the supporting and surrounding structure for interference checking. These piping design systems have achieved a high degree of integration.

An example of the current state of the art in integrated structural design is the Intergraph RandMicas design system (IRM) [Intergraph87]. The heart of this system is the Common Structural Database (CSD) [Intergraph86] which allows common structural data to be passed among several application programs as illustrated in figure 2-1. The common database can be used for description of both steel and concrete structures. Steel members are defined by an orientation vector, a section and the position of the section relative to the vector. Each section has a "notional box", a minimal size rectangle which encloses the section. Within this box nine "cardinal points" are defined—these occur at the corners of the rectangle and at its center. The section can be positioned relative to the orientation vector at any of these points. The database accesses a separate table of sections which includes the standard steel sections and which may be expanded to include custom built-up sections defined by the user. The data record for each element also records end connectivity, end loads, and member end cut-backs.

The description of concrete members is basically similar defining both the geometry of the concrete member and the location of reinforcing bars. Bars may be defined according to standard type or as custom shapes.

The common database serves primarily as a means of transferring descriptive data between applications and does not replace databases associated directly with those applications which may contain data more directly pertaining to the application.

The Intergraph system is an exemplar of the fundamental concepts of integrated computer-aided design systems. The system is fashioned around a central database that stores building description and design data. The core system provides a consistent user interface, graphical interaction, design visualization (both 2D and 3D), data management, and document preparation. Other applications may be added to this core system.

Taskmaster [Grimble83] is a commercial system that makes use of a central independent database and application programs relating to structural analysis and design. The database representation includes node data, member incidences, member loads and member forces.

---

10 "A complete 3D model of the pipework is of little use without the other elements of the model (i.e., steelwork for clash detection purposes)" [Button84].
NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilming. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, S.R.C. 1970, c. C-30, et ses amendements subséquents.
A System for Describing Design Artefacts Using the Knowledge Representation Technique of Frames

by

Alan Glass, P.Eng.
B.Eng., Concordia University, Montreal
M.Eng., Carleton University, Ottawa

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of the
requirements for the degree of

Doctor of Philosophy

Department of Civil Engineering

Carleton University
Ottawa, Canada
September, 1989
© copyright 1989, Alan Glass
The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of this thesis,

A System for Describing Design Artefacts
Using the Knowledge Representation
Technique of Frames

submitted by

Alan Glass

in partial fulfilment of the requirements
for the degree of Doctor of Philosophy

Chair, Department of Civil Engineering

Thesis Supervisor

External Examiner

Carleton University
Abstract

This thesis presents a system for describing design artefacts. Description is based on a knowledge base of explicitly defined abstractions called "forms." An artefact, and its components, are described as instances of these forms. A form is a relation of data items representing the attributes of the abstraction, and incorporating associated data and procedural information. This associated information constitutes knowledge of the description domain and allows implementation of various useful facilities including the computation of design values, integrity maintenance, value constraints, default values and explanation. Description through the instantiation of abstractions provides a concise and meaningful medium of communication between the description system and the user. As well, this style of description suits the design process which evolves a description through the resolution of abstract types. The system is implemented using the knowledge representation technique of "frames." The system is based on a computing strategy in which an application-independent problem-solving program is provided with an explicit representation of knowledge pertaining to a particular problem domain. The advantage of this strategy is that, because the knowledge is represented explicitly, the scope of the problem domain can be incrementally modified and expanded. An application to the description of a simple class of building structures is demonstrated.

Keywords: building description, civil engineering design, computer-aided design, data model, description system, frames, knowledge representation, structural modelling.
Acknowledgements

Thanks are due to my thesis supervisor, Neal Holtz, for his always welcomed advice and encouragement; to Don Duchesne of Adjeleian Allen Rubeli Ltd., Ottawa, for kindly supplying the structural drawings which were used as the basis for one of the example descriptions; and to the CASCADE (Center for Advance Studies in Computer-aided Design and Engineering) organization for the use of their computing facilities. This work was partially supported by NSERC Operating Grant A1419.
## Table of Contents

Abstract .................................................. iii
Acknowledgements ........................................ iv
Table of Contents .......................................... v
List of Figures ............................................. viii
Quotation ................................................ x

1 Introduction ............................................... 1
   1.1 Scope and limitations. .......................... 5
   1.2 Organization .................................. 6

2 Literature Review ......................................... 8
   2.1 Integrated design ............................... 8
   2.2 Computer-aided description ................. 9
   2.3 Databases ................................ 11
   2.4 Geometric modelling ........................ 14
   2.5 Expert systems .............................. 14
   2.6 Frames .................................. 18

3 Abstraction and Knowledge .............................. 24
   3.1 Abstraction ................................ 24
   3.2 Generalization ................................ 26
   3.3 Forms ................................... 27
   3.4 Specialization ................................ 28
   3.5 The AKO hierarchy .......................... 29
   3.6 Aggregation .................................. 3:
   3.7 Summary .................................... 32

4 Forms and Descriptions ................................ 34
   4.1 A simple implementation .................... 34
   4.2 Extensions and modifications ............... 38
      4.2.1 Forms and instances .................. 41
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.2</td>
<td>Associations</td>
<td>42</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Facets</td>
<td>43</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Inheritance</td>
<td>44</td>
</tr>
<tr>
<td>4.2.5</td>
<td>System modularity</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>Description using frames</td>
<td>48</td>
</tr>
<tr>
<td>4.4</td>
<td>Summary</td>
<td>51</td>
</tr>
<tr>
<td>5</td>
<td>Values and Methods</td>
<td>52</td>
</tr>
<tr>
<td>5.1</td>
<td>Evaluation modes</td>
<td>53</td>
</tr>
<tr>
<td>5.2</td>
<td>Slot evaluation</td>
<td>56</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Forward chaining</td>
<td>57</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Backward chaining</td>
<td>58</td>
</tr>
<tr>
<td>5.4</td>
<td>Summary</td>
<td>62</td>
</tr>
<tr>
<td>6</td>
<td>Integrity Maintenance</td>
<td>63</td>
</tr>
<tr>
<td>6.1</td>
<td>Data dependency</td>
<td>63</td>
</tr>
<tr>
<td>6.2</td>
<td>Dependency links</td>
<td>66</td>
</tr>
<tr>
<td>6.3</td>
<td>Reduction</td>
<td>68</td>
</tr>
<tr>
<td>6.4</td>
<td>Change propagation</td>
<td>69</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Bulletin board</td>
<td>70</td>
</tr>
<tr>
<td>6.5</td>
<td>Summary</td>
<td>72</td>
</tr>
<tr>
<td>7</td>
<td>Defaults</td>
<td>73</td>
</tr>
<tr>
<td>7.1</td>
<td>Soft defaults</td>
<td>73</td>
</tr>
<tr>
<td>7.2</td>
<td>Integrity maintenance</td>
<td>78</td>
</tr>
<tr>
<td>7.3</td>
<td>Summary</td>
<td>79</td>
</tr>
<tr>
<td>8</td>
<td>User interface</td>
<td>81</td>
</tr>
<tr>
<td>8.1</td>
<td>Forms and instances</td>
<td>82</td>
</tr>
<tr>
<td>8.2</td>
<td>Menus</td>
<td>83</td>
</tr>
<tr>
<td>8.3</td>
<td>Summary</td>
<td>85</td>
</tr>
<tr>
<td>9</td>
<td>Demonstration</td>
<td>86</td>
</tr>
<tr>
<td>9.1</td>
<td>Forms</td>
<td>86</td>
</tr>
<tr>
<td>9.2</td>
<td>Instances</td>
<td>90</td>
</tr>
<tr>
<td>9.3</td>
<td>Specialization</td>
<td>91</td>
</tr>
<tr>
<td>9.4</td>
<td>Aggregation</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table of Contents

9.5 External applications ............................................. 96
  9.5.1 Constructive solid geometry ................................. 97
  9.5.2 Constructive plane geometry ............................... 99
  9.5.3 Example ..................................................... 100

9.6 Integrity Maintenance ............................................... 104
  9.6.1 Semantic links ............................................... 104
  9.6.2 Functional links ............................................. 106
  9.6.3 Change propagation ......................................... 109

9.7 Reduction .......................................................... 113

9.8 Defaults ........................................................... 118
  9.8.1 Default integrity ............................................. 121

9.9 User Interface ...................................................... 124
  9.9.1 Menu specialization .......................................... 126

10 Building Description ................................................. 128
  10.1 Assembly and Connectivity ..................................... 128
    10.1.1 Knowledge base ........................................... 129
    10.1.2 Description ............................................... 133
    10.1.3 Summary ................................................... 141

10.2 Higher level abstractions ....................................... 142
    10.2.1 Knowledge base ........................................... 142
    10.2.2 Description ............................................... 145
    10.2.3 Summary ................................................... 154

11 Conclusion ........................................................... 155
  11.1 Summary ........................................................ 155
  11.2 Recommendations ............................................... 158
  11.3 Conclusions .................................................... 163

Appendices
  A Knowledge base KB0 .............................................. 166
  B Knowledge base KB1 .............................................. 186
  C Knowledge base KB3 .............................................. 195
  D A note on the program code and computing resources ........... 202

