and reason about loops. Second, the higher-order series functions such as **collect-fn** and **scan-fn** provide powerful

and useful abstractions for loop computations. This simplifies the need to reason about loop computations. Third, combinations of series functions are automatically optimized. This removes an important component of the detailed design: automatic merging of simple loops for efficiency. For example, in the scenario presented in Chapter 2, the pre-computations of the `width`, `stretch`, and `next` mappings in the `tokenize` routine pass a series from one computation to another without regard to the need for loop merging. Without SERIES, they would have to be merged into one loop explicitly by some design step.

4.4 The Cliché Library

Significant effort in this project has gone into identifying the various clichés that are used to describe the paragraph justification program in the scenario. Most of the major clichés have already been informally described in Chapter 2. Many of them have been codified in the Plan Calculus and compiled successfully in CAKE. In this section, we discuss a key principle in the codification of knowledge: knowledge factoring, and some techniques used in the codification of programming clichés in particular. The last subsection contains a brief description of the cliché library in the DA.

Knowledge Factoring: Knowledge factoring refers to the organization of knowledge such that commonalities among different codified facts are identified and made explicit. How well knowledge is factored is an important criterion in any knowledge codification effort.

Factoring is particularly important in the domain of programming knowledge where a good amount of programming knowledge is not only independent of the programming language used, but it is also independent of the exact data structures used to implement any abstract operations the knowledge may involve. For example, the knowledge to view a path (modeled as a list of arcs) in a graph as a list of nodes is independent of the implementations involved. If common knowledge is not appropriately factored out, we will see a combinatorial explosion in the number of closely related clichés in the library. Besides providing a more compact representation of the same knowledge, knowledge factoring is important as a technique for organizing the library so that it will be comprehensible to the programmer. It helps to break up the huge mass of knowledge into manageable and meaningful chunks to help focus the attention of the programmer. An organized library can also help the DA search for feasible implementations more efficiently.

In the following subsections we describe some specific techniques we have used in constructing the cliché library.

4.4.1 Abstract Data Types

We can always represent specific algorithms down to their concrete operations. However, it is far better to capture the algorithm abstractly so that the same algorithmic plan can be used in many different situations. In many algorithms, the concrete implementation of a mapping or a data type are immaterial to the essence of the algorithm. In such a case, the representation of the algorithm should reflect this fact. The mapping or abstract type could be left un-implemented so that it can be implemented in different ways when the need arises. This is in line with the way algorithms are presented in textbooks [2, 3, 7].

In this work, we codify algorithmic clichés in terms of more basic operations on a few commonly-used abstract data types, such as sets, lists, and mappings. The different data structures that can be used to implement the various abstract types are also specified. Each of them can be modeled as a type in the logic where their salient properties can be conveniently clustered together. For the kind of routine design being explored here, the important part of these data structures lies in the input-output specifications of the operations the data structures support. We codify the implementation relationships via overlays between the concrete operations and the corresponding abstract ones. In more sophisticated designs we may need to reason in detail about each of the operations being achieved by a concrete operation chosen. In such a case, we need to model the concrete type constructively so that its inherent properties are made explicit. In this way, we can open up the *black box* to look at how each concrete operation is being constructed out of more primitive ones. Clearly, formalization and knowledge acquisition in this rich domain is a time-consuming task. We have only looked at a few data structures that can support the two abstract data types needed in the scenario: List and Mapping.

4.4.2 Taxonomy of Related Clichés

The Plan Calculus has two primary mechanisms for abstracting commonalities among related clichés. First, structural similarities can be captured by plan specializations and extensions. For example the type `(!list arc)` is a subtype of the `list` type, and the **add-one** plan in Figure 4-6 is a specialization of the generic plan for adding two numbers. Extensions of plans are used implicitly when new computational steps are added to an existing plan during the design process. Second, functional similarities between plans are captured by implementation overlays that view them as the same input-output specification.

Input-output specifications, however, have rigid structures: the number of inputs and outputs must be fixed and they have static types. Many input-output specifications which have similar purposes (to a programmer) are not supported by the two forms of abstraction. For them, we create a separate taxonomic hierarchy that is used for aggregating different input-output specifications with similar *purposes* under one roof. It serves as a general specification for the specific kind of computation that

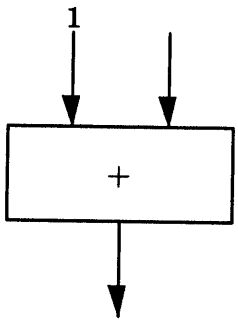


Figure 4-6: The Add-One Plan.

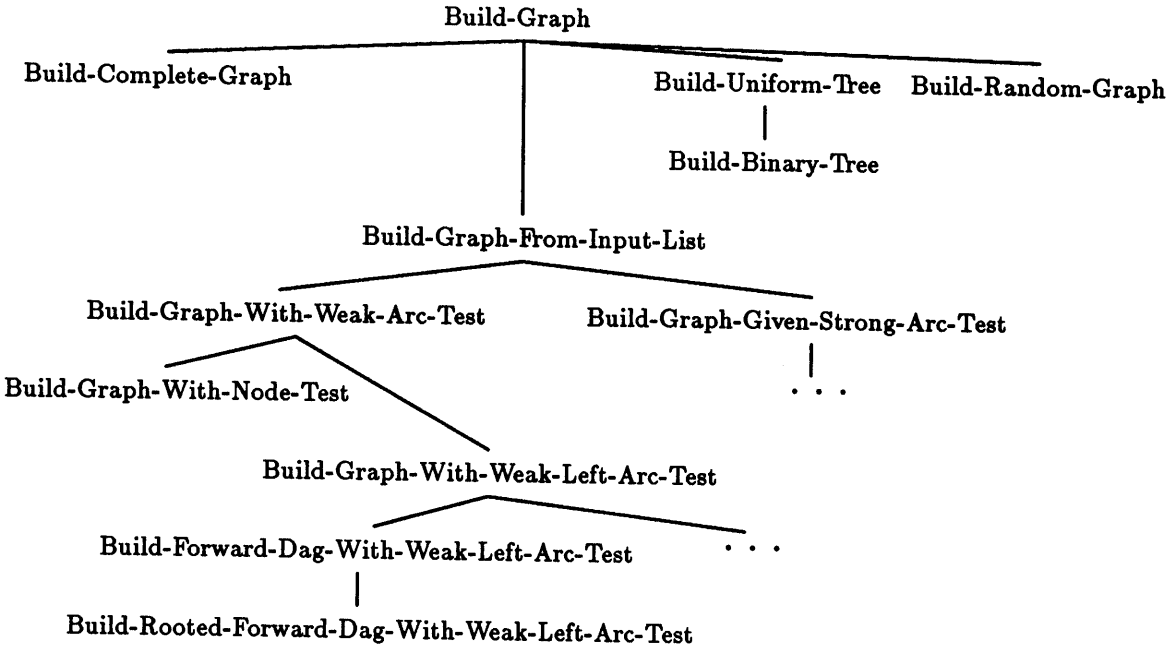


Figure 4-7: The Family of Build-Graph Clichés.

might be associated with the name of a cliché. We termed them *cliché frames* associated with the cliché, partly because CAKE frames are used to represent them. It is an aggregation of the relevant knowledge about the cliché just like a specific plan is an attempt to capture part of the knowledge involved in a cliché. This hierarchy is organized by the similarities in the purposes they serve. In Figure 4-7, the `build-graph` family of clichés is organized in one hierarchy because they are intended to serve a common purpose: they compute graphs out of some intentional descriptions.

Placing related clichés in such a taxonomy helps make explicit the relationships between them. At the top of the hierarchy may be some under-specified input-output specifications that embody some computational idea but they may not have any executable plans associated with them. Further down the hierarchy are more constrained clichés that may have implementations associated with them. For example, in the build-graph family of clichés, the top `build-graph` input-output specification simply expects some input data, and constrains its output to be a graph. At this level of generality, there are no plans associated with the input-output specification. The more specific algorithmic clichés in the hierarchy, such as the `build-complete-graph` input-output specification, have an associated plan which gives the abstract algorithm for constructing a complete graph.

There are many ways to build a graph. How do we go about organizing them into a hierarchy? We observe that there are different ways of specifying the inputs from which to build a graph. A graph is defined by a set of nodes, and a set of arcs. Based on this, we can expect the inputs to a build graph algorithm to include a node test indicating whether an item is a node, and an arc test indicating whether there is an arc between two given nodes. If the tests are given, we also expect the input to include some potential node set on which we can test for nodes, or more generally, some input set from whose elements the nodes of the output graph can be computed. There are other graphs which can be concisely specified by its abstract properties, e.g., building a generic complete graph with a given number of nodes. The node test and the arc test are implicit and known. Other examples include building a uniform tree of a given depth and a given branching factor.

We also observe that different kinds of graphs can be constructed by different algorithms efficiently. If the output graph is forward with respect to the input sequence from which it is constructed, then we have an algorithm that capitalizes on this fact to halve the running time. On constructing a node, the algorithm does not have to look for arcs that point from this node backwards to other nodes that might have already been made. The output graph is also acyclic.

From these observations, we organize the build graph algorithms according to the abstract properties of the inputs they are given and the abstract properties of the output graph they produce. Thus, the levels are organized by the nature of the output graph the algorithm computes, e.g., *build-uniform-tree* and *build-random-graph*, and by the kind of inputs the algorithms expect, e.g., *build-graph-from-input-list*. Under *build-graph-from-input-list*, the algorithms are organized by the presence

or the absence of an *arc-test* role and a *node-test* role. They are further distinguished by the nature of the given tests. The given tests can further be classified: An arc test is *weak* if it requires at least one of its given inputs to have a corresponding node. An arc test is it strong otherwise, for there are fewer conditions required of its two inputs. A *weak-left* arc test is a weak arc test whose first input item has a corresponding node.

A hierarchy of related clichés serves as a convenient place for placing information common to the various clichés. Design heuristics about choosing between member clichés can be kept in the hierarchy. There can be classifier procedures associated with the cliché family so that when some inputs or roles of an instance of the most general cliché are given, the procedure can analyze them and provide abstract descriptors that help classify the instance into a more specific cliché type in the hierarchy. A procedure attached to the build graph hierarchy examines the arc test given and labels it weak or strong, descriptors useful for classifying the given instance.

A previous attempt to organize build graph algorithms into a cluster focuses exclusively on their structural similarities. It turns out that this group of algorithms have rather disparate structures that cannot be easily structured for sharing. One attempt was made to construct a huge plan with a great number of roles and many constraints that limit the implementation of the roles automatically. It is, however, too big and complicated to be easily understandable, and goes against the idea that a cliché should be self-contained and concise.

As a concrete example of an algorithmic cliché in the library, consider the *build-rooted-forward-dag-with-weak-left-arc-test* plan which is used in our scenario. It occurs further down in the build graph cliché hierarchy. The plan for this cliché is rather big and complicated; we use a code template to illustrate the plan in Figure 4-8.