Glossary ............................................................... 203

Bibliography ........................................................... 207

vii
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>The design process</td>
<td>2</td>
</tr>
<tr>
<td>2-1</td>
<td>The Common Structural Database</td>
<td>11</td>
</tr>
<tr>
<td>2-2</td>
<td>A logic programming system</td>
<td>16</td>
</tr>
<tr>
<td>3-1</td>
<td>The family of quadrilaterals</td>
<td>30</td>
</tr>
<tr>
<td>3-2</td>
<td>The parameters of quadrilateral, trapezoid and rectangle</td>
<td>31</td>
</tr>
<tr>
<td>3-3</td>
<td>Some 2-D shapes created by aggregation</td>
<td>32</td>
</tr>
<tr>
<td>4-1</td>
<td>A frame as a 2-tier association list</td>
<td>35</td>
</tr>
<tr>
<td>4-2</td>
<td>The layout of a simple frame</td>
<td>36</td>
</tr>
<tr>
<td>4-3</td>
<td>The layout of the form</td>
<td>41</td>
</tr>
<tr>
<td>4-4</td>
<td>The layout of the instance</td>
<td>42</td>
</tr>
<tr>
<td>4-5</td>
<td>A description of a wide-flange component</td>
<td>49</td>
</tr>
<tr>
<td>4-6</td>
<td>A description of a wide-flange component with aggregation</td>
<td>50</td>
</tr>
<tr>
<td>5-1</td>
<td>Hierarchical relation of Rectangle and local form Center</td>
<td>61</td>
</tr>
<tr>
<td>6-1</td>
<td>An input and output file linked by a program</td>
<td>64</td>
</tr>
<tr>
<td>6-2</td>
<td>A system of linked data files and programs</td>
<td>64</td>
</tr>
<tr>
<td>6-3</td>
<td>A central database system</td>
<td>65</td>
</tr>
<tr>
<td>9-1</td>
<td>The parameters of the family of Quadrilaterals</td>
<td>87</td>
</tr>
<tr>
<td>9-2</td>
<td>The parameters of the form Wshape</td>
<td>94</td>
</tr>
<tr>
<td>9-3</td>
<td>Three PADL-2 primitives</td>
<td>97</td>
</tr>
<tr>
<td>9-4</td>
<td>A constructive solid geometry tree</td>
<td>98</td>
</tr>
<tr>
<td>9-5</td>
<td>Two-dimensional constructive geometry primitives</td>
<td>99</td>
</tr>
<tr>
<td>9-6</td>
<td>Triangle built up from primitive right triangles</td>
<td>100</td>
</tr>
<tr>
<td>9-7</td>
<td>The wireframe display of a quadrilateral</td>
<td>102</td>
</tr>
<tr>
<td>9-8</td>
<td>Wireframe image of Wshape beam</td>
<td>103</td>
</tr>
<tr>
<td>9-9</td>
<td>Semantic links among forms and instances</td>
<td>105</td>
</tr>
<tr>
<td>9-10</td>
<td>Integrity links in an instance of Component</td>
<td>107</td>
</tr>
<tr>
<td>9-11</td>
<td>Integrity links with a named instance as slot value</td>
<td>110</td>
</tr>
<tr>
<td>9-12</td>
<td>Integrity links in an instance of Square</td>
<td>114</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6-13</td>
<td>Integrity links in an instance of Rectangle</td>
<td>116</td>
</tr>
<tr>
<td>9-14</td>
<td>Integrity links in an instance of Parallelogram</td>
<td>117</td>
</tr>
<tr>
<td>10-1</td>
<td>Floor plans of two-storey structure</td>
<td>134</td>
</tr>
<tr>
<td>10-2</td>
<td>PADL-2 image of two-storey structure</td>
<td>141</td>
</tr>
<tr>
<td>10-3</td>
<td>Plan of a single-storey steel-framed structure</td>
<td>146</td>
</tr>
<tr>
<td>10-4</td>
<td>Plan of regular structure</td>
<td>148</td>
</tr>
<tr>
<td>10-5</td>
<td>PADL-2 image of regular structure</td>
<td>148</td>
</tr>
<tr>
<td>10-6</td>
<td>Plan of reduced and updated structure</td>
<td>150</td>
</tr>
<tr>
<td>10-7</td>
<td>PADL-2 image of reduced and updated structure</td>
<td>150</td>
</tr>
<tr>
<td>10-8</td>
<td>Plan of further updated structure</td>
<td>152</td>
</tr>
<tr>
<td>10-9</td>
<td>PADL-2 image of further updated structure</td>
<td>152</td>
</tr>
<tr>
<td>10-10</td>
<td>Plan of final structure</td>
<td>153</td>
</tr>
<tr>
<td>10-11</td>
<td>PADL-2 image of final structure</td>
<td>153</td>
</tr>
</tbody>
</table>
Quotation

For everything that exists there are three classes of objects through which knowledge must come. The knowledge is a fourth, and we must put as a fifth entity the actual object of knowledge which is the true reality. We have then, first a name, second a description, third an image, and fourth knowledge of the object.

Take a particular case if you want to understand the meaning of what I have just said: then apply the theory to every object in the same way. There is something for instance called a circle, the name of which is the very word I just now uttered. In the second place there is description of it which is composed of nouns and verbal expressions. For example the description of that which is named round and circumference and circle would run as follows: the thing which has everywhere equal distances between its extremities and its center. In the third place there is the class of object which is drawn and erased and turned on the lathe and destroyed—processes which do not affect the real circle to which these other circles are all related, because it is different from them. In the fourth place there are knowledge and understanding and correct opinion concerning them, all of which we must set down as one thing more that is found not in sounds nor in shapes of bodies, but in minds, whereby it evidently differs in its nature from the real circle and from the aforementioned three. Of all these four, understanding approaches nearest in affinity and likeness to the fifth entity, while the others are more remote from it.

The same doctrine holds good in relation to shapes and surfaces, both straight and curved, in regard to the good and the beautiful and the just, in regard to all bodies artificial and natural, in regard to fire and water and the like, and in regard to every animal, and in regard to every quality of character, and in respect to all states active and passive. For if in the case of any of these a man does not somehow or other get hold of the first four, he will never gain a complete understanding of the fifth. Furthermore, these four [names, descriptions, bodily forms, concepts] do as much to illustrate the particular quality of any object as they do to illustrate its essential reality because of the inadequacy of language.

—Plato (Epis. 7.342b)
Chapter 1

Introduction

This thesis presents a system for describing design artefacts. Description is based on a knowledge base of explicitly defined abstractions called "forms." An artefact, and its components, are described as instances of these forms. A form is a relation of data items representing the attributes of the abstraction, and incorporating associated data and procedural information. This associated information constitutes knowledge of the description domain and allows implementation of various useful facilities including the computation of design values, integrity maintenance, value constraints, default values and explanation. Description through the instantiation of abstractions provides a concise and meaningful medium of communication between the description system and the user. As well, this style of description suits the design process which evolves a description through the resolution of abstract types. The system is implemented using the knowledge representation technique of "frames." The system is based on a computing strategy in which an application-independent problem-solving program is provided with an explicit representation of knowledge pertaining to a particular problem domain. The advantage of this strategy is that, because the knowledge is represented explicitly, the scope of the problem domain can be incrementally modified and expanded.

The description of the design artefact has a vital role in the design process. Figure 1-1(a) presents one model of the design process. Because engineered artefacts are functional, the design process starts with a more or less clearly defined specification of the performance expected of the artefact. In addition to these explicit requirements, the design must also satisfy a broad range of implicit or inferred requirements. The designer develops a description with a view to satisfying these design criteria. The fabricator then fabricates the artefact as directed in accordance with the description. The design process is successful if the observed performance of the design artefact satisfies the performance criteria.

In this model the description acts primarily as a means of communication between the designer and the fabricator. But this communication is only a minor part of the role of description.
Figure 1-1: The design process.

Engineering is the science of predicting the performance of a physical artefact based on its description. The design engineer does not depend on observed performance to evaluate the design artefact, but calls on specialized scientific knowledge of mechanics and material behaviour to accurately evaluate the proposed artefact from its description. The design process therefore is not usually considered to include the fabrication of the artefact. The end-product of the design process is not the artefact but its description only.

The part of the design process which directly involves the engineer itself comprises two separate procedures—these are illustrated in figure 1-1(b). The first, synthesis, is the creation of a description of a design artefact which potentially satisfies the performance criteria. The second, analysis, computes the performance of the described artefact. Computed performance thus replaces observed performance. If the results of the analysis determine
that the proposed solution does not satisfy all performance criteria, the results can be used in a second round of synthesis. Design is thus an iterative process, evolving through successive cycles of synthesis and analysis.

Description being central to the design process, deserves serious attention. In particular, any attempt to implement computer aids to the design process must resolve the problem of representing the design artefact in a manner that is suited both to computer manipulation and to the design engineer. The representation must provide a wide variety of data types and permit non-standard design solutions. The disadvantage of the usual database approach to design description is that the abstractions in terms of which the designer may describe the design artefact are implicit in the data model. For example, if the data model views a building structure as consisting of vertical columns and horizontal beams, the designer is constrained to design solutions comprising only vertical and horizontal elements (as is so often the case, the solution of the design problem is constrained by the ability to describe it or analyse it). Application of the knowledge representation technique of frames to design description avoids this disadvantage by providing for the explicit definition of the abstractions used in the description.

"Frames" is one of several techniques that have been developed for representing knowledge. The most familiar of these techniques—that used in the typical Expert System—represents knowledge as rules of inference; these rules are processed by an independent "inference engine." The frames technique, in contrast, expresses knowledge as an aspect of the representation of an object. With this technique, knowledge in the domain of description is associated with the typical objects of description. Knowledge consists of data and procedures which are attached to each object. An application-independent program acts on this knowledge to aid the user in creating and interpreting descriptions.

The frames approach provides an important model for representing design artefacts and the knowledge associated with them. In general, the design process is not standardized. Each design problem is unique and leads to a unique solution. For any design problem a large number of solutions may satisfy the design criteria. For example, a set of criteria for a building structure may be satisfied by a design description calling for the structure to be fabricated of steel, reinforced concrete, timber or a mixture of these and other construction materials. As well, the design may employ one of a number of structural systems. Besides these major design options, a wide range of options is available in the design of structure
sub-systems and details. Engineering design deals with a wide variety of data types and their relationships.

The frame-based description system can respond to the need for representing a wide range of data types in the description of design artefacts. Description is in terms of explicitly defined classes of design objects. The definitions of these classes form a "knowledge base." Because knowledge is factored in classes, any single design description may use only a selection from the large number of classes available. Because knowledge—being associated with specific classes—is modular, the knowledge base can build on existing classes. For example, knowledge associated with the class of compression members can be used both in the design of truss members and in the design of building columns. Design knowledge can furthermore be associated with the appropriate level of detail. For example, the class of concrete beams may rely on the more detailed class of concrete beam cross-sections to establish the varying section properties along the beam. In this way it becomes possible to modify or augment specific classes independent of the classes which may interact with them. For example, the class of cross-sections can be revised independently of the class of beams.

The description system comprises three levels of program abstraction: the system shell, the knowledge base and the description. The system shell is an application-independent program which, in effect, defines a language for the creation of knowledge bases and descriptions. The knowledge base defines the typical objects of description, thus defining the domain of description. The description is made up of instances of the typical objects defined in the knowledge base. Each of these three levels—the knowledge base, the description and the shell—is separate.

The frames approach to design description has many advantages over the database approach:

- The chief of these advantages is greater ease of expansion and update. This advantage is the result of the object-oriented representation of knowledge. The data and the processes that generate and maintain the data are integrated at a very detailed level; this arrangement makes the relation of data and process explicit. The encapsulating of data and related processes within a self-contained data structure makes the system highly modular.
• Because the typical objects of description are explicitly defined in an independent knowledge base, the designer can expand or customize the knowledge base to suit a given project. The typical objects can be highly specialized. The description system allows modification of instances of these special types to permit the introduction of non-standard features.