In Figure 4-8 those operators in curly brackets are roles in the plan that need to be implemented because they are abstract input-output specifications. Arguments in curly brackets are input data to the plan, e.g., `root-item`. The key information missing in the template are the types of the roles (rendered as operators here). Even though the code looks like Lisp, the types of the data items in the template may be abstract. For example, the type of `arcs`, `node-q` and `input-items` are abstract lists. For clarity, they are all left out of the template to give the reader a bird's eye view of the plan. The computation specified in this plan has a nested iteration: the outer loop scans through the input items in order, and the inner loop scans through the intermediate node list that is being computed. In the inner loop, if two input items pass the `arc-test`, then an arc is made out of their corresponding nodes. If the corresponding node for the second input item is not made yet, it is made. All nodes and arcs made are kept in respective lists, in the order they are made. An index counts the number of nodes made.

```
(Let ((Root (MAKE-NODE ROOT-ITEM 0)))
  (Multiple-Value-Bind (Arcs Node-Q Index Current-Node)
    (Collect-Fn
      '(Values T T T T)
      #'(Lambda () (Values Nil (MAKE-1-LIST Root) 1 Nil))
      #'(Lambda (Arcs Node-Q Index Current-Node Current-Token)
          (Multiple-Value-Bind (Arcs Node-Q Index Current-Node
                                Current-Token)
            (Collect-Fn
              '(Values T T T T T)
              #'(Lambda () (Values Arcs Node-Q Index Nil Current-Token))
              #'(Lambda (Arcs Node-Q Index Current-Node Current-Token
                         Queue-Node)
                  (Let* ((Queue-token (NODE-TO-INPUT-MAP Queue-Node)))
                    (When (ARC-TEST Queue-Token Current-Token)
                      (When (Null Current-Node)
                        (Setq Current-Node (MAKE-NODE Current-Token))
                        (Setq Index (+ Index 1))
                        (ENDPUSH Current-Node Node-Q))
                      (Let ((Arc (MAKE-ARC Queue-Node Current-Node)))
                        (ENDPUSH Arc Arcs))))
                  (Values Arcs Node-Q Index Current-Node Current-Token))
              (SCAN-SEQUENCE Node-Q))
            (Values Arcs Node-Q Index Current-Node)))
      (SCAN-SEQUENCE Input-Items))
    (MAKE-GRAPH Root Arcs Node-Q Index Current-Node)))
```

Figure 4-8: The Build-Rooted-Forward-Dag-With-Weak-Left-Arc-Test Cliche.

4.4.3 Going Beyond the Plan Calculus

Many algorithmic clichés can be conveniently represented in the Plan Calculus, but there are some clichés needed in the scenario which cannot be represented as plans. For example, the Concatenate cliché embodies the computation that takes in a variable number of inputs, coerces non-sequences into sequences, and returns the concatenation of the sequences. Such a scheme cannot be represented in the current plan formalism. In such a case, a program generator can be programmed to generate appropriate plans given the actual number of arguments used for Concatenate, and their respective types. Another example involving a program generator is the tokenize cliché. Given the specification of a grammar written using regular expressions, a lexical analyzer generator can be used to generate a tokenizer to parse a text file.

It is unlikely that a single knowledge representation formalism can provide all the facilities one needs in the course of a knowledge codification effort. It is important to use different formalisms to capitalize the special leverage provided by each formalism.

4.4.4 Families in the Cliché Library

The library of clichés built up for the scenario in Chapter 2 is made up of several collections of input-output specifications and plans. The main ones are briefly described below:

The Lisp Family: In this collection, we have mainly input-output specifications for the Lisp primitives used in the scenario, such as the `null` function and the `zerop` predicate. It also contains input-output specifications for a number of simple Lisp procedures, such as the Last-Character function, which takes a string and returns the last character in the string.

The Iteration Family: The Common Lisp SERIES [35] macro package is used to represent iterative computations. They provide rich and powerful iteration abstractions that simplify the design process. The key abstraction functions are codified as input-output specifications in the iteration collection.

The List Family: The abstract data type, `list`, is defined in this collection. Many commonly-used abstract operations that act on the list type are found here, including the plans for `segment` and `sequence-in-between`. This collection also models the various sequence types used in Lisp such as Vector, Lisp-List, and Chain. (We call a Lisp structure a chain if it contains a field for storing a pointer to the next structure; this can be used to encode an implicit sequence of such structures.)

The Graph Family: In this collection, various types of abstract graphs are defined and standard operations on graphs are specified. Specifically, it has plans for the single-source shortest path algorithms and the build-graph algorithms.

The Tokenize Family: This collection is intended to contain a number of tokenize clichés. They codify a number of different lexical analyzer generator algorithms. It also defines the basic token types and generic functions typically associated with

tokens and the tokenization process.

The Miscellaneous Family: This contains input-output specifications for the Prorate clichés, the Ascii-File cliché, and the Output clichés.

4.5 Representing Design Artifacts

The DA uses several artifacts in the detailed design process. The input program description is translated into plans and they serve as the initial design for the automatic detailed design process. During this process, each design step manipulates an evolving design represented as plans and records the design changes made in a decision record associated with the design step. The entire design process is documented by a global *design record* which makes explicit the dependencies between the design steps taken during the process.

4.5.1 Translating Input Descriptions into Plans

The input program description given to the DA is translated into plans, and forms the initial design of the program. This has been done manually. This process can be automated but this was not done since the input language is expected to change in the course of this work. The plans representing the initial program descriptions have been compiled successfully in CAKE. A few points about the translation process are noted below:

- A box is created for each operator application form in the input with the appropriate number of inputs and outputs. All inputs and outputs are by default given the most general type in the modeling level, **data**, because their more specific types are unknown as yet. Throughout the design, their types will become more specific as more information is added. A box may be an input-output specification or a test specification.
- The `FOR-EACH` form in the input description can be viewed as a macro that gets expanded into the SERIES function, `collect-fn`, and parsed into plans.
- Each use of a cliché in the library is recognized and an instance of the cliché frame is associated with the use.
- For simplicity, we assume the input descriptions are functional. Though output to streams are global side-effects, we model such side-effects by explicit data flow.

As indicated earlier, the `ratio` plan shown in Figure 4-1 on Page 57 is a translation of part of the program description given in the scenario.

4.5.2 Finer Points about Representing Designs

The name of a cliché codified in the Plan Calculus is a type, i.e., it denotes the set of all possible execution traces involving that plan. So far, we have used the term *plan* for both the type and the token. It is important to clarify the distinction here: we shall from now on call the name of a cliché, a plan type and an instance of the cliché, a plan.

The plan calculus formalism given so far is clumsy when it comes to defining an overlay between two plans. Overlays are functions from plan types to plan types. Since CAKE is chiefly a propositional reasoner, the design process is carried out on specific anonymous plans, not on the respective plan types. Intermediate designs change frequently, and if we are to remain true to the semantics of the Plan Calculus, we need to generalize an intermediate design whenever an overlay is needed. This is a lot of work to do without much attendant benefits. In this work, we simulate an implementation overlay between two plan types by a set of correspondences (equalities) between the anonymous instances of the respective types. If the formal overlay is needed, we can extract its definition from the set of correspondences.

Similarly, even though data plans are designed for modeling structured objects in plan calculus, these are not used because many structured objects in a current design are frequently changing. We choose to model Lisp structures in CAKE with separate unary functions for each field and annotating outside CAKE logic the constituent fields of the current structured type.

It is useful to note that the above two points are measures taken solely for efficiency reasons. Frequently, it is more convenient and conceptually clearer to think with plan types and overlay functions than with the optimized representations.

4.5.3 Representing Design Dependencies

Each design step has an associated *decision record* which records the changes the step makes on the previous design. A design decision may depend directly on other design decisions or indirectly via design constraints. A *design record* is a set of these interdependent decision records that is the outcome of the detailed design process.

Decision Record: An explicit decision record is made for each design step taken. Each design step can either be initiated by the user or by the automatic detailed design mechanism of the DA. There are several fields in such a record:

- Step-Type: This records the type of the design step that is associated with this record.
- Status-Node: This contains a boolean CAKE node whose truth indicates that the associated design step has been taken. If the design step has been retracted, this status node will revert to unknown.

- Assert-Constraints: This contains a list of CAKE nodes that are asserted to be true in the current design by this design step.
- Retract-Constraints: This contains a list of CAKE nodes that have been retracted in the current design by this design step. For monotonic design steps, this field is empty.
- Supporting-Nodes: This contains a list of CAKE nodes whose conjunctive truth supports the design step represented by this record. If any of the supporting nodes are unknown or false, then this design step must be retracted.

For each decision record, a constraint is added to CAKE which asserts that the truth of the status-node of the decision record implies all the nodes in its assert-constraints field. When the DA decides to take the design step, it first retracts the nodes that are in the retract-constraints field of the record, and then asserts the truth of the status-node of the record. To support automatic retraction of the design step, a *notice-change-truth* procedure can be added to the status-node of the record. This procedure asserts the nodes in the retract-constraints list when any of the supports become unknown or false. Note that it does not need to retract the assertions made in the assert-constraints list because those constraints automatically go out when the status node is retracted due to the first constraint mentioned in this paragraph.

The design record of a design process can be kept as a chronologically ordered list of all design steps taken during the entire design process. Those design decisions which are no longer active at the end of the design process can be identified easily since the status nodes of their respective decision records will not be true.

4.6 Manipulating Design Artifacts

This section describes how design artifacts are modified by the design steps and three notions of when a design represented as a plan is considered complete.

4.6.1 Design Steps

In this subsection, we expand on the discussion in the previous chapter on the repertoire of design steps the DA is expected to have. We also describe how each of the steps can install design dependencies among themselves. The design steps described here have not been implemented. The nature of the discussion in Chapter 3 mandates the use of more general terminology which is not done in this chapter. The design steps described in this subsection are specific to the plan representation of programs. Their relationships with those discussed in Chapter 3 will be pointed out in their respective discussions.

Before delving into the details of the design steps, it is convenient to define some terminology which will become useful in later description. A design (represented as a

plan) has four kinds of constraints: box constraints (or type constraints on the roles of the plan), data flow constraints, control flow constraints, and other arbitrary logical constraints associated with the design. Every constraint has a *sponsor*. A sponsor of a constraint is a boolean node that brings about the existence of the constraint in the design. A sponsor is defined intentionally: if the sponsor decision of a constraint is retracted, then the constraint is also retracted. For example, the sponsor of the boxes and constraints of a plan instance may be the assertion that the plan instance is of a particular plan type. In plan calculus, it is convenient to represent constraints on a plan instance in a bundle by a single assertion about the type of the plan instance. This need not be the case, we could have separate individual constraints on a plan instance. A design step may also add new boxes and constraints to a design. The decision corresponding to the design step becomes the sponsor of these new boxes and constraints. By default, a constraint without any sponsor will have itself as its sponsor. A constraint can have multiple sponsors: the constraint remains true until all its sponsors are retracted.

Since a data flow or control flow constraint cannot exist without the existence of their source and destination boxes, whenever we add such a constraint in a design step, the sponsors of the source and destination boxes of the constraint must also be part of the supporting nodes of the step. We call these implicit sponsors of data flow and control constraints *flow sponsors*.

Select Algorithmic Cliché: This step is invoked on an input-output specification that has no current implementation associated with it. For example, if this design step is invoked on an instance of the `collect` input-output specification, it looks up the cliché library to see if there are any overlays whose range is `collect`. If there are more than one feasible overlay, the select algorithmic cliché design step chooses one based on some design heuristics.

Once an overlay is chosen, an instance of the domain type of the overlay is made, and the overlay is applied to the instance and asserted to be equal to the `collect` input-output specification instance. In Figure 4-9, we have implemented the instance of `collect`, `c1`, using an instance of `series-collect`, `s1`. This implementation decision adds the following constraints: `(= (in s1) (in c1))`, `(= (out s1) (out c1))`, `(= (arg1 s1) (arg1 c1))`, `(= (arg2 s1) (arg2 c1))`, `(= ((!as series list) (arg3 s1)) (arg3 c1))`, and `(= (output s1) (output c1))`. The above constraints, together with the type of the `s1` box, are kept on the assert-constraints field of a new decision record created for this design step. A new status node, called it `d1`, representing this design decision is also created, and it becomes the sponsor for all the abovementioned new constraints. A constraint is asserted in the design that `(implies d1 ac1)` where `ac1` is the conjunction of all constraints in the `assert-constraints` field of the decision record. In this way, the new constraints `ac1` are automatically added whenever the node `d1` is true, and retracted when `d1` is no longer true.

This design step, being monotonic, does not remove any existing constraint from

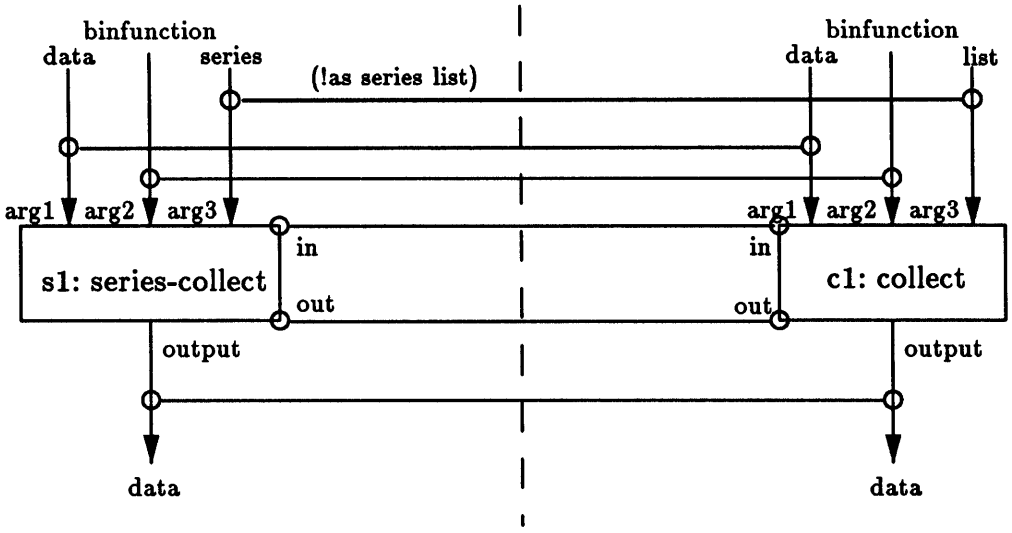


Figure 4-9: An Example of Implementing an Input-Output Specification.

the design, and therefore has an empty `retract-constraints` field on its decision record. Note that if the `c1` box did not exist at all, then this design decision should not be in the design because it would be useless. Thus one of the supporting nodes for the decision record is the sponsor of `c1`. In general, a design decision that involves the implementation of a box is dependent on the sponsor of the box. This is because if the box is removed by the retraction of the sponsor, then the implementation decision may no longer be valid.

Depending on how this design step is triggered into action, there can be other nodes in the `supporting-nodes` field of its decision record. For example, if the decision has been to implement a single-source shortest path specification using an algorithm for DAGS, then the conditions asserting the fact that the input graph is a DAG should be in the `supporting-nodes` field. In this way, if the assertion that the input graph is a DAG is retracted later, the choice can be retracted automatically.

For an input-output specification that stands for some cliché instance, this design step consults its cliché frame as to how the current input-output specification should be implemented. Local design heuristics about choosing are kept in the cliché frame for the ease of this design step.

For those input-output specifications which are operations on an abstract type codified in the library, they can also be implemented by this step. However, as discussed earlier in Chapter 3, they are better viewed as selecting a data structure for the abstract type.

Select Data Structure: This design step has been discussed in some depth in the previous chapter. It suffices to show how the decision record for such a step

is constructed. Take the same example shown in the last chapter, where the set of operations on the token sequence in the scenario are: Sequence-In-Between, EndCons and Insert.

This design step is really a union of several select algorithmic cliché design steps. Each of the constituent design step will have a decision record like before. In addition, there is a master record for this design step itself whose **assert-constraints** consists of the status nodes of all its constituent design steps. Its **supporting-nodes** field, likewise, consists of the union of all the respective **supporting-nodes** of its constituent steps.

Select View: When an algorithmic cliché is mentioned in the program description, a box is created in its place. The box is marked to be associated with the cliché but the types of its inputs and outputs are the most general type available. For example, the use of the prorate cliché in the `lineout` design has two input arguments and one output argument, all of them are given the initial type, **data**. For convenience of later reference, let the new box associated with this use of the prorate cliché be called (`!box prorate.1 2`). `Prorate.1` is an instance of the prorate spec cliché. The `!box` operator is used to provide a convenient syntax for relating the box and its associated cliché instance. The second argument indicates how many input arguments the box takes. We say that (`!box prorate.1 2`) *stands for* `prorate.1`, alluding to the intended use of (`!box prorate.1 2`) as a proxy for the specific prorate specification that may be chosen later.

In addition to the motivation for this design step explained in the previous chapter, this step is also important because our input language do not require type declarations. The DA can make use of type declarations the programmer specifies but they are not mandatory. The underlying knowledge representation and reasoning system is, however, based on a typed logic, i.e., every term in CAKE has a type. Much of the reasoning needed in this work is thus cast in a type framework.

The select view design step makes a guess of what the correspondences might be using the typical-call property kept on the cliché frame. It does so by installing the correspondences between the parts of the cliché frame and the inputs and outputs of the box that stands for the cliché in the design. These correspondences are the **assert-constraints** of the decision record created for such a design step. A supporting condition for this step is the sponsor of the box. No constraints are removed by this design step and hence, the **retract-constraints** field of its decision record is empty.

Coerce Type: To make the description of this design step in the last chapter more concrete, we reconsider the example given in the previous explanation of this step. The top half of Figure 4-10 shows the plan representation of part of the input `justify` design taken from the scenario. The specific part of the input design represented is shown in Figure 4-11. `auxiliary-justify` is an auxiliary function used to encode the computation in the inner loop of `justify`. The lower half of the figure shows the structure of the plan after the coerce type step has been applied to the **outfile**

argument. Part of the Ascii-File cliché is some knowledge about how we can view an ASCII file as an output stream or an input stream.

This is an example of a design step that involves non-monotonic changes in the design, i.e., some constraints about the design are retracted. In this case, the modified data flow affects the correspondences in the overlay application. With the help of the labels in Figure 4-10, we can see that the new constraints added by this step are as follows. Those involving the overlay correspondences: `(= (out j1) (out k1))`, `(= (arg1 e1) (arg2 j1))`, and `(= (output c1) (output j1))`, those involving new data flow constraints, `(= (output e1) (arg1 c1))`, `(= (out e1) (in c1))`, `(= (output op1) (arg1 k1))`, and `(= (out op1) (in k1))`, and those involving the new boxes added, `(close k1)`, and `(open e1)`. These should all go into the `assert-constraints` field of the decision record for this design step. The `retract-constraints` field of the record will contain the following constraints which are retracted by this step: `(= (out op1) (out j1))`, `(= (output op1) (output j1))`, and `(= (arg1 c1) (arg2 j1))`.

The supporting nodes for this design step are the sponsors for all the constraints that are removed by it and the flow sponsors for all the the data flow constraints added. The first group of sponsors, in the scenario, turns out to be a single design decision: the *do-this* design step implicitly given by the program description provided by the user. This design decision brings in to the current design all the constraints that were removed by the later coerce type step.

Omit Operation: This is another example of a non-monotonic design step. Similar to the coerce type step, the decision record for this step can be constructed from the constraints added and deleted from the current design. The supporting nodes of this step should include the conditions that satisfy the postconditions of the omitted operation.

Exchange Operation for Input: This design step removes a role in a plan, adding an input to the plan in place of the removed role, thereby changing the input-output specification that the plan implements. Figure 4-12 shows the `width` plan with its `preceding-item` box. Figure 4-13 shows how the `preceding-item` box is moved out of the `width` plan to become an input to a new `width-2` plan. A new input-output specification and a new overlay are also created in the process. As this step is used more as a subroutine for other design steps that make use of the new plans created, one supporting node of this step is the design decision that invoked it. Other parts of the decision record is constructed in a similar fashion as previous non-monotonic design steps.