• User interaction with the description is data-directed. The user interacts directly with the data without the need to invoke processes. The process is subservient to the data and is invoked by the system, unseen by the user, as needed to satisfy commands directed at the data items. Because interaction is in terms of the data items, the processes can be updated without affecting the interaction.

• The description system is able to represent many different aspects of knowledge relating to the typical objects of description. The generation of design values is one important aspect. Others include integrity maintenance, value constraints, default values, and explanation. Each of these aspects is explicitly represented.

• The description system supports both internal and external applications. Applications that are data-intensive can be processed efficiently within the system. As well the system can interface with external programs which may more efficiently process computation-intensive applications.

• Currently the system takes advantage of the wide general computing powers of the host language, which includes logic and symbolic computing.

1.1 Scope and limitations.

The objective of this thesis is to demonstrate the frames model as a suitable medium for representing design artefacts. This work focuses on the development of a representation suitable for describing engineering design artefacts and the investigation of how various aspects of engineering design knowledge can be implemented in this scheme of representation.

This work comprises three separate but related tasks: development of a shell program for the description of design artefacts; development of an example knowledge base defining a particular domain of description, in this case, the description of a class of building structure; and presentation of an example description using this knowledge base.
Chapter 1: Introduction

The frames concept has been implemented in many forms and is the basis of some successful high-level programming languages. The system presented here modifies the basic frames model to better meet the specific needs of describing artefacts for design. In particular, the frame system implemented includes data-directed computations for value generation, the dynamic maintenance of data integrity, value constraints and a system of default values which are generated and maintained separately from input values.

The knowledge base presented will describe a simple class of building structure and include include the computation of geometric properties of the structure and its components, the computation of gravity loads, and an interface to a solids modelling program.

1.2 Organization

The literature review of Chapter 2 serves as an introduction to the various concepts that contribute to the thesis topic. Special attention is given to computing techniques borrowed from the realm of Artificial Intelligence, specifically, rule-based expert systems and the knowledge representation technique of frames.

Chapter 3 looks at the concrete representation of abstractions as data structures. An abstraction is represented as a Platonic “form,” the list of the essential attributes of the class of objects to which the abstraction applies. Attributes are considered to be of two kinds: parameters and properties. Parameters are those attributes that suffice to identify a unique instance of a class. Properties are those attributes that can be derived from the values of the class parameters. Sub-classes of forms are created through specialization, a process which defines a sub-class as a kind of the super-class with certain parameters constrained. Classes and subclasses form a specialization hierarchy.

Chapter 4 introduces the implementation of forms and instances using the knowledge representation technique of frames. The implementation is an extension and modification of a simpler implementation based on the frames language FRL. Descriptions take the shape of a hierarchy of nested instances.

The manner of computing attribute values is discussed in Chapter 5. There are two modes of evaluation. Parameter values are evaluated in forward chaining mode. Property values are evaluated in backward chaining mode. In backward chaining, a query for an
attribute value causes its evaluation. Evaluation usually results in the internal generation of further queries.

Chapter 6 discusses the implementation of the integrity maintenance facility. Integrity is based on the maintenance of dependency links between ingredient and dependent data values. Changes to data values are propagated along the network of dependency links. Only those values affected by a change are updated.

Chapter 7 discusses the implementation of default values. Default values serve an important purpose in providing an illusion of completeness to the description when the description is still incomplete due to the absence of necessary parameter data. Default values do not take the place of input values, but rather, are clearly identified as default values; thus identified, their use can be strictly limited.

Chapter 8 discusses implementation of the explanation facility and the “menu” constraint which specifies the class of acceptable attribute values. The explanation facility offers textual definitions of the forms, instances and slots in the knowledge base. When the user is asked to input a parameter value, a complete menu of permissible values is provided.

Chapter 9 demonstrates the features of the description system. The reader is invited to refer to the relevant sections of Chapter 10 when reading the previous chapters. To this end, the organization of sections within this chapter follows as much as possible the sequence of the preceding chapters.

Chapter 10 presents example applications of the description system to the description of a simple class of building structures. These examples demonstrate extensions of the knowledge base which provide a richer representation of the description domain. The applications include an interface to a solids modelling program, and the computation of building loads.

A summary of the thesis, recommendations for further development, and the overall conclusions of the thesis are presented in Chapter 11.

Appendices A to C present a listing of the basic knowledge base and the knowledge base extensions used in the example applications of Chapter 10. Appendix D briefly discusses the description system code and describes the computer resources used. A Glossary of important terms used in the thesis follows, as also does the Bibliography.
Chapter 2

Literature Review

This literature review serves as an introduction to the major concepts upon which this work is based. The concept of integrated design, which provides the motivation for developing an independent design description, is introduced. A survey of some integrated design systems is presented and note made of the important features that these systems share. Database issues are considered, as these are inseparable from discussion of description. Geometric modelling is noted as being important in providing an unambiguous representation of solid shapes and providing facilities for computing solid properties and creating graphic images. Finally, the contribution of symbolic computing techniques for knowledge representation is examined: both Expert Systems and Frames are reviewed.

2.1 Integrated design

A desirable and increasingly attainable goal in computer-aided design is the development of integrated design systems.\(^1\) Means for integrating stand-alone programs into a single comprehensive system are discussed by Eastman [Eastman88] among others. Tomiyama [Tomiyama85] identifies three levels of integration:

1) integration of system;
2) integration of data; and
3) integration of knowledge.

Integration of system means that “[a]ll design works can be done on the one system.” Integration of data means that “[a]ll the information necessary for design can be obtained from the system.” Integration of knowledge means that one data description method is used throughout the design; “it should be unique and commonly used in all the designing stages.”

\(^1\) “The 1980s will see a major trend towards integrating individual, stand-alone, engineering application programs into comprehensive computer-aided design (CAD) systems. The unifying element of such systems will be an integrated database, containing up-to-date information about the evolving design” [Ras dor85].
Integration of data can be taken as an expression of the concept of data capture. Data once input is thereafter available to all applications; any piece of data need be input only once.

The need for general purpose building description systems is looked at by Eastman [Eastman76a]

2.2 Computer-aided description

A recently published survey of commercial computer-aided design systems for building design is the work by Port [Port84]. Among the systems described are RUCAPS,² GDS,³ IGDS,⁴ DMRS,⁵ ACROPOLIS,⁶ and CADRAW.⁷ These systems have in common that they are all principally draughting tools that store drawing information rather than the underlying structural or architectural information. On this account they cannot generally be extended to engineering analysis or design, although some of these systems can associate non-graphic data with the stored graphic entities. Such associated data can be used for quantity surveys or report generation, and can be used for simple sizing.

Interesting design/draughting systems have been developed in the field of process plant design. Leesley’s informative review of computer-aided process plant design [Leesley82] contains descriptions of two systems: PDMS⁸ [Trickett82] and Computervision Piping Design⁹ [Handel82]. These systems provide more than draughting capability as they are both based on a descriptive design database. Draughting is treated as one of many possible applications using a common database. Other applications include piping analysis and design, interference detection, quantity surveys, etc. Piping design demands an awareness of component locations, connectivity and compatibility. Piping design also requires description of

² GMW Computers Ltd. This system is also discussed by Davison [Davison78] [Davison80].
³ Applied Research of Cambridge Ltd.
⁴ Intergraph Ltd.
⁵ Intergraph Ltd.
⁶ BDP Computing Services Ltd. and CARBS Ltd.
⁷ Ove Druq Partnership.
⁸ Compeda Ltd.
⁹ Computervision Corp.
the supporting and surrounding structure for interference checking.\textsuperscript{10} These piping design systems have achieved a high degree of integration.

An example of the current state of the art in integrated structural design is the Intergraph RandMicas design system (IRM) [Intergraph87]. The heart of this system is the Common Structural Database (CSD) [Intergraph86] which allows common structural data to be passed among several application programs as illustrated in figure 2-1. The common database can be used for description of both steel and concrete structures. Steel members are defined by an orientation vector, a section and the position of the section relative to the vector. Each section has a “notional box”, a minimal size rectangle which encloses the section. Within this box nine “cardinal points” are defined—these occur at the corners of the rectangle and at its center. The section can be positioned relative to the orientation vector at any of these points. The database accesses a separate table of sections which includes the standard steel sections and which may be expanded to include custom built-up sections defined by the user. The data record for each element also records end connectivity, end loads, and member end cut-backs.

The description of concrete members is basically similar defining both the geometry of the concrete member and the location of reinforcing bars. Bars may be defined according to standard type or as custom shapes.

The common database serves primarily as a means of transferring descriptive data between applications and does not replace databases associated directly with those applications which may contain data more directly pertaining to the application.

The Intergraph system is an exemplar of the fundamental concepts of integrated computer-aided design systems. The system is fashioned around a central database that stores building description and design data. The core system provides a consistent user interface, graphical interaction, design visualization (both 2D and 3D), data management, and document preparation. Other applications may be added to this core system.

Taskmaster [Grimble83] is a commercial system that makes use of a central independent database and application programs relating to structural analysis and design. The database representation includes node data, member incidences, member loads and member forces.

\textsuperscript{10} "A complete 3D model of the pipework is of little use without the other elements of the model (i.e., steelwork for clash detection purposes)" [Button84].
Figure 2-1: The Common Structural Database.

Data is automatically formatted for the individual applications. The user interacts with the system rather than applications.

2.3 Databases

Database issues are pertinent because description of even a modest engineered artefact entails the storage and use of a considerable amount of data. The data must be so stored that it can be accessed to supply input values to the design application programs.

Any database that is be more than a simple input/output file requires the services of a Database Management System (DBMS). The DBMS presents the user with an apparent organization of the data while taking care of the actual manipulation of data in physical storage. The user accesses the data by means of a Data Manipulation Language, in terms

\footnote{The data storage requirements of large engineering projects are quite remarkable: "Approximately 3000 drawings are needed to describe a medium sized petrochemical plant" [Lessey78]; "A large project ... can consist of 20,000 pipelines, 50,000 drawings and schedules, and 30 million data items" [Hall84a].}
of the DBMS data organization. The general scheme of data organization is referred to as the data model.

A good introduction to databases and data models is provided by Date [Date86], particularly with regard to the relational model. The relational model is considered to be especially suited to the requirements of the engineering design database. The features most often cited in favour of the relational model are its logical data organization, which represents relations among data in the same manner as it represents data, and the existence of a firm mathematical foundation, the mathematics of relations, which provides a rational means of manipulating data.

The ideal form of the integrated design system is a central, independent database upon which the body of application programs all draw. The central database holds data in common storage to be used by all applications. Applications access only that part of the data which they require. The organization of the data within the database can be independent of the applications, that is, data representation is not referred to in the application program. Rather, the database manipulation language is used to create an input file for the application, and to take the output file and handle the storage of output data to the database [Eastman81a]. With this layout, either the database or the application can be changed without affecting the other.

Date clearly explains the role of the DBMS and the value of data independence [Date86]. Eastman covers all the main topics concerning engineering databases, including data models, and the layout of integrated systems [Eastman78] [Eastman81a] [Eastman81b]. With regard to process plant databases Buchmann covers the same topics [Buchmann82]. Buchmann also discusses the methodology of database design; that is, given a data model how to best represent the data in terms of that model. He includes an example taken from process plant design.

---

12 In some systems there is also a Data Query Language by which users query the database.

13 Introductions to the relational model are also presented by Sandberg [Sandberg81], and (with structural engineering examples) by Schaefer [Schaefer82].

14 "The relational model has been gaining popularity in most areas of database applications because of its high level user interfaces and its underlying clarity. It is definitely a model that is well suited to representing CAD databases and its ease of use can do much to simplify the implementations" [Buchmann84]. "The versatility and flexibility of the relational model make it ideal for managing structural engineering data" [Fenves82].
Engineering databases are different from business application databases in that they are “generative,” i.e., the data is continually growing in the database as it is generated by the application programs. The point at which the database is complete essentially marks the end of its usefulness in design. This aspect of engineering databases is discussed explicitly by Rasdorf [Rasdorf85]. The fact that applications do not only change data, but far more importantly, create vast quantities of new data, is of critical importance to designing engineering design databases.15

Rising directly from this is the need for integrity management in the design database. Integrity exists when data items that are related functionally have values that are consistent with that relation. Given the complex functional relations that exist among data items, often established through extensive computations, integrity is difficult to maintain. Yet inconsistency cannot be permitted to exist unnoted in the database.

An additional desirable feature in the design database is constraint management. Integrity relations are one kind of constraint [Fenves82]. Design requirements can usually be expressed as constraints. Interaction between design disciplines often takes the form of communicating constraints across discipline boundaries.

It is a small step from enforcing constraints to using constraints to assign values to data items. With this step the database can take an active part in the design process. The extent to which process can be accommodated in the database is limited by the complexity of some of the constraint relations:

Functional integrity and consistency with respect to governing laws must be enforced by application programs. These constraints cannot be built into the databases because they must invoke high-level complex sequences of computations . . . [but] there are many constraints that can be. Design criteria, interaction constraints, consistency constraints on redundant data, and iteration control are examples of constraints that must be enforced in a structural engineering database. [Fenves82]

15 "The engineering design process is an evolution of a representation. Only when the process is finished is a 'complete' database achieved. The emphasis in the engineering design process, therefore, is on the computational generation of new data" [Rasdorf85].
2.4 Geometric modelling

The purpose of design is to develop a description of a physical artefact. From the description, the artefact itself can be fabricated. The accurate description of solid objects involves some manner of geometric modelling. Engineering drawings are one means to describe the geometry of an artefact, but such drawings are not unambiguous. To provide a more exact description of geometric entities, several geometric modelling systems have been developed. These systems "do not manipulate physical objects; rather, they manipulate data (symbol structures) which represent solids" [Requicha80].

Useful geometry systems have four primary components: (1) symbol structures which represent solid objects; (2) processes which use such representations for answering geometric questions (such as "What is the volume?"); (3) input facilities, that is, means for creating and editing object representations and for evoking processes; and (4) output facilities and representation of results. [Requicha80]

One of the important offerings of geometry modelling systems is a graphics display of the described solid. This allows visual verification of the description, as well as providing a valuable communication tool.

There are several surveys of the state of geometric modelling available [Baer79] [Davies80] [Press83]. The reader is especially directed to the paper by Requicha [Requicha80] for a review of the principles and applications of geometric modelling.

2.5 Expert systems

An expert system is a program designed to emulate the reasoning processes of a human expert in some limited application area. The typical expert system relies on production rules to represent expert knowledge. A production rule is a conditional of the form "IF (some set of antecedent conditions is true) THEN (some set of consequent conclusions can be drawn)" [Graham88]. Each rule represents an "independent nugget of know-how" [Hayes-Roth85]. The set of rules establishes the problem domain of the expert system. In its simplest implementation the knowledge base is an unordered set of production rules. By repeatedly

---

16 "The traditional means for specifying solids—engineering drawings—are best viewed as informal means of communication between humans" [Requicha80].
evaluating the rule set, and asserting any new conclusions, the expert system can establish a line of expert reasoning.

Expert systems share some fundamental properties with logic programming languages. The following view of logic programming applies equally to expert systems.

The key idea underlying logic programming is programming by description. In traditional software engineering, one builds a program by specifying the operations to be performed in solving the problem, that is, by saying how the problem is to be solved. The assumptions on which the program is based are usually left implicit. In logic programming, one constructs a program by describing its application area, that is, by saying what is true. The assumptions are explicit, but the choice of operations is implicit.

A description of this sort becomes a program when it is combined with an application-independent inference procedure. Applying such a procedure to a description of an application area makes it possible for a machine to draw conclusions about the application area and to answer questions even though these answers are not explicitly recorded in the description. This capability is the basis for the technology of logic programming. [Figure 2-2] illustrates the configuration of a typical logic programming system. At the heart of the program is an application-independent inference procedure, which accepts queries from the user, accesses the facts in its knowledge base (the description), and draws appropriate conclusions. It is thus able to answer users' questions and, in some cases, to record its conclusions in its knowledge base.

Because the inference procedure used by a logic program is independent of the knowledge base it accesses, program development amounts to the development of an appropriate knowledge base—finding a suitable description of the application area. There are several advantages to this. Chief among these is incremental development. As new information about an application area is discovered (or perhaps just discovered to be important to the problem the program is designed to solve), that information can be added to the program's knowledge base and so incorporated into the program itself. There is no need for algorithm development or revisions.

A second advantage is explanation. With the piece-meal nature of automated reasoning, it is easy to save a record of steps taken in solving the problem. By presenting this record to the user, it is possible for a program to explain how it solves each problem and, therefore, why it believes the result to be correct. [Genesereth85]
Figure 2-2: A logic programming system (after Genesereth).

The explicit representation of knowledge is a significant advance in programming, bringing the program much closer to the problem domain. As Abbott states:

We have known all along that every program is in some sense "frozen domain knowledge."
But it is only with the advent of knowledge-based systems that we have become aware of the significance of this... It is the domain-level knowledge embodied by a program that represents the real connection to the problem domain... We have finally become aware that it is possible to develop programming languages in which we can express problem domain knowledge explicitly. [Abbott87]

Despite their fundamental similarity, expert systems differ from logic programming languages in several important ways. Foremost of these is the use of inexact reasoning or "heuristics." It is argued that experts are able to solve problems that are not amenable to simple algorithmic solutions by making use of inexact reasoning based on experience. This inexact reasoning is usually characterized as rules-of-thumb. The emphasis in expert systems is on "practical problem-solving knowledge" [Hayes-Roth83]. The expert system therefore promises a plausible solution to a problem without being able to guarantee the correctness of that solution [Harmon85]. Expert systems are seen as particularly suited to "real problems" which have "a complex texture of regularity punctured by exceptions"—"[t]he

---

17 Romanycia and Pelletier have conducted a thorough survey of the use of the term heuristic in Artificial Intelligence (AI) literature and have come up with the following definition which they believe "summarises what the majority of AI theorists mean by the term": "Concisely put, a heuristic in AI is any device, be it a program, rule, piece of knowledge, etc., which one is not entirely confident will be useful in providing a practical solution, but which one has reason to believe will be useful, and which is added to a problem-solving system in expectation that on average performance will improve" [Romanycia85].
regularity of the domain enables us to actually build something whereas the exceptions foil conventional programming technology" [Filman88].

The use of inexact reasoning has made the evaluation of uncertainty a major issue in expert systems. Most expert systems have some means of indicating the level of certainty of their conclusions. Evaluation of certainty, from the nature of the approximate reasoning in expert systems, must itself be approximate. The uncertainty of their conclusions is an accepted and necessary feature of expert systems. As Waterman says:

While conventional programs are designed to produce the correct answer every time, expert systems are designed to behave like experts, usually producing correct answers but sometimes producing incorrect ones... At first glance it would seem that conventional programs have a distinct advantage over expert systems in this regard. However, the advantage is an illusion. Conventional programs for performing complex tasks, like those suitable for expert systems, will also make mistakes. But their mistakes will be difficult to remedy since the strategies, heuristics, and basic assumptions upon which these programs are based will not be explicitly stated in the program code. [Waterman86]

Expert systems have attracted much interest and an extensive literature has been generated. The standard introduction to the topic is Building Expert Systems [Hayes-Roth83], a joint achievement of leading researchers in the field. A Guide to Expert Systems [Waterman86] is a suitable companion volume which includes a comprehensive survey of existing expert systems. The work by Graham and Jones [Graham88] is an excellent introduction to logic and uncertainty in expert systems. There are a large number of introductory texts on the subject; a selection of works consulted includes [Forsythe84a], [Frost86], [Harmon85], [Jackson86] and [Rauch-Hindin86]. Among the many papers dealing with the topic, a good general introduction is given by Hayes-Roth [Hayes-Roth84].

The number of expert systems dealing with engineering issues is small but growing. Gero has edited collections of papers dealing with expert system applications to computer-aided design [Gero85a] [Gero87]. Pham has collected papers dealing with expert systems in engineering [Pham88]. A paper by Sriram looks at the use of expert systems in structural engineering [Sriram85]. Another by the same author looks at expert system development in progress circa 1986 and includes some engineering related systems [Sriram86]. Brief descriptions of operational expert systems are given by both Waterman [Waterman86] and
Harmon [Harmon88]. Recent developments at Carleton University include a demonstration expert system to diagnose cracks in masonry walls [Cornick85].

2.6 Frames

The theory of "Frames" is just one of many schemes for representing knowledge in computers. Among other schemes are logic programming, production systems and semantic networks.

Logic programming and expert systems are two approaches to representing knowledge in terms of logical relations (hence logic programming is also referred to as "relational programming" [Davis85]).

Semantic networks represent knowledge as relations between objects. For example, the relation "Mary likes John" may be represented as a link—the "likes" relation—pointing from Mary to John. Semantic networks lend themselves to graphical presentation. Semantic networks include the idea of "inheriting attributes along a special link for constructing taxonomic hierarchies" [Tichy87]. That is, the generic object within a network may represent a class of instances. The network may also recognize subclasses of these classes; the subclasses will share the associative links of the superclass. Semantic relations can be expressed as individual facts [Winston77].