Pre-Compute Mapping: An example of this design step which is used in the scenario is shown in Figure 4-14. The first half of Figure 4-14 shows the `(!store width-2)` plan. This step uses the *exchange-operation-for-input* design step described earlier to produce the `width-2` plan. `(!put width)` is an input-output specification that stores its second argument in the field chosen to represent the width mapping in its first argument. The first argument must be a Lisp structure. The input-output

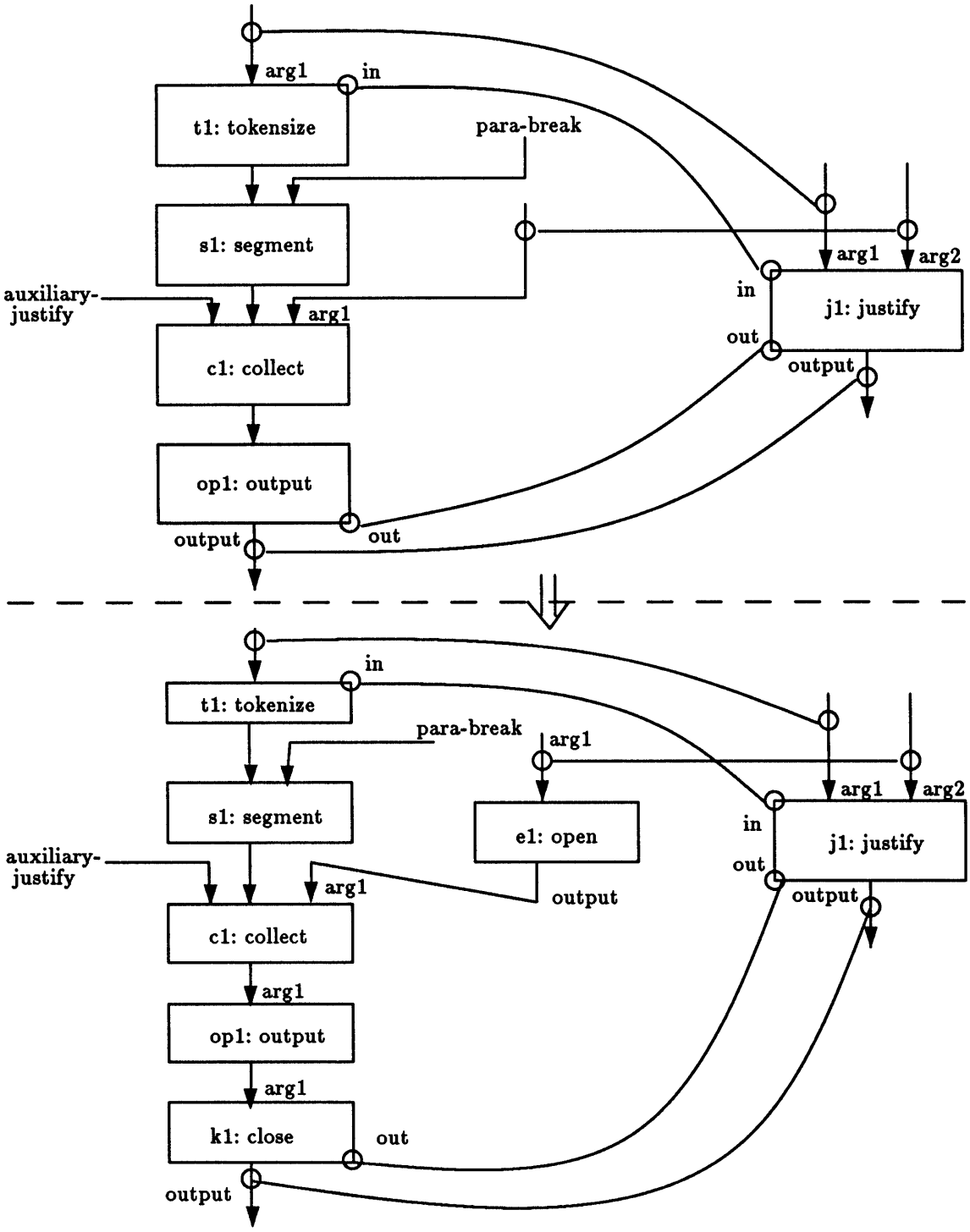


Figure 4-10: Viewing an Ascii File as an Output Stream.

```
(DESIGN justify (infile outfile)
  (FOR-EACH ((paragraph (Segment (Tokenize infile) 'para-break)))
    (FOR-EACH ((line ( ... paragraph)))
      (... outfile line))
    (Output outfile #\return)))
```

Figure 4-11: Input Program Description for Justify.

specification (!put width) can be generated dynamically by the DA once we have decided to store the width mapping explicitly in a field on the domain structure. The second half of Figure 4-14 shows the plan to compute the entire mapping. The previous function came as an implementation of the preceding-item cliché. A design heuristic in the preceding-item cliché motivated the transformation to move the computation out of the loop that computes the mapping.

Figure 4-15 shows where the actual invocation of the pre-computation is placed. The dotted line shows where there is implicit data flow from (!put width) to a use of (!get width). The DA must ensure that all uses of (!get width) come after the pre-computation of width. A note is also made to implement future accesses to this mapping by (!get width); all such implementations will depend on this design step. The installation of design dependencies of this non-monotonic step is similar to those described earlier.

Other examples in the scenario that require the use of this design step are: the stretch mapping and the successor mapping of tokens which arises from implementing the token sequence as a chain-shelf.

Cache Mapping: One condition for which this design step is applicable has been given in the previous discussion of the step. Here, we reconsider the applicability condition with the same example: we want to cache the mapping ratio for a second mapping arc-ratio. As explained in the last chapter, we can store the mapping on the domain elements of arc-ratio which are Lisp structures. As shown in Figure 4-16, the design step adds a new field and stores the computed ratio values directly in the field when ratio is being computed. The (!put ratio) box and the new data flow constraints (rendered as thick lines in the figure) are added by this step.

The only place where arcs are created in the justify design is in the make-arc box in the build graph plan. The DA searches backwards, from the make-arc box, for a ratio box. It does so, when necessary, by looking into the implementations of the intervening boxes. In this case, the first ratio box it comes across is in the implementation of arc-test.

For ease of reference, various ports in Figure 4-16 are annotated, and the proof is shown in the same figure. Before adding the new box and the new data flow constraints, it must verify that (M a) = s. The key observation to make here is that all the constraints used in the proof are local to the plan. The only constraint that

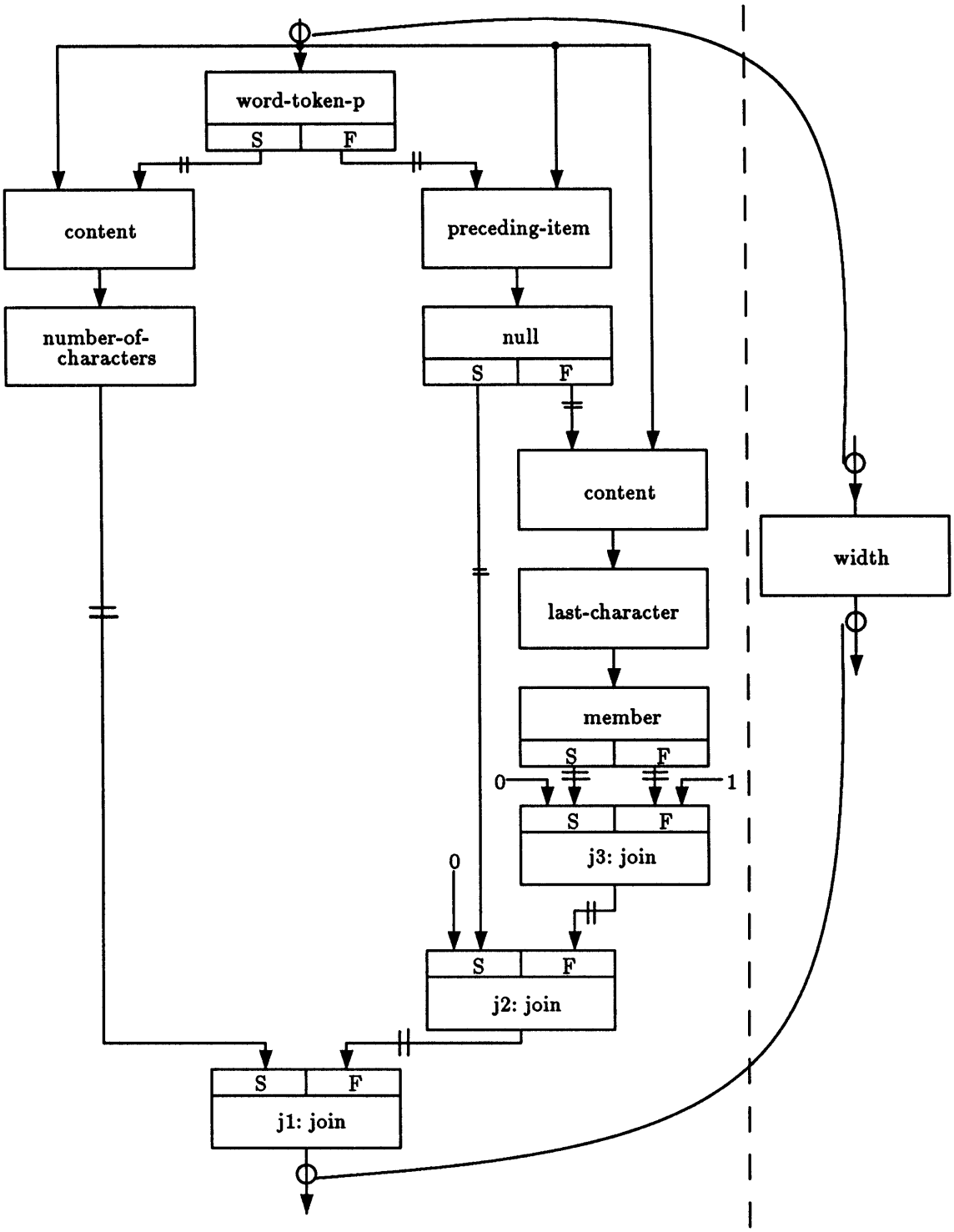


Figure 4-12: Exchange Operation For Input Design Step: The Initial Width Plan.