Frames are a development of semantic networks, but whereas semantic networks emphasize relationships, frames emphasize objects [Winston77]. A frame is a data structure that consists of a number of slots (data fields). Each slot has its own list of properties, i.e., its own attached knowledge. The attached knowledge may include both data and procedures; for example, there may be a procedure to compute the slot value, a default value, a

---

18 "There seems to be little agreement on how knowledge is represented in the human mind. Without an agreed objective model, it is inevitable that many competing and complementary formalisms will be proposed and used with varying degrees of success in the endeavour to create adequate computerized replications. Indeed, it is recognised that many formalisms are necessary, and will tend to be used in parallel, each having its own strengths and weaknesses under various circumstances." [Graham88]

19 A survey of these and other representations, including frames, is given by Barr [Barr81]. Also, both IEEE Computer and the Communications of the ACM have devoted special issues to the topic of knowledge representation [Computer83] [ACM85]. Cercone [Cercone87] offers a similar selection of papers, many based on articles appearing in the special issue of IEEE Computer.

20 Brachman, in describing the frames language KL-ONE, observes that frames provide "an object-centered factorisation of knowledge, rather than the more common factorisation in which knowledge is structured as a set of facts, each referring to one or more objects" [Brachman85].
constraint on the slot value, comments, etc. A frame can be used to describe an object, an event or any stereotypical situation. For example the frame Mary may have the slot likes whose value is a pointer to the frame John. As with semantic networks, inheritance plays an important role. One inheritance relation is the instance relation. The frame Mary, being an instance of the generic frame Girl, inherits the slots of that frame. A generic frame represents a class of instances with a common set of slots. The inheritance relation a-kind-of (or is-a) lets subclasses inherit slots from their super-classes. While these two inheritance relations are standard to frame implementations, additional inheritance relations are also possible.\textsuperscript{21} Several modes of inheritance can also be applied within the standard relations (as in the Units language [Stefik79]).

The theory of frames was introduced by Minsky in a seminal paper, in which he gives the following as the essence of the theory:

When one encounters a new situation (or makes a substantial change in one's view of the present problem), one selects from memory a structure called a frame. This is a remembered framework to be adapted to fit reality by changing details as necessary.

A frame is a data-structure for representing a stereotyped situation ... Attached to each frame are several types of information. Some of this information is about how to use the frame. Some is about what one can expect to happen next. Some is about what to do if the expectations are not confirmed.

We can think of a frame as a network of nodes or relations. The top levels of a frame are fixed, and represent things that are always true about the supposed situation. The lower levels have many terminals—slots that must be filled by specific instances of data. Each terminal can specify conditions its assignments must meet. [Minsky75]

Or as Goldstein summarizes:

Frame theory contends that (1) intelligence arises from the application of large amounts of highly specific knowledge, as opposed to a few general inferencing mechanisms. and (2) this is accomplished through the use of a library of frames, packets of knowledge that provide descriptions of typical objects and events. These descriptions contain both an abstract

\textsuperscript{21} "While many frame languages provide only two relations (variants of is-a and instance), advanced languages allow the user to define new relations. Commonly needed relations express connections such as aggregation, elaboration, abstraction, revision, default, precedence, similarity and causality. Inheritance coupled with these relations simplifies processing the network because inheritance eliminates redundant representation yet gathers together relevant information automatically" [Tichy87].
Chapter 2: Literature Review

Template providing a skeleton for describing any instance and a set of defaults for typical members of a class. The defaults allow the information system to apply missing detail, maintain expectations, and notice anomalies. [Goldstein77]

Frames have some interesting properties which have been selectively exploited in various implementations. Minsky's theory of frames, in attempting a general representation of knowledge, covers a lot of ground. As Winston explains, the theory of frames, far from being "a single, easily debated notion" is "an abstract theory of network descriptions, terminals, prerequisite criteria, transformations, defaults, expectations, and information retrieval—in short, frames theory is an armamentorium rather than a weapon" [Winston77]. The wealth of ideas in the theory has proved a rich fund from which many have drawn, but as Schank says, "The frames idea is so general ... that it does not lend itself to applications without further specialization" [Schank75].

One property of frames that has proved of considerable interest is that of providing an object-centered representation. Bobrow states:

Although there are variations in exactly what is meant by object-oriented programming, in all these languages there are objects that combine state and behavior... There are three major ideas in object-oriented programming: (i) objects are defined in terms of classes that determine their structure and behavior; (ii) behavior is invoked by sending a message to an object; and (iii) descriptions of objects may be inherited from more general classes. [Bobrow86]

The object-oriented language Smalltalk [Bte81] is based on frame-like computational objects. These objects represent classes and instances. Classes are arranged in taxonomies.

22 Hayes comments: "Minsky introduced the terminology of 'frames' to unify and denote a loose collection of related ideas on knowledge representation: a collection which, since the publication of his paper has become even looser. It is not clear now what frames are, or were ever intended to be ... [A]s an historical fact, 'frames' have been extraordinarily influential. Perhaps this is in part because the original idea was interesting, but vague enough to leave scope for creative imagination" [Hayes79].

23 While message sending is indeed a common feature of object-oriented languages, it is not an essential feature of object-oriented programming. "It is often assumed that an important aspect of object-oriented programming is the notion of sending a message to an object—that is, a frame—in order to invoke an operation associated with that object. While the association of objects with operations is important, message sending is not. Message sending is merely a convenient notation for invoking an operation associated with a frame. In that light, object-oriented programming languages would hardly differ from programming languages with abstract data types. The essential characteristic is the ability to organise abstract data types into a hierarchy of sub- and supertypes and to inherit slots and operations from a type to its subtypes. Frame languages support this style of programming because of the built-in inheritance mechanism" [Tichy87].
and sub-classes inherit the properties of their super-classes. Smalltalk objects differ from
standard frames in that the attached procedures are attached directly to the frame instead
of to the individual slots. This simpler organization gives Smalltalk greater consistency as
a language for general computing.

In most frame systems, data and procedures are attached directly to the frame slots.
This allows each of these frame attributes to be more richly defined. An additional benefit
is that procedures can be implemented as "demons." A demon is a procedure that is
performed as a side effect of some other action. In the FRL frames language the slots have
*if-added*, *if-removed* and *if-needed* demons [Goldstein77]. These procedures are not directly
invoked. The if-added and if-removed procedures are run as side-effects whenever a value is
added to or removed from the slot. The if-needed procedure computes a slot value as a side-
effect of a request for the value. The object-oriented language Loops recognizes what are
called "active values" [Stefik86b]. These slots have procedures that are activated whenever
the slot value is accessed. The use of demons gives rise to data-directed programming.\(^\text{24}\)
In data-directed programming the data drives the process. Rather than having a fixed
sequence of procedures operating on a set of data, the data itself determines the sequence
in which the procedures are to be run. Requesting a data item triggers the procedure for
its evaluation, or adding a value triggers procedures that make use of that value.

The idea of matching is also central to the theory of frames. Frames can represent a
class of objects (or a stereotypical object). Faced with an unclassified object, one can classify
it by finding its matching frame. An example of this aspect of frames is provided by the
expert system Centaur [Aikins83]. This expert system "performs tasks in the domain of
pulmonary (lung) physiology," namely diagnosing a class of lung diseases based on patient
data and the results of respiratory tests. The system matches the objective data with frames
representing various disease states.\(^\text{25}\) The matching however does not depend on complete
data. Partial data can be matched to several applicable frames, one of which is taken as the
current hypothesis. Frames are "expectation-driven processing" [Barr81]—once a disease
frame is chosen, the system can go on to fill the remaining slots of the frame, either by
performing some internal computation or by asking the user. In this sense, "[a] frame is

\(^{24}\) Bobrow and Stefik term this "access-oriented" programming [Bobrow86b] [Stefik86b].

\(^{25}\) "Each pulmonary disease prototype [frame] in Centaur represents expected lung test results for
one pulmonary disease or its subtype... The overall goal of the system is to match the actual test
results and patient data with one or more of the prototypes" [Aikins83].
a collection of questions to be asked about a hypothetical situation: it specifies issues to be raised and methods to be used in dealing with them" [Minsky75]. If the newly acquired data does not meet the expectations of the hypothesis, that frame can be discarded and a more suitable one chosen. In Centaur the hypothesis frames are rated on how well they fit the data—this measure replaces the certainty factor used by most expert systems.

Another example of a system that makes effective use of matching is 3D-Form [Walker88]. This system completes partial descriptions of 3D shapes by finding a specialized frame that fits the data. The system thus arrives at a plausible complete description that satisfies the partial data. Of course, correct completion of the shape cannot be guaranteed.

Frames are becoming popular adjuncts to rule-based expert systems. The rationale behind incorporating frames in rule systems is that "[i]nasmuch as the typical objects [dealt with in an expert system] are fairly abstract, a rule-based approach in and of itself does not provide an adequate representation. This is because rule-based systems typically do not provide a way to describe objects with a large number of complex attributes" [Dym88]. As frames accommodate procedures attached to the frame slots, there is no reason why these procedures cannot use production rules. Frames provide a description of the knowledge objects and also serve to group rules around the objects they affect, thus streamlining the reasoning process by reducing the extent of searches through the knowledge base. The merging of frames and production rules is an important aspect of Loops [Stefik83a] and KEE [Fikes85] (both these languages also incorporate object- and access-oriented programming).

A fair number of frame languages have been developed since the concept of frames was introduced. Each was developed for a different application and so exploits different aspects of the frame theory. The more important and interesting implementations are: CENTAUR [Aikins83], FRL [Goldstein77], KEE [Fikes85], FILM [Fikes85], KL-ONE [Brachman85], KRL [Bobra077a], KBcra077b], KRYPTON [Brachman83], LOOPS [Stefik83a], [Stefik86a] [Stefik86b],

---

26 "The major inadequacies of production rules are in areas that are handled by frames. A great deal of success, in fact, has been achieved by integrating frame and production rules languages to form hybrid representation facilities that combine the advantages of both component representation techniques (e.g., Loops, KEE and Centaur). These systems have shown how a frame language can serve as a powerful foundation for a rule language." [Fikes85]

27 "Prototypes [frames] guide the invocation of the production rules, focusing the search for new information and eliciting the most relevant information from the user. The prototypes provide a structure for representing knowledge not represented in rule-based systems: expected patterns of data. They also provide an explicit means of representing some of the knowledge that was represented implicitly in rules, including control knowledge and default values." [Aikins83]
SRL [Fox86] [Reddy86], UNITS [Stefik79], XRL [Barbuceanu84] [Barbuceanu85], and 3D-FORM [Walker88]. Waterman gives a very brief description of a list of frame-based languages that includes most of the above and some others [Waterman86].
Chapter 3
Abstraction and Knowledge

This chapter will look at abstraction as data structure. This structure is based on the Platonic "form" which is taken as a list of the essential attributes of a class of objects. The attributes of a class are those things that characterize the class. Two kinds of attributes are considered: parameters and properties. The relation of sub-classes to their super-classes is examined and the concept of specialization hierarchy is introduced.

3.1 Abstraction

In design it is desirable, one might even say that it is essential, to be able to both express and interpret ambiguous description. One means of expressing ambiguity is through abstractions.

An abstraction is a representation that purposely ignores detail in order to present a clearer and more immediately useful view of an object.

An abstraction of some system is a model of that system in which certain details are deliberately omitted ... The objective is to allow users to heed details of the system which are relevant to the application [i.e., the task at hand] and to ignore others. [Smith??]

The use of abstraction is not only important for description, it is also important to the process of design itself. Obviously, abstraction is needed in conceptual design, but even those aspects of design most firmly based on the scientific need abstraction. Partly, this is so because we can generally solve only simple problems. Much of engineering involves the reduction of large difficult problems to many small, easy problems. Analysis relies on the construction of simplified models. The model is a view of the problem with the unimportant detail removed so that the full power of analysis can be brought to bear on the essential feature of the problem.

The abstraction process in description is distinct from the modelling process in analysis. Abstractions that are useful in description will not necessarily be those that are best for analysis. The analytical model often represents a rephrasing of the structural description
that, while excusable in terms of analysis, is nevertheless a corruption of the description. The analytical model usually proceeds from the description; it is not often a good medium for description in itself.

Description in terms of explicit abstractions offers a useful model for the design process.

It can be argued that designs can not only be described by their conformity to type, but also produced through them. It is quite usual in design to start with a particular type and bring 'elements' together to produce an artefact by which the type is exemplified. In other words, design is a process of instantiation—proceeding from a class description to a description of an instance. So, according to this model, in design we select a type and progressively refine it in response to a particular context. [Coyne88]

The use of abstraction in design is further illustrated by Bryant [Bryant77]. Bryant's paper proposes three useful abstractions in the architectural design of buildings. The authors emphasize the importance of the abstract in design:

While it is true that the physical object is the common denominator of design representation, it is also true that the designer—whether architect, structural engineer or consultant—nearly always works with an abstraction of the building rather than with a complete physical description. Indeed, when many important decisions are being made, there is as yet no physical description; instead there is only a collection of abstractions.

The purpose of abstraction is to present an incomplete description of an object at some useful level of detail. The complete description grows out of the abstract description by the addition of detail. The relation of the description to the abstraction is not necessarily based on physical geometry but may be purely functional. The abstraction forms the basis for completing the physical description.

A view of abstraction from the perspective of artificial intelligence research affirms the importance of abstraction in problem solving:

The value of abstraction is well-known in artificial intelligence. The basic idea is that in order to efficiently solve a complex problem, a problem solver should at first ignore low level details and concentrate on essential features of the problem, filling in the details later. The idea readily generalizes to multiple hierarchical levels of abstraction, each focused on a different level of detail. Empirically, the technique has proven to be very effective in reducing the complexity of large problems. [R. E. Korf quoted in Darden87]
3.2 Generalization

There are two kinds of abstraction. Abstraction can be achieved either through aggregation or generalization:

Aggregation refers to an abstraction in which a relationship between objects is regarded as a higher level object . . .

Generalization refers to an abstraction in which a set of similar objects is regarded as a generic object. In making such an abstraction many individual differences between objects are ignored. [Smith77]

In terms of object description, aggregation applies to assemblies. The kind of relation represented by aggregation is expressed by the complementary expressions “consists of” and “is a part of.” With aggregation, an assembly of individual parts is viewed as an individual entity in itself.

Eastman discusses the use of abstraction in representing design problems with examples from building architecture. He proposes an organization of abstraction hierarchy based principally on aggregation [Eastman81b]. The concept of abstraction hierarchy is given by Smith:

In some applications a system may have too many relevant details for a single abstraction to be intellectually manageable. Such manageability can be provided by decomposing the model into a hierarchy of abstractions. A hierarchy allows relevant details to be introduced in a controlled manner. The abstractions on any given level of the hierarchy allow many relevant details to be (temporarily) ignored in understanding the abstractions at the next higher level. [Smith77]

Generalization considers the generic object. Differences in basically similar items are ignored so that they may be treated in the same way. The concept of generalization is discussed by Plato in The Republic:

Shall we then, start the inquiry at this point by our customary procedure? We are in the habit. I take it, of positing a single idea or form in the case of the various multiplicities to which we give the same name. Do you not understand?

I do

In the present case, then, let us take any multiplicity you please; for example, there are many couches and tables.
Of course.

But these utensils imply, I suppose, only two ideas or forms, one of a couch and one of a table.

Yes.

And are we not also in the habit of saying that a craftsman who produces either of them fixes his eyes on the the idea or form, and so makes in the one case the couches, and in the other the tables that we use, and similarly of other things? ... What of the cabinetmaker? Were you just not now saying that he does not make the idea or form which we say is the real couch, the couch in itself, but only some particular couch?

Yes I was.

Then if he does not make that which really is, he could not be said to make the real being but something that resembles the real being but is not that. But if anyone should say that being in the complete sense belongs to the work of the cabinetmaker or the that of any other handicraftsman, it seems that he would say what is not true.

That would be the view of those who are versed in this kind of reasoning. [Plato63]

What Plato calls a form is a generalization. He separates, for example the idea of couch from the physical couch. The idea of couch is indistinct. One cannot in reality create an ideal couch. No couch can be the ideal couch but all couches are a single vision of the ideal.

### 3.3 Forms

A form, then, represents the essential attributes of a class of objects. The form may be understood as a class definition. Objects within a class have common attributes. For example, all objects belonging to the class of rectangles have attributes of depth, breadth, area, etc. Every object in this class will give different values to the common attributes, and so be one of an infinite number of unique instances of the class.

Every form will have a certain set of attributes that are sufficient to fully define any instance. These attributes are the “parameters” of the form. The value of all other attributes will depend in whole or in part on the values of the parameters. In the case of the class of rectangles, one may consider the attributes depth and breadth as the parameters of the form. The attributes that are computed from the parameter values are the “properties” of the form. Thus the parameters are the smallest set of attributes necessary to define an instance of a form and the properties are attributes that can be derived from these.
The form is a self-contained data structure consisting of the list of attributes. Each attribute may be taken as an individual data field. Thus a form is a relation of several data fields.

Every instance of the form is a copy of the form with unique attribute values. The form itself is the pattern from which the instances are fashioned. While the form, being only a template will not contain instance values, it may very well contain knowledge of how those instance values can be computed. Instance property values can be derived from instance parameter values with the appropriate property functions. If these functions for computing property values are included in the form along with the list of attributes, the form can be directly useful in computing data values. It is possible that knowledge other than the property function can also be included in the form.

3.4 Specialization

It is the nature of a class to have sub-classes. Thus the class of two-dimensional shapes has the sub-class of rectangles, which in turn has the sub-class of squares. Every square is a rectangle, and every rectangle is a two-dimensional shape.

The form Square is a specialization of the form Rectangle. Specialization is achieved by constraining some parameter of a form. The relation of a specialization or sub-class to its super-class can be expressed as the AKO (a kind of) relation. Thus, Square is "a kind of" Rectangle with one parameter of the super-class constrained. If Rectangle is defined with parameters $d$ and $b$, then a Square can be fully defined as a Rectangle that is constrained by the relation $b = d$. This definition of Square can take advantage of the fact that all members of a class have common attributes. Square therefore will share the same properties as Rectangle.

It is always possible to express an instance of a sub-class as an instance of the super-class. Square can be expressed as a Rectangle by restoring the parameter $b$ and giving it the value dictated by the specialization constraint, $b = d$. If Rectangle has any property functions, for example the property function $Area = b \times d$, the area of Square can also be computed by this function by expressing the instance of Square as an instance of Rectangle. In general, the property functions of any class can be used to compute the property values of
any of its sub-classes. The definition of a sub-class therefore does not require any property functions as they can use the property functions of their super-class.

A sub-class does not have to have the same parameters as its super-class. It is possible to define new parameters as long as the parameters of the super-class are constrained in relation to these new parameters. For example, Square can be defined in terms of the single parameter $a$, the length of the diagonal. In this case the constraint on the parameters of the super-class Rectangle would be $b = a/\sqrt{2}$ and $d = a/\sqrt{2}$. A definition of a sub-class is achieved by redefining the parameters of its super-class in terms of the parameters of the sub-class. This ensures that it is possible to express any instance of the sub-class as an instance of the super-class, and ensures that the property functions of the super-class can be used to compute the property values of the sub-class.

This view of specialization contrasts with the usual implementation in frames systems and object oriented programming languages [Brachman83b] [Stefik86a]. In most systems, a specialization inherits the data and procedures of its super-class but not the constraints of that class; in fact, the sub-class may have more than one super-class. It is not possible to change an instance of a sub-class to an instance of its super-class because the sub-class may have properties that its super-class does not. However, it is important in a description system to be able to make this change. For example, the designer should be able to initially describing a structure in a concise manner as an instance of a highly specialized (constrained) sub-class, then make modifications to it as an instance of its less constrained super-class. Furthermore, as commonly implemented, the super-class, because it represents an intersection of the properties of its various sub-classes, is less rich in knowledge than any one of its sub-classes—moving from the sub-class to the super-class represents a loss of knowledge. In contrast to this, as implemented in the description system, the super-class is richer in knowledge than any of its sub-classes because each sub-class is a true subset of the knowledge in its super-class.

3.5 The AKO hierarchy

Any class may have a single super-class and any number of sub-classes. It is clear from this that classes can be organized in simple hierarchies. As the relation between classes is the AKO relation this hierarchy is the AKO hierarchy. Within any class all sub-classes will
Figure 3-1: The family of Quadrilaterals.

share the same properties, for example, all 2D shapes have the properties of area, moment of inertia, etc. As any sub-class can compute its property values with the property functions of its super-class, it is obvious that only the top-most super-class must in fact define its property functions. These same functions can be used by all sub-classes. Introduction of new sub-classes therefore does not involve any concern for the computation of property values. By simply ensuring that the parameters of the super-class can be computed from any new parameters of the new sub-class, the sub-class can use the property functions of the super-class.