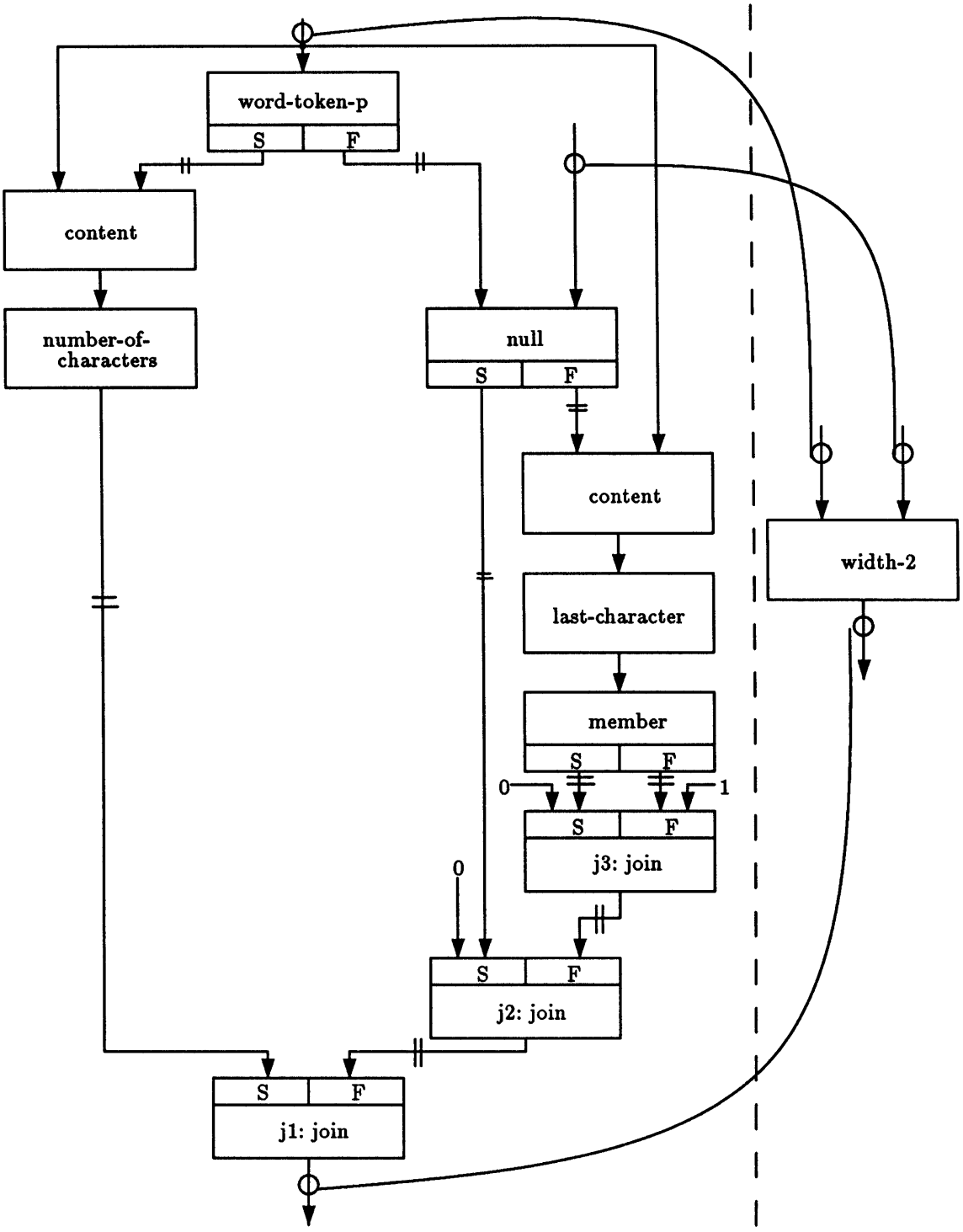


Figure 4-13: Exchange Operation For Input: The Result Width Plan.

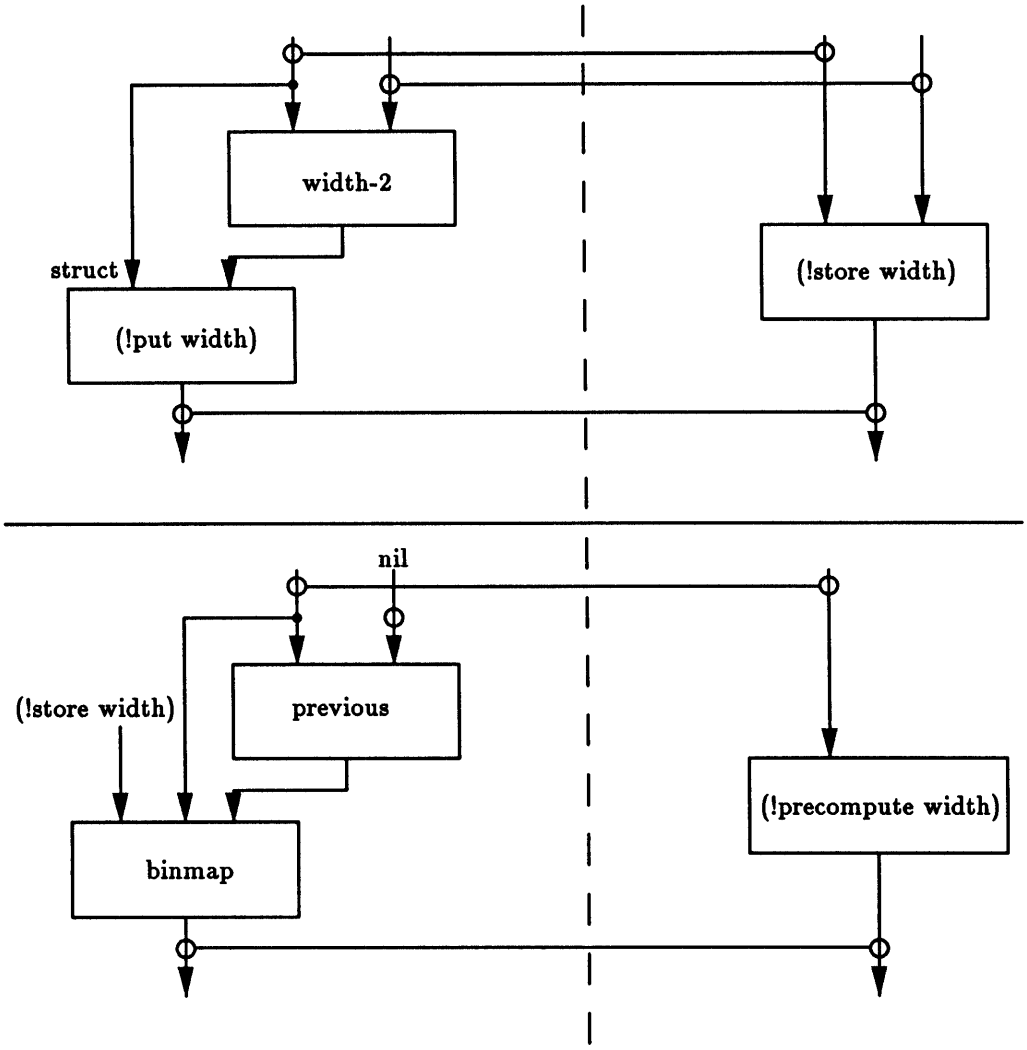


Figure 4-14: Pre-Computing the Width Mapping.

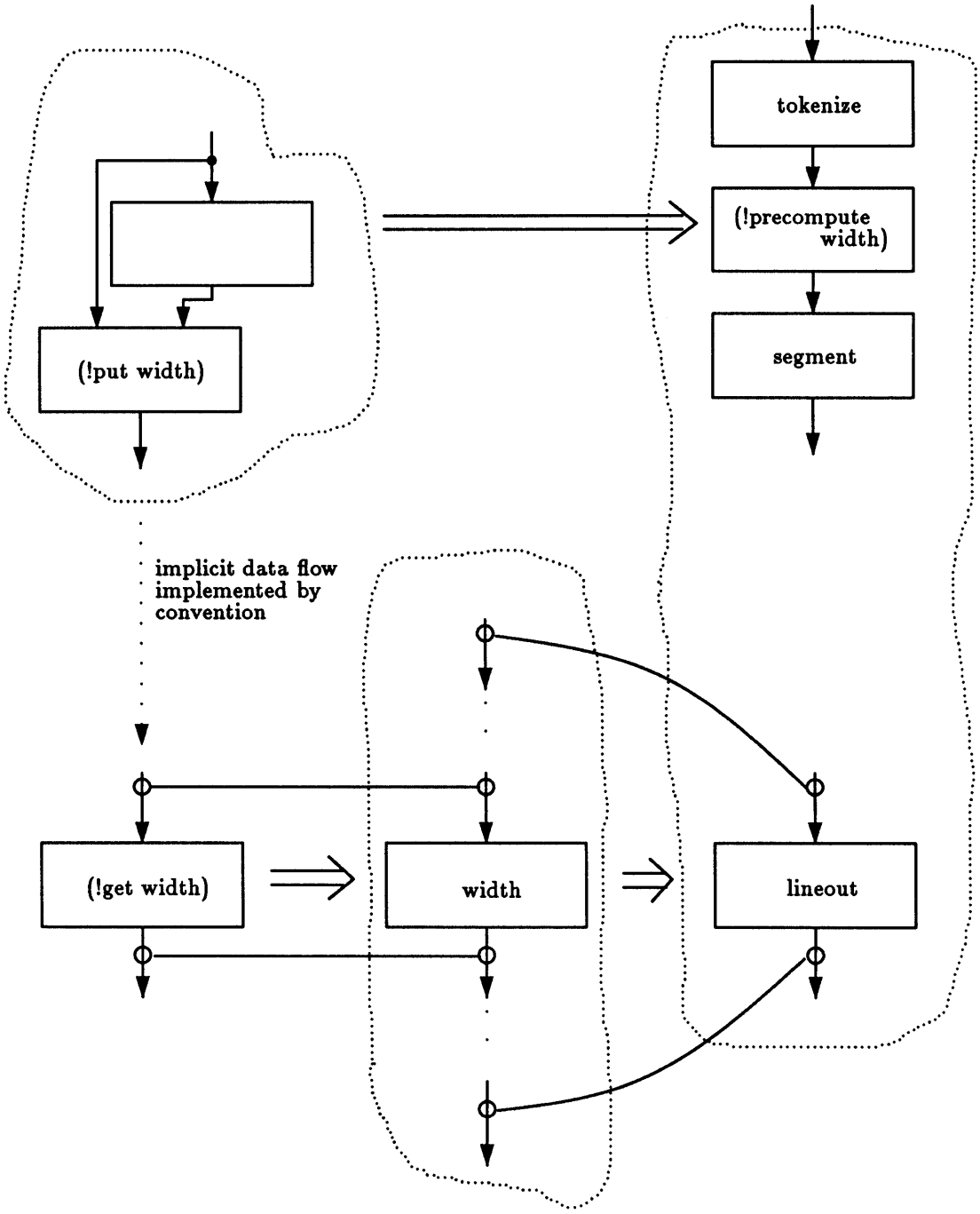
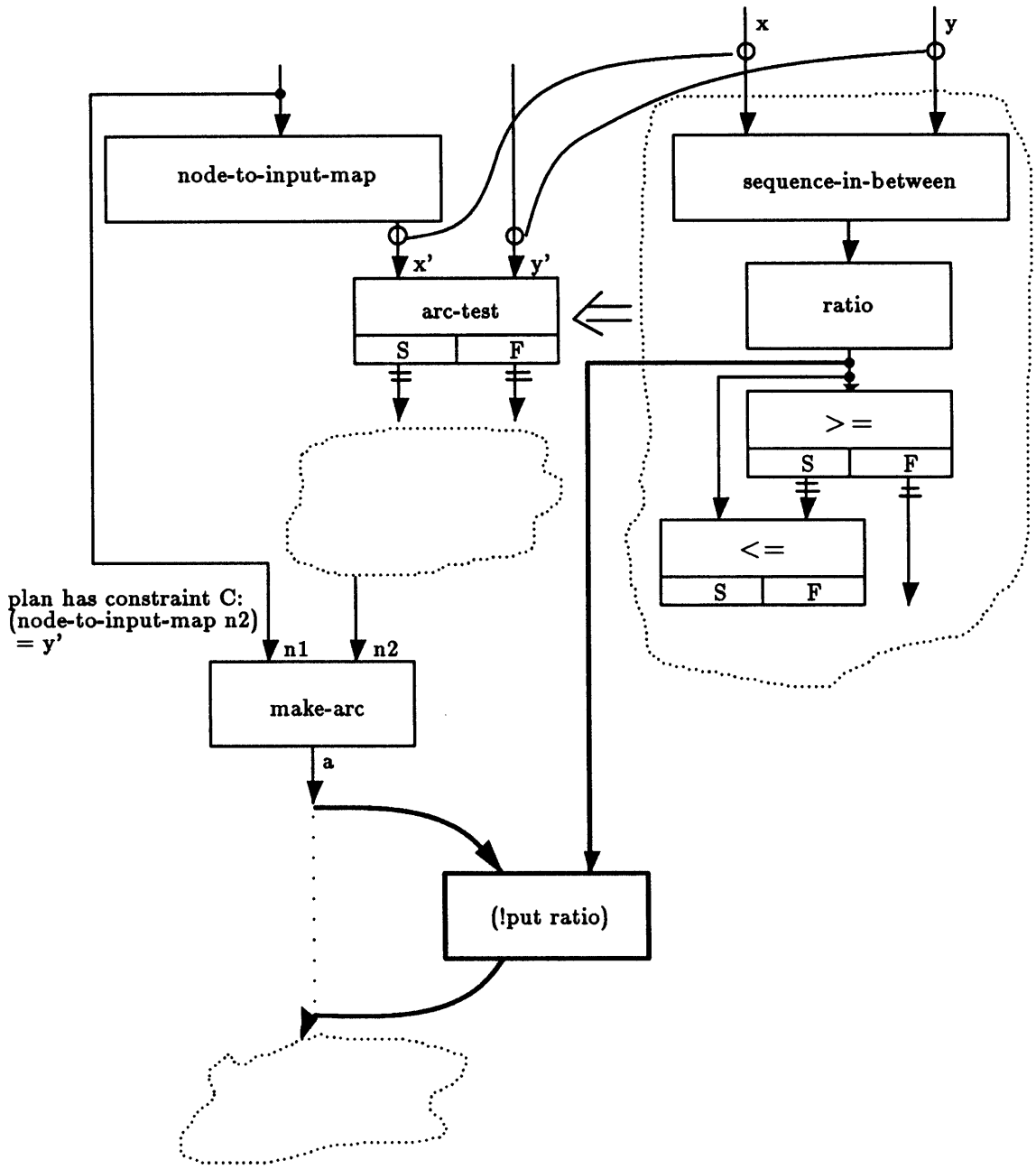


Figure 4-15: Positioning the Pre-Computation of Width.



(token-sequence a)
  = (sequence-in-between (node-to-input-map (source-node a))
                         (node-to-input-map (destination-node a)))
    ; by definition of token-sequence
  = (sequence-in-between (node-to-input-map n1) (node-to-input-map n2))
    ; by constraint of make-arc box in the above build graph plan
  = (sequence-in-between x' y')
    ; by constraint of node-to-input-map box in the above build graph plan
    ; and by constraint C in the above build graph plan
  = (sequence-in-between x y) ; by the overlay correspondences in V
  = s ; by the constraint of sequence-in-between box in ratio plan

is not obvious is `(input-to-node-map n2) = y1`. In the context of the build graph cliché, it says that the corresponding input item of the destination node of an arc is derived from the second argument to the `arc-test` box. This, we assume, is a constraint in the build graph cliché.

It is also necessary to make sure that the ratio box from which the result is fanned out is always there whenever an arc is made. If the ratio box is hidden within some conditionals in the implementation of `arc-test`, then there could be times when the result is not available when the arcs needed it. In our scenario, this is not the case; ratio is always computed before any arc is made.

The design dependencies that need to be installed in this design step are similar to the other non-monotonic design steps. The only difference of note here is that all the constraints used in the verification proof should also be made supporting nodes for the decision to take this design step.

Design Tree

We have drawn plan diagrams to illustrate the action of the above design steps. It must be apparent by now that the illustration can get to be very large and complicated when the design is large. As a shorthand, we introduce a different illustration of a program design which is more compact but does not contain as much information. A *design tree* shows the hierarchical decomposition of a program design without its data flow and control flow arcs. As an example, Figure 4-17 shows the design tree for a program design of the `ratio` function from the scenario. An arrow shows the application of an implementation overlay from some plan to an input-output specification. The roles of the plan used are drawn as the children nodes of a tree whose root is the plan. In Figure 4-17, `ratio-plan` has five roles whose types are shown as its immediate descendants. A small dot below a role indicates that the role is implementable. In the figure, `zerop`, `/`, `series-map`, `series-sum`, and `(!get token-stretch)` are implementable. They form the *fringe* of the design tree.

The *select algorithmic cliché* design step and the *select data structure* design step can both be seen as adding implementations to non-fringe nodes in a design tree. In an *idealized design process*, no non-monotonic design steps are taken. In such a case, every design step taken augments the design tree of the program design from the previous design step. In such a case, the design tree can be viewed as a rather compact representation for the design record of the idealized design process. In general, non-monotonic design steps may add new roles to and remove old roles from an existing plan in the design tree. The design tree is still useful to show the change in the design more compactly than using plan diagrams.

4.6.2 Propagating Constraints

A key step in the design cycle described in the previous chapter is the constraint propagation step. Various design steps bring specific design information from a cliché

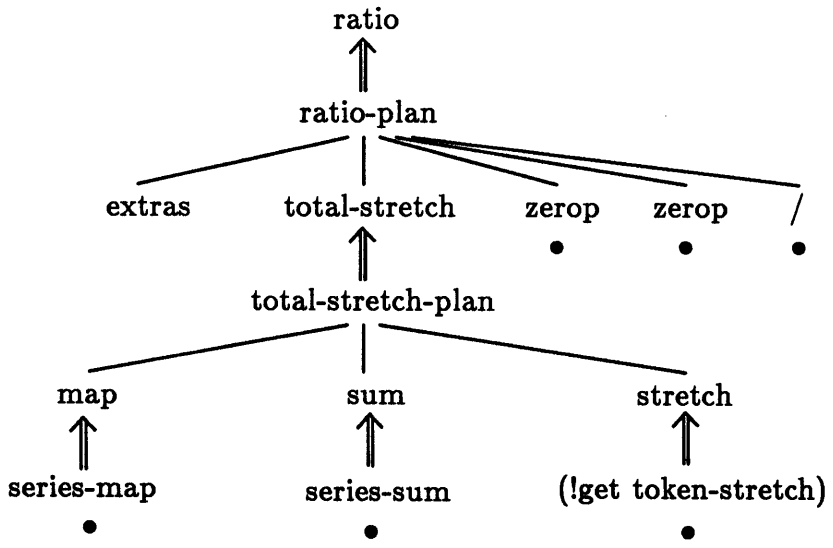


Figure 4-17: A Design Tree.

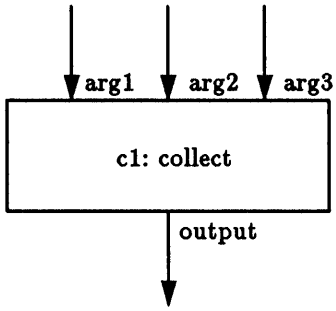
to bear on the current design. This, however, has to be propagated to other parts of the design where they may act. For example, the typical call mechanism described above constrains the output of the single-source shortest path computation to be a path, i.e., a sequence of arcs. This information is propagated through data flow arcs to the `lineout` computation.

We illustrate the kind of reasoning needed to propagate such constraints in CAKE using the following example with the help of Figure 4-18. This will also serve to show how some *make-assumption* design steps are triggered into place by this constraint propagation step. In the figure we show the `collect` box (named `c1`) from the `justify` design with the known type constraints on two of its input arguments. Suppose the current CAKE database contains the following true assertions:

```
((!list arc) (arg3 c1)) ∧ (output-stream (arg1 c1))
∧ (= lineout (arg2 c1))
```

where `(!list arc)` is a specialization of the abstract list type whose elements are constrained to be arcs. How can we deduce that `lineout` has the signature: `output-stream x arc` → `output-stream`?

We have programmed CAKE to instantiate the parameterized projection axiom shown in the same figure. The axioms can be used to deduce new type constraints on the second input argument and on the output of the `collect` box based on type constraints on the other arguments. `(!from output-stream (!list arc) output-stream)` denotes a functional type. If `f` is a member of the functional type `(!from t1 t2 t3)`, then `(f a b)` is constrained to have type `t3` if `(t1 a)` and `(t2 b)` are both true. The



Current Status: ((!list arc) (arg3 c1)) ∧ (output-stream (arg1 c1))
∧ (= lineout (arg2 c1))
Axiom: ∀ i,f,l,t1,t2 s.t. t1 is (!list bt1)
(defined (collect i f l)) ⇒
(t1 l) ∧ (t2 l) ⇒
(t2 (collect i f l)) ∧ ((!from t2 bt1 t2) f)
Assumption 1: (defined (collect (arg c1) (arg2 c1) (arg3 c1)))
New Deductions: (output-stream (output c1))
∧ ((!from output-stream arc output-stream) lineout)
Assumption 2: triggered by the last deduction
((!on output-stream arc output-stream) lineout)
or lineout: output-stream x arc → output-stream

Figure 4-18: Parameterized Type Propagation.

type does not constrain `f`, however, to be undefined on applications whose arguments are disjoint from the types of the formal parameters of `f`. Thus to draw the first step of the desirable conclusion, we need to assume that the application is defined. Furthermore we need to assume that we can generalize from this single use of `lineout` that it is the only correct use, and restrict it to be undefined outside the projection types given.

In short, with the help of the parameterized projection axioms, we need to make two explicit assumptions in order to arrive at the desired conclusion. These assumptions are in fact explicit *design decisions* the DA has to make in an effort to advance the design. A list of such assumptions can be maintained and categorized so that they can be retracted if contradictions involving them arise later. For example, if there is a new use of `lineout` in the design which indicates that `lineout` is a member of another functional type (`!from output-stream t output-stream`) (where `t` is any type), then we need to retract the strong assumption made earlier about `lineout` being undefined outside `output-stream x arc`. Instead, we can assume that we can generalize from the two uses of `lineout` that `lineout` has the signature: `output-stream x (*or t arc)` $\rightarrow$ `output-stream`.

The example given indicates a potential problem with this design step: if CAKE is not properly guided in the instantiations of the given axioms, it will be very slow because it will make a lot of useless deductions. In CAKE, instantiations of these axioms are done via *noticers* or pattern-directed procedures. We can selectively instantiate the axiom on instances by their fixed syntactic properties. For example, we can consider only type instances that are strictly subtypes of the `data` type. If there are two types one of which is a subtype of the other, then we need only instantiate the more specific type because type subsumption in CAKE can complete most of the reasoning needed in the more general type.