To further the example of Rectangle and Square, consider that both of these forms are in the family of Quadrilateral. This family of shapes is shown in figure 3-1. The form Quadrilateral will itself be defined as a kind of Two-dimensional Shape. It will have a set of five parameters, \((dl, dr, b, a, c)\), as defined in figure 3-2. The area of the Quadrilateral is computed in terms of its parameters by the property function

\[
Area = \frac{dl \times a}{2} + c \times \frac{dl + dr}{2} + \frac{dr \times (b - c - a)}{2}.
\]
Figure 3-2: The parameters of quadrilateral, trapezoid and rectangle.

Trapezoid is a kind of Quadrilateral with the constraint that two sides must be parallel or, in terms of the parameters of the Quadrilateral, $dl = dr$. Trapezoid has a set of four parameters, $\{d, b, a, c\}$. The parameters of the Quadrilateral are related to those of the trapezoid by two equations: $dl = d$, and $dr = d$. To compute the area of a Trapezoid one may apply these functions to get the equivalent parameters of Quadrilateral, then perform the Quadrilateral area function.

Consider Rectangle as a sub-class of Trapezoid. Rectangle has two parameters, $b$ and $d$. It is related to Trapezoid by the following functions: $c = b$, and $a = 0$. Using these functions one can get the equivalent parameters of Trapezoid and then those of Quadrilateral, and again use the Quadrilateral area function.

While inheritance of property functions is important for consistency, one can see that it is not necessarily an efficient way of computing property values. It is generally true that as a form becomes more constrained the property functions are, in proportion, simpler.

Specialization is a restrictive means of creating new forms. It has the advantage of having a simple and effective rule of inheritance. Both properties and the means for computing properties are inherited by sub-classes.

### 3.6 Aggregation

Aggregation is the putting together of parts into a whole. Two types of aggregation can be considered: union and assembly. In both types of aggregation an object is described as consisting of a number of component objects. In union the component objects do not have a distinct identity, while in assembly each component is itself a discrete object. The difference between the two types of aggregation is significant.
Union is a versatile and important means for describing objects. It is often difficult to define a new form as a special case of an existing form, whereas to describe it as an aggregate of two or more existing forms might be quite easy. Continuing with the example of two-dimensional shapes, one might want to create a special Two-dimensional Shape, a Wide-flange Shape, as an aggregate of three Rectangles. This can be accomplished by defining a new class of Two-dimensional Shapes called "Built-up-shape." The form for Built-up-shape will differ in no way from other forms. As it is a kind of Two-dimensional Shape it must have the same properties as its super-class. Its own parameters, those attributes that define the instance of the form, will likely include a list of component parts and some information on how those components fit together. Each parameter of the super-class will be defined as a function of these parameters.

All the two-dimensional shapes discussed thus far can be defined using the aggregating form Built-up-shape and a single primitive shape, the Right-triangle: the shape Triangle may be defined as a union of two Right-triangles; Quadrilateral may be defined as a union of two Triangles; Rectangle is a specialization of Quadrilateral (through Trapezoid and Parallelogram); and Wide-flange Shape is a union of three Rectangles. Given that shape properties can be inherited along the AKO links, the properties of all these shapes can be computed with the property functions of the two primitive shapes Right-triangle and Built-up-shape. Some aggregate shapes are illustrated in figure 3-3.

3.7 Summary

Abstraction is both useful and necessary for description. An abstraction can be explicitly defined as a list of attributes. This list is called the form of the class of objects
encompassed by the abstraction. The attributes of a class are the parameters and properties shared by the class. The parameters are given values by the user of the form; the evaluation of the properties will depend of the values of the parameters. Properties are those things that one wants to know about an instance of a form; parameters define that instance.

Forms may be defined as specializations or aggregations of already existing forms. With aggregation one can describe a class or object as a union or assembly of separately defined classes or objects. A specialization is a definition of a form as a kind of existing form with additional constraints put on its parameter values. The special relation created by specialization is the AKO relation, the new form is a kind of existing form. It is the AKO relation that makes it possible for a specialization to inherit the properties of its super-class form as well as the means to compute the values of those properties.

A form can be viewed practically as a data structure in which each attribute is a data field. The form asserts the relation among the attributes. The form as data structure provides a foundation for creating a description system based on abstractions. The description can be stored as instance data relating directly to these abstractions. The inclusion of other knowledge associated with form attributes will even further increase the usefulness of the description system.
Chapter 4
Forms and Descriptions

This chapter will consider an implementation of the forms introduced in the previous chapter using the knowledge representation technique of frames. The frame slots represent the form attributes, and the slot facets hold knowledge associated with those attributes. The simple frame model, however, must be extended and modified to include what have been identified as key features of forms. These features are (i) the differentiation of parameters and properties, (ii) a strict definition of specialization which includes the inheritance of properties and property functions, and (iii) the separation of forms and instances. The description is implemented in Scheme, a modern dialect of Lisp [Scheme87a] [Scheme87b].

The implementation presented differs from other frames implementations in a number of ways: the facets are tailored to accommodate the special needs of design description; the evaluation of data values is strictly data-directed; sub-classes are true subsets of their super-classes; and within each frame may be defined other local frames.

4.1 A simple implementation

A simple implementation of frames, which is representative of frame implementations, is presented by Winston [Winston81]. This implementation is a simplified version of the FRL frames representation language of Roberts and Goldstein [Goldstein77]. This version of frames, written in Lisp, is based on association lists. An association is a list consisting of a key and one or more associated items. An association list is a list of associations. An association list takes the following form:

$\langle (\text{key} \text{ item} \ldots) \ (\text{key} \text{ item} \ldots) \ldots \rangle$

To get the associated items one specifies the appropriate key. As an example of an association list consider the following one that applies to the description of a person:

$\langle (\text{name} \text{ Dianne}) \ (\text{sex} \text{ female}) \ (\text{occupation} \text{ student}) \rangle$
This person's occupation can be retrieved by selecting the key "occupation." In frame terminology, each association is called a "slot," the key is the "slot name" and the associated item is the "slot filler."

A frame is an association list nested within an association list. In a frame, the slot filler is not associated directly with the slot name. Rather, it resides in a second level of nesting. The first level contains the slots; the second level contains the "facets" of the slot. The slot filler will be paired with a facet called "filler." Other items will be associated with other facets such as the facet called "default." Thus, the frame accommodates not only slot fillers but other aspects of knowledge associated with the slot as well. Some facets will contain procedural information, some simple data. The frame then, in general, will take the form shown in Figure 4-1. One facet in each slot will have the facet name "filler" and its value will be the slot filler.

```
(frame-name
   (slot-name (facet-name value ... value)
   ...
   (facet-name value ... value))
   ...
   (slot-name (facet-name value ... value)
   ...
   (facet-name value ... value)))
```

Figure 4-1: A frame as a 2-tier association list.

In Winston's implementation the filler facet is called "value." When referring to the slot value, what is meant is the value of the "value" facet of the slot. Slot value and slot filler are synonymous terms.

Winston provides for his frames the following facets: Value, Default, If-needed, If-added, and If-removed. Each represents one aspect of the knowledge associated with the slot. The Value facet takes the actual value of the slot, i.e., the slot value or slot filler. The Default facet takes a default value which is used if the Value facet is empty. The If-needed facet holds a procedure for computing the slot value. The If-added facet holds a procedure that is implemented if a value is added to the slot. The If-removed facet holds a procedure that is implemented if the value is removed. The last three facets are referred to as "demons"
because they are initiated not by a direct command but as a side-effect of another procedure. Thus a request for a slot value may trigger the if-needed procedure which computes the value. The storing of this value may in turn trigger the if-added procedure. Both procedures may request or change the values of slots in other frames and thus arouse other demons. The layout of this kind of simple frame is shown in figure 5-2.

```
<FRAME-NAME>
  <SLOT-NAME>
    value: <value>
    default: <value>
    if-needed: <procedure>
    if-added: <procedure>
    if-removed: <procedure>
  <SLOT-NAME>
    value: <value>
    ...
    if-removed: <procedure>
  <SLOT-NAME>
    ...

Figure 4-2: The layout of a simple frame.
```

The basic frame-handling functions are Fget, Fput and Fremove. Fget gets the value associated with a particular facet of a frame slot. Fget is the primary means of retrieving data from frames and relies on an access path consisting of frame-name, slot-name and facet-name. For example,

```
(Fget 'Dianne 'occupation 'value)
```

will return the value of the Value facet of the slot Occupation in the frame Dianne.

The function Fput inserts facet values. As with Fget the access path of frame-name, slot-name, and facet-name are parameters of the function; the facet value is an additional parameter. This function will associate a facet value with an existing facet but it will also create the appropriate facet or slot if one does not already exist. This function therefore, not only inserts facet values in frames but also alters the frame itself by adding slots and facets.
The complementary function for removing facet values is the function Fremove. Fremove deletes a facet value from a named slot facet. If as a result of deleting a facet value the facet becomes empty, i.e., there are no facet values associated with it, the facet is also deleted. If as a result of thus deleting a facet the slot becomes empty of facets, the slot as well is deleted.

In addition to these basic functions is another set of functions that are used specifically to get the slot value. These functions make use of the Default and If-needed facets so that if the Value facet is empty, either a computed value or a default value can be returned. These functions require only the frame name and the slot name as arguments.

The function Fget-v-d returns the value of the Default facet if the Value facet has no value. Fget-v-d-f will do the same, but if the Default facet also has no value it will return the value computed by the procedure associated with the If-needed facet. An example of a procedure that would be widely used in the If-needed facet is one that interactively queries the user to supply a slot value. Such a procedure would be necessary for interaction between the user and the frames system. In general, the If-needed demon can be any valid procedure.

The function Fput+ and Fremove+ are functions for inserting or deleting a slot value—i.e., the value of the value facet of the slot. These functions activate the If-added and If-removed demons.

In Winston's implementation of frames there is a special slot called "AKO." The AKO (a kind of) relation makes possible the sharing of slots and procedures between frames. The AKO relation is created by putting the name of the parent frame as the value of the AKO slot of the child frame. If a frame is unable to answer a request for a slot value the request may be passed on to the AKO-related frame. The three functions that make use of the AKO relation are Fget-I, Fget-N and Fget-Z. Fget-I will try to get the value of the named frame slot; if it fails it will try the same slot in the parent frame, and so on up the AKO chain. Fget-I looks at values only. Fget-N does the same as Fget-I, but if the AKO chain is exhausted without success it will go through the chain again looking for default values, and if this fails it will go through the chain looking for if-needed functions. Fget-Z is like Fget-N in that it looks at values, defaults and if-needed functions; Fget-Z however will exhaust the Value, Default and If-needed facets of each frame along the AKO chain.
4.2 Extensions and modifications

The frame organization used in the description system is an extension and modification of Winston’s simple implementation. This modification introduces features which are valuable for design description—some of which are available in other frames system and other which are not.