A related note to make here is how the decision record for the *make-assumption* design step can be constructed. This design step is obviously a monotonic step, and the constraints added is the assumption made. The supporting node in the first assumption about the defined-ness of an application does not need any supporting nodes since there are no good candidates for removing the assumption automatically. For the second assumption, the supporting node is the node that triggers it: ((`!from output-stream arc output-stream`) `lineout`).

4.6.3 When is a Design Complete?

A key need of any automatic design process is to know when a design is done so the process can stop. In the representation framework of the plan calculus, there are three useful notions of completeness:

1. Complete with respect to Roles: A design is complete with respect to roles if all the fringe boxes in the current design tree are implementable.

2. Complete with respect to Structures: This notion subsumes the previous notion of completeness and furthermore, requires that the data flow and control flow constraints at every level in the design tree be implied by constraints involving the level below it.

3. Complete with respect to Plan: This notion subsumes the above two notions, and furthermore, requires that *all* constraints, both structural constraints, as well as general logical constraints (in the preconditions and postconditions of boxes) in every level are implied by the same in the level below it.

Complete with respect to plan is the strongest kind of completeness. It is however difficult to prove that arbitrary logical constraints follow from other constraints given CAKE's incomplete reasoning. In this work, we assume that complete with respect to structures suffices. Given that almost all constraints in our plans come from data flow and control flow constraints, this captures a large part of the design process.

This weaker notion of completeness can be tested by the following checks: A current design is complete with respect to structures if (1) all fringe boxes in the design tree are implementable; and (2) every input of every fringe box in the design tree gets its input from either an output of an implementable box in the design tree, or it is a constant that is implementable, or it is mapped to an input of the top-level input-output specification. (1) is simply a check for completeness with respect to roles, and (2) ensures that all data flow constraints are accounted for. We do not need checks for control flow constraints because they follow logically from the formalization of control flow arcs and overlays.

Chapter 5

Related Work

Our work on the DA spans several different dimensions of automatic programming research. The key dimensions are:

- Program Synthesis: Program synthesis systems accept some kind of user specifications, and output some *executable code.*
- Support for Detailed Design: These systems support detailed program design by automatically making some implementation choices and maintaining the design rationale behind those choices. These include systems that select data structures and algorithms; they may or may not produce executable code.
- Support for Specification: Systems that support specification can *criticize* the given specification by detecting inconsistencies and incompleteness, and they may provide corrections to the detected problems.

As some of the reviewed work spans more than one of the above dimensions, the following account will discuss each work individually in broad categories and indicate how they are related to the DA along the above dimensions.

5.1 KBEmacs

In the Programmer's Apprentice project, KBEmacs [34] demonstrated the use of clichés in program construction. Also part of the PA project, the DA is intended to go beyond KBEmacs by automatically selecting the more specific plans to construct a program, based on abstract clichés and design criteria given.

As a successor to KBEmacs, the DA extends KBEmacs in the following respects:

Design Editor: KBEmacs goes beyond syntax-directed program editors by supporting editing in terms of the algorithmic structures of programs. The DA supports programming in a fundamentally different level: it interacts with the programmer in terms of design decisions. It allows the programmer to specify more abstract clichés

that may not have a direct implementable algorithmic cliché associated with it. The DA chooses appropriate more specific clichés to implement the given specification, according to the implementation guidelines the programmer sets. By propagating design decisions, the DA can relieve the programmer from specifying details which are already constrained by other design choices made.

General-Purpose Automated Deduction: In the DA, the relationships between clichés and design concepts are explicitly represented as constraints in a general way. The DA propagates such constraints throughout the specification and reasons via such constraints. In contrast, KBEmacs represents constraints procedurally.

Detection of Contradictions: The DA is able to detect contradictions in the program description and design criteria given. KBEmacs does not support such error detections.

Different Kinds of Inputs: KBEmacs accepts inputs from the user in the form of imperative editor commands that act on the contents of the editor buffer. The sequence of commands are highly order-dependent. The input descriptions accepted by the DA are more declarative, and order-independent.

5.2 Deductive Synthesis

Work on deductive synthesis is based on the idea that a program can be extracted from a constructive proof of the theorem about the desired input-output behavior. This approach is very general but it reduces the problem of program synthesis to that of automatic theorem proving, another very difficult problem. Manna and Waldinger's work [19] and Smith's CYPRESS [30] are examples in this category. The key advantage of this approach is that a verification proof comes with any program synthesized. However, this approach is not likely to scale up for larger programs and it is not clear how to generate efficient code based on such an approach.

The deductive synthesis approach is related to the DA only in its goal of program synthesis. The approach taken by the DA explicitly shuns deep deductions and emphasizes the use of detailed knowledge.

5.3 Program Transformation

This approach emphasizes the use of correctness-preserving transformations to either implement programs from given specifications or to improve the efficiency of given programs. The former is commonly known as transformational implementation or vertical transformation, and the latter, lateral transformation. Transformational implementation approach typically takes a high level description of a program and through successive transformations, turn it into an executable program. As it is closely related to the very high level language approach, some systems which belong to both approaches will be discussed later under very high level languages.

An example of lateral transformations is Burstall and Darlington's transformation system [6]. It uses a small set of transformation rules to improve programs which are more understandable to humans but are inefficient. Research in lateral transformation is complementary to the DA research here. Some of the results can potentially be used to optimize the output of the DA.

One of the earliest system to have incorporated vertical program transformation ideas is the PSI system [10]. It consists of two modules, the PECOS module [4] which generates the program based on a library of synthesis refinement rules and the LIBRA [11] system which controls the search in the generation of the program. Researchers at the Kestrel Institute and the Information Sciences Institute, University of Southern California are pursuing this approach to implement high level specification languages.

The vertical transformation approach is similar to our approach; the difference lies mainly in emphasis. Most program transformation systems start with more declarative and more abstract specifications than those accepted by the DA, and through a long series of transformations, arrive at an implementation. Some design steps in the DA can also be viewed as vertical transformations. However, in addition to allowing vertical transformations, the DA also maintains explicit design dependencies for the implementations chosen.

5.4 Very High Level Languages

This approach aims to provide a high level language in which abstract specifications and concrete implementations can be written in the same language. Typically, a transformation system is built to transform the abstract specifications into executable constructs.

The SETL language [29] is a set-based, tuple-based language specifically designed to allow the programmer to ignore the detailed design of data structures. Kestrel Institute's REFINE language is another example of a very high level language [1]. REFINE is a general-purpose, set-based, logic-based, wide-spectrum language with facilities for manipulating a knowledge base of transformation rules. The REFINE compiler is a set of transformation rules that turns a REFINE program into executable Common Lisp code. The GIST specification language [8] is designed to retain the expressiveness of natural language. An interpreter has been built to study some transformation rules in this context.

The DA can be viewed as supporting a specialized, very high level specification language. Like all systems which follow this approach, the DA aims to provide the programmer with a language which is higher level than that provided by conventional programming languages, and strives to free the programmer from detailed design of data structures. However, most of these systems expect the user to provide complete specifications. In contrast, the DA is expected to help the user with completing some partial description. In addition, the input language of the DA is an interactive design language intended to capture some of the intermediate vocabulary used

by programmers in order to make the program description more understandable to human programmers.

5.5 Program Generators

A program generator uses a high level specification language typically designed for a specific domain, and turns a description in such a language into a program in some programming language. Program generators are similar to the very high level languages except that they typically use more traditional compilation techniques, and their focus is typically very narrow.

The Draco approach to automatic programming [22] can be viewed as advocating program generators for specific domains. Suitable domains are domains in which many systems need to be built. A domain analyst studies such a domain and creates the objects and operations of interest to the domain, possibly in terms of other domain objects and operations known to the Draco system. A domain designer creates alternative implementations for each object and operation. The resultant Draco system is then used by system analysts and system designers. A system analyst specifies a new system in the domain using the objects and operations provided by the Draco system; a system designer takes this specification and creates an executable system by selecting the appropriate implementations for each object and operation in the specification using the Draco system.

Our approach, when appropriately viewed, is similar to that of Draco's. The clichés we are trying to codify are similar to the software components the Draco approach calls for. The domain analysis and design is similar to our analysis of some programming domains in order to come up with appropriate clichés. One way of viewing our work is an attempt to automate the process of detailed design in Draco: Draco expects a system designer to select implementations for objects and operations in the specifications, the DA is expected to choose implementations automatically. Besides automatic selection of implementations, the DA also maintains the design rationale for the chosen implementations and critiques the given specification. The framework established by the DA is also broad enough to capitalize on program generators. In the scenario, Tokenize and Concatenate are clichés represented as program generators.

Another domain-specific program generator is the Φnix system [5]. It applies an extensive body of oil well logging knowledge to synthesize analysis programs. The Φnix approach and our approach shares a common emphasis on domain-specific knowledge if we view programs manipulating graphs as a domain. Taking such an emphasis allows the users of our systems to specify their needs concisely and in terms natural to them.

5.6 Algorithm Design

Another dimension to automatic programming is algorithm design. Kant [12] defined algorithm design as *the process of coming up with a sketch, in a very high level language, of a computationally feasible technique for accomplishing a specified behavior.* Kant and Steier at CMU [31] studied human algorithm designers at work. Steier and Newell's work on DESIGNER-SOAR [32] investigated using production system architecture to design and discover algorithms. DESIGNER-SOAR explores algorithm design by successive refinement using symbolic execution and test-case execution as primary control mechanisms. It integrated nine sources of knowledge: weak methods, design strategies, program transformation, efficiency, symbolic execution, target language, domain definition, domain equations and learning.

The scope of these works included original algorithm design and discovery. In contrast, our work aims at re-using commonly known data structures and algorithms.

5.7 Selection of Data Structures and Algorithms

Low's system [18] automatically selects data structures based on how the data are used, the domains and sizes of data, and the operations performed on them. Techniques used include static flow analysis, monitoring execution of sample runs, and user interaction. The system concentrated on selection of data structures for sets and lists. Our work draws on the experience of Low's system, and extends the scope to include the selection of algorithms.

Rowe and Tonge [28] described a class of abstract data structures, called modeling structures, in terms of properties involving their component elements, relations between elements, and operations on them. Each modeling structure may have several implementation structures which implement it. They described a system that accepts specifications which contain modeling structures and turns the modeling structures into implementation structures with the help of a library of mappings of modeling structures to implementation structures. They also described an algorithm which can implement the remaining modeling structures that failed to match any of the mappings available in the library. They acknowledged that their system is difficult to use partly because the descriptions of modeling structures were complicated. The question of how to combine several modeling structures into one representation is left open. Our work also makes use of descriptions of familiar modeling structures to help select implementations of data structures. The descriptions in the DA, however, are designed to be easily understandable to programmers.

Katz and Zimmerman [13] built an interactive advisory system for choosing data structures. It was able to provide new combinations of known structures which satisfy complex requirements not anticipated in advance. The key idea embodied in the system is that it provides a linguistic base of known vocabulary about data structures in which the selection of implementation structures was carried out. This is also

the same idea being exploited in our approach by providing a familiar vocabulary for describing more abstract data structures and algorithms, and for describing the selection criteria.

McCartney's MEDUSA system [20] synthesizes a number of functional geometric algorithms based on the given input-output specifications and performance constraints. MEDUSA achieves this using a library of templates and a small number of design methods. Its emphasis lies in making the algorithm synthesis process efficient and fully automatic. It also assumes complete and consistent specifications. In contrast, the DA can detect some incomplete and inconsistent specifications. It is meant to be an interactive tool, and it supports incremental design changes made by the user.

Chapter 6

Future Work and Conclusions

This report describes specific progress made in constructing the DA which embodies our approach to supporting reuse and evolution in program design. This work can also be viewed as charting out, in significant detail, the key issues involved in the construction of the DA. In the following we describe what needs to be done next.

Implementation: Completing the partial implementation that currently exists is useful to help debug the ideas presented in this thesis. This will serve to demonstrate the feasibility of the DA. A key missing component in the current work is a more refined control structure that uses the various design heuristics given to automatically select the design steps.

Program Metrics: To make effective tradeoff decisions, a formal language is needed to characterize the efficiency of programs and incomplete programs based on the distribution of their input data. Algorithmic clichés should have annotations about their performance properties so that when they are combined or modified, the performance properties of the resultant design can be estimated from its constituent parts. Some work in the framework of deriving and proving the computational properties of while-programs has been done [21, 23]. It will be interesting to adapt and extend them for use in our current framework.

Adding New Capabilities: The DA may be able to add auxiliary code into a design that performs standard tasks that programmers frequently do. For instance, to help in debugging and simulation, code can be inserted into a program to trace the execution of the program. Instrumentation code can be added to collect statistics about program runs. In a system that does explicit storage management, code can be added to do garbage collection. It will also be useful if the DA can specify domain-level constraints that are automatically translated into specific checks in the program. The input description accepted by the DA is currently rather operational and formal. It would be useful to relax that requirement by adding a module that can help acquire the program description from less formal input descriptions. Some work, such as [24], has been done in the context of requirements acquisition. Similar techniques might be applicable here to relax the rigor of the input description.

Another important capability the DA should eventually have is an explanation generation system. Currently only the dependencies of various design decisions are recorded. CAKE makes a distinction between assertions that are retractable, *premises*, and those that are non-retractable, *axioms*. CAKE compiles axioms into constraints in the system to optimize run-time speed. In the process, the domain model represented by these axioms is not easily accessible. Even if the domain model is represented as retractable premises in CAKE, generating good explanations is a non-trivial task on its own. Some work [16] has been done to remove clutter in a dependency tree but the underlying deeper problem about generating appropriate explanations has not been addressed.

Extensions: The Programmer's Apprentice is intended to support the entire process of software development. The DA starts at one end of the spectrum, looking for ways to formalize higher level program descriptions into executable code. Another project, the Requirements Apprentice (RA) [24], is concerned with the other end of the spectrum – automating the acquisition of informal requirement descriptions. The gap between the Requirements Apprentice and the DA remains to be bridged. This may be done by extending the capability of the DA to include high-level design of programs, or by a separate module that starts with the requirement descriptions produced by the RA, and produces a high-level design that is acceptable to the DA.

Another direction to pursue is to study the designs of programs with complex side-effects in the DA framework. The Plan Calculus formalism has facilities for reasoning about side-effects. We have, however, focused our attention on programs which are essentially side-effect free.

Codification of Programming Knowledge: Given that complete reasoners are computationally intractable, incomplete reasoners are here to stay. Our approach in the use of incomplete reasoners has been to add domain lemmas (assertions that can be derived from the domain axioms) so that CAKE, an incomplete reasoner, can succeed in deriving the desired conclusions. This has meant a strong reliance on the contents of the cliché library. There are few general operational guidelines about how to make the tradeoffs between "reasoning harder" and "encoding more domain lemmas".

A related problem in using the Plan Calculus is the lack of a well-informed discipline towards this task of codifying programming knowledge. This problem has to be informed strongly by the nature of the uses the knowledge will be put to, and the capabilities of the reasoner that might act on them.

Another dimension to pursue in knowledge codification is in increasing the depth of the cliché library. Besides codifying more kinds of general algorithmic clichés, a richer vocabulary of design steps is needed to describe the detailed design of programs.

6.1 Conclusions

We have described the progress made in establishing a framework in which reuse and evolution in software design can be achieved. This framework is supported by a proposed and partially implemented tool, the DA, which supports software reuse by a library of clichés and a suite of design steps suitable for automating the detailed design of programs. The DA supports software evolution by maintaining dependencies among design decisions. The DA is also able detect some kinds of inconsistencies and incompleteness in the input descriptions.

We have also described a novel process model, programming by successive elaboration, that underlies the capabilities of the DA. This process is characterized by the use of breadth-first exposition of layered program descriptions and the successive modifications of descriptions. It allows a programmer to describe a desired program in a concise and comprehensible manner using clichés. We argue that this process is useful, familiar, and natural.

Bibliography

[1] L. M. Abráido-Fandiño. An overview of $refine^{TM}$ 2.0. A revised edition of the paper published in Proc. 2nd Int. Symposium on Knowledge Engineering – Soft. Eng., Madrid, Spain, April, 1987.

[2] A. V. Aho, J. D. Ullman, and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[3] A. V. Aho, J. D. Ullman, and J. E. Hopcroft. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[4] D. R. Barstow. An experiment in knowledge-based automatic programming. *Artificial Intelligence*, 12(1 & 2):73–119, 1979. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[5] D. R. Barstow. A perspective on automatic programming. *AI Magazine*, 5(1):5–27, spring 1984. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[6] R. M. Burstall and J. L. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1), January 1977.

[7] C.E. Cormen, T.H. Leiserson and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[8] M. S. Feather and P. E. London. Implementing specification freedoms. *Science of Computer Programming*, 2:91–131, 1982.

[9] Y. A. Feldman and C. Rich. Bread, Frappe, and Cake: The gourmet's guide to automated deduction. In *Proc. 5th Israeli Symp. on Artificial Intelligence*, Tel Aviv, Israel, December 1988.

[10] C. Green. A summary of the PSI program synthesis system. In *Proc. 5th Int. Joint Conf. Artificial Intelligence*, pages 380–381, Cambridge, MA, August 1977.

[11] E. Kant. A knowledge-based approach to using efficiency estimation in program synthesis. In *Proc. 6th Int. Joint Conf. Artificial Intelligence*, pages 457–462, Tokyo, Japan, August 1979. Vol. 1.

[12] E. Kant. Understanding and automating algorithm design. In *Proc. 9th Int. Joint Conf. Artificial Intelligence*, pages 1243–1253, 1985.

[13] S. Katz and R. Zimmerman. An advisory system for developing data representations. In *Proc. 7th Int. Joint Conf. Artificial Intelligence*, pages 1030–1036, Vancouver, British Columbia, Canada, August 1981. Vol. 2.

[14] D. E. Knuth. *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 1968, 1969, 1973. Volumes 1, 2, and 3.

[15] D. E. Knuth and M. P. Plass. Breaking paragraphs into lines. *Software – Practice and Experience*, 11:1119–1184, 1981.

[16] P. M. Lefelhocz. An experiment in knowledge acquisition for software requirements. Working Paper 330, MIT Artificial Intelligence Lab., May 1990.

[17] M.E. Lesk and E. Schmidt. Lex: A lexical analyzer generator. In *Unix Programmer's Manual, B.W. Kernighan and M.D. McIlroy, Bell Labs, 7th Edition*, 1978.

[18] J. R. Low. Automatic data structure selection: An example and overview. *Comm. ACM*, 21(5):376–384, May 1978.

[19] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Programming Languages and Systems*, 2(1):90–121, January 1980. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[20] R. McCartney. Synthesizing algorithms with performance techniques. In *Proc. 6th National Conf. on Artificial Intelligence*, pages 149–154, Seattle, WN, July 1987.

[21] Daniel Le Métayer. Ace: An automatic complexity evaluator. *ACM Trans. Programming Languages and Systems*, 10(2):248–266, April 1988.

[22] J. M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Trans. Software Engineering*, 10(5):564–574, September 1984. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[23] Hanne Riis Nielson. A hoare-like proof system for analysing the computation time of programs. *Science of Computer Programming*, 9(2):107–136, October 1987.

[24] H. B. Reubenstein. Automated acquisition of evolving informal descriptions. Technical Report 1205, MIT Artificial Intelligence Lab., June 1990.

[25] C. Rich. A formal representation for plans in the Programmer's Apprentice. In *Proc. 7th Int. Joint Conf. Artificial Intelligence*, pages 1044–1052, Vancouver, British Columbia, Canada, August 1981. Reprinted in M. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modelling*, Springer Verlag, 1984 and in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[26] C. Rich. The layered architecture of a system for reasoning about programs. In *Proc. 9th Int. Joint Conf. Artificial Intelligence*, pages 540–546, Los Angeles, CA, 1985.

[27] C. Rich and R. C. Waters. *The Programmer's Apprentice.* Addison-Wesley, Reading, MA and ACM Press, Baltimore, MD, 1990.

[28] L. A. Rowe and F. M. Tonge. Automating the selection of implementation structures. *IEEE Trans. Software Engineering*, 4(6):494–506, November 1978. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[29] E. Schonberg, J. T. Schwartz, and M. Sharir. An automatic technique for selection of data representation in SETL programs. *ACM Trans. Programming Languages and Systems*, 3(2):126–143, April 1981. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[30] D. R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, 1985. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[31] D. M. Steier and E. Kant. Symbolic execution in algorithm design. In *Proc. 9th Int. Joint Conf. Artificial Intelligence*, pages 225–231, 1985.

[32] D. M. Steier and A. Newell. Integrating multiple sources of knowledge into designer-soar, an automatic algorithm designer. In *Proc. 7th National Conf. on Artificial Intelligence*, pages 8–13, 1988.

[33] Y. M. Tan. ACE: A cliché-based program structure editor. Working Paper 294, MIT Artificial Intelligence Lab., May 1987.

[34] R. C. Waters. The Programmer's Apprentice: A session with KBEmacs. *IEEE Trans. Software Engineering*, 11(11):1296–1320, November 1985. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and*

Software Engineering, Morgan Kaufmann, 1986 and in T. Ichekawa, editor, *Language Architectures and Programming Environments*, MIT Press, in preparation.

[35] R. C. Waters. Series. In G.L. Steele Jr., editor, *Common Lisp the Language, Second Edition.* Digital Press, Burlington, MA, 1990.