One of the key modifications to Winston’s system is the separation of instances and classes. This modification, which is a common feature of most frames systems, allows the independent definition of classes and their instances. Thus an independent knowledge base of class definitions can be compiled. An independent knowledge base is essential to engineering design as the same knowledge must be used for many different projects.

In Winston’s implementation the frame has a dual role of defining a class or an instance of a class. The AKO relation has the dual meaning of “a sub-class of” and “an instance of.” This is not an unreasonable implementation. A sub-class constrains one or more parameters of the parent, an instance constrains all parameters. An instance, therefore, can be viewed as a sub-class with all parameters constrained. However, in the extended implementation the AKO relation will mean exclusively “a sub-class of” while a new AIO relation will be introduced for “an instance of.”

An important extension of the frames model, not found in other frames systems, is the definition of local forms. A local form is a class definition that is defined within the computing environment of another class and is thus accessible to instances of that class. The definition of local forms allows the nesting of computing environments and makes possible ‘block structure’ programming within frames for a more perspicuous organization of knowledge. A practical example of a local form is the form Center, defined within the form Rectangle, which a kind of Vector-xy locating the centroid of the shape. In describing an instance of Rectangle, one can locate a point on the shape in terms of this local form. Because Center is defined locally, a form of the same name can also be defined for the shape Circle in terms of the parameters of Circle. The local form can also be used to customize classes for special applications. For example, a specialized class of building structure may
customize the classes of building components to draw on the parameters of the structure class, or to interact with other customized components. Thus the class Truss may contain specialized classes of tension and compression members customized for inclusion in trusses.

The description system also implements an uncommon rule of inheritance of super-class slots by the sub-class. While in most systems—including the most successful—the sub-class may inherit from multiple super-classes and may override super-class slots, this kind of inheritance has disadvantages for design description. It must be recognized that under such a rule the sub-class is not a true sub-class of its super-class, which is to say that an instance of the sub-class is not also an instance of the super-class. The inheritance mechanism serves as a means of "elision" [Stefik86a], that is, a means of eliminating redundant code by sharing code among similar objects. The specialization within a description, however, must express a more semantically rigorous relation such as that propounded by Brachman [Brachman85a] [Brachman85b]. It is important in description that instances of highly specialized classes be reducible to instances of more general classes. For example, if a revised analysis of an instance of the class Beam indicates that the member is axially loaded, one should be able to reduce the instance of Beam to an instance of the more general class Beam-column to take the axial load into account. For this reduction to be possible, the sub-class must not only inherit the properties of its super-class but the constraints of the super-class as well. An important additional benefit is that because the super-class is richer in knowledge than its sub-classes, the knowledge base must be structured as general classes containing the most general and widely applicable knowledge and the sub-classes amplifying knowledge of special cases. For example, the general class of solids would contain knowledge of the mechanics of solids. More specialized classes, such as Beam and Column, would not only draw on this general knowledge but supplement it with specialized knowledge. Thus, empirical knowledge about buckling of steel columns would supplement the more general knowledge of solids.

The description system also introduces facets which are useful in design description but not commonly implemented in frames systems. It is especially important in engineering design to actively maintain the integrity of design data. Throughout the process of design a description will undergo constant revision. It is absolutely imperative that dependent data values are kept current and accurate. In most frames systems, instances will represent individual "objects" within an interacting system of "objects". In such cases integrity is
maintained through the If-added and If-removed facets. When a new object is introduced
to the system it is "connected" to those objects with which it interacts. The If-added and
If-removed facets serve to inform connected objects to update when slot values are changed.
In a design description system, this type of integrity maintenance system is not sufficient.
Data is not always exchanged between discrete system objects. For example, the parameter
values of the class Rectangle, are used in the computation of the various properties of an
instance of the class, and are also accessed by other classes. Thus an instance of the class
Beam may have an instance of the class Rectangle as the value of its slot Cross-section.
The value of the Beam Volume will depend on the value of the Rectangle Area. However
Beam and Rectangle are not interacting system objects and the If-removed facet in the slot
Area cannot contain a general procedure to update Beam Volume, as this is only one of a
large number of possible classes and slots that may depend on its value. Rather what is
required is a direct integrity link between the dependent slot and its ingredient. By directly
linking slots instead of objects it is possible to implement an efficient integrity maintenance
facility which will ensure that all dependent data are updated when ingredient data values
are changed. Such a facility has been implemented.

The description system also introduces an uncommon means of handling default values.
Commonly, a default value is an initial value which the user has the option of changing.
In many cases, such defaults can be of use for supplying the user with typical or suggested
values of description parameters. The disadvantage of this type of default is that it does not
give the designer the additional option of entering "I don't know" as a parameter value. In
the early stages of design, the values of many parameters are not known and that they are not
known should be stated explicitly. Inclusion of this third option in the description system
makes possible a complete change in the meaning of the default value. The default value can
now be used as a temporary fill-in for the input value. Such a value will be non-authoritative
and values derived from it will similarly be non-authoritative. The description system in
implementing this kind of non-authoritative default treats default values as a separate
system of values with their own value-generating procedures. This permits dependent
values to be computed based on default parameters, but it also distinguishes these inferior
values from value based on authoritative input values.
4.2.1 Forms and instances

The frames implementation of the description system splits the frame into two parts: the "form," which contains only the generic information; and the "instance," which contains only instance data. The layout of the form is shown in figure 4-3 and that of the instance in figure 4-4.

In dividing the frame into form and instance, the form and the instance each have the same set of slots, but the form slots contain facets that deal with generic knowledge, while the instance slots contain facets that deal with instance data. For the most part, all slot
facets except the Filler facet deal with generic knowledge. The role of the various other facets is to aid in the evaluation of the Filler facet.

The result of the joining together of an instance and its form is a complete frame of the instance containing both instance and generic data. Interpretation of an instance can only be done in terms of this complete frame.

### 4.2.2 Associations

To accommodate the definition of local forms the form layout is given an extra outer level of associations. The local forms are grouped in an association with the key “Local-forms.” Within the added level of association the slots are grouped in an association with the key “Slots.”

Except with regard to its restricted availability, a local form is in every way a complete form. It may therefore have its own local forms. A local form may be a kind of global form or a kind of local form defined within the same form environment, that is, it may be a kind of local form inherited from its AKO parent, or a kind of local form defined in the same form.

Within the outer level of association it is also possible to gather together the slot methods. The slot may therefore contain the name of the method instead of the method procedure itself. This minor change does not increase the powers of the representation, but
as methods may be lengthy expressions, it does make for a more convenient organization. The slot methods are grouped in an association with the key "Methods."

The outer association also gives a more appropriate home to the AKO slot. Although considered a slot in the simple implementation, the AKO relation is not an attribute of the class represented by the form, but an attribute of the form representation. Therefore the AKO "slot" is brought out to the outer level as a simple association holding the name of the form's AKO parent; it does not have facets. Also added to the outer association list is the association "Defn." This association holds a short textual definition of the form for use by the explanation facility. The frame layout therefore consists of five associations: AKO, Defn, Slots, Methods and Local-forms.

4.2.3 Facets

The facets in the extended implementation are substantially different from those of the simple implementation. A principal difference is in the Value facet. The role of the Value facet in the simple implementation is split between the form and the instance. Its role as the holder of the instance value is taken by the facet Filler in the instance. Its role of specifying the instance value is taken by the facet Value in the form. In taking on the role of specifying the slot filler the Value facet also takes over the role of the If-needed facet. The Value facet may contain a simple value that is taken directly by the Filler facet, or it may contain the name of the slot method used to compute the value. Additionally, if the Value facet contains the name of a form, the Filler facet is given a local instance of that form. For example, the slots of a sub-class of the form Rectangle may contain the following Value facets:

(d ($value 20))
(b ($value Ask-user))
(reference-point ($value Center))
(area ($value (method)) )

If the user supplies the value 15 for the slot B, the slots of the instance will have the following Filler facets:

(d ($filler 20))
(b ($filler 15))
(reference-point ($filler (instance of Center)))
(area ($filler 300))
The filler for slot D is taken directly from the value facet. The filler for slot B is the result of the special Ask- user function. The filler for Area is computed by the named slot method. The filler for Reference-point is an instance of the specified form.

Because of the extended role of the Value facet, the If-needed facet is not used.

The Default facet is retained and is given an extended role similar to the new role of the Value facet. The Default facet specifies the default filler as a simple value, a method or a form.

The facets If-added and If-removed are retained—although, because of the introduction of special facets for integrity maintenance, they serve a much less important role—and some new facets are introduced. The most important of these new facets are the Menu and Defn facets. The Menu facet holds a list of the classes of acceptable slot fillers. Thus the slot Cross-section in the form Beam may have the Menu facet (§menu shape-2d) to limit the slot value to instances of the form Shape-2d or its sub-classes. The Defn facet holds a short textual definition of the slot for use by the explanation facility.

The organization of the instance mirrors that of the form. It too is given an outer association list, with the slots grouped in an association with the key “Slots.” The instance also has an outer association called “Defn.” This association holds a textual definition of the instance. Instead of the outer AKO association, the instance has an AIO association. Besides the Filler facet, the instance also has some facets for holding integrity pointers.

4.2.4 Inheritance

The internal representation of the form is basically the same as its external representation defined by the user. However, the system expands the representation of the form to take into account the inheritance of slots through the AKO network.

The inheritance mechanism in the description system works through transition slots. According to this mechanism the child form must contain slots for each of its parent’s parameters so that the values of these parameters can be constrained. As these inherited parameter slots are neither parameters of the child nor properties, they are called transition slots. The child also inherits its parent’s property slots. In effect, the child inherits all the slots of its parent. In the present implementation the slots are physically inherited by the
child. When a new form is defined by the user, the system adds the inherited slots to the form. The expanded form has all the properties and methods of its parent.

Inheritance may apply at the slot or the facet level. A child form may update an entire slot, or it may update a single facet of a slot. Those facets of a slot that are not specifically redefined are inherited from the parent.

An added advantage of this manner of inheritance is that it allows for some consistency checking of facet values. For example, the child form should only be allowed to put values on the parent's parameter slots; changing the value of a property slot would violate the rules for specialization. Similarly, the slot menu in the child must be a subset of the parent's slot menu; that is, the child may change the menu to make it more restrictive, but cannot change it to make more liberal. Each facet has its own rules of inheritance that can be checked for violation. This checking can be done as each form is defined and the expanded form is created.

The expanded form contains extra associations not found in the forms defined by the user. These are the top-level associations "Subclasses" and "Instances." Both of these associations are used in the creation of AKO menus. These two top-level associations hold the complementary pointers to the dependants and instances of the form. These associations are updated whenever a new dependant or instance is created.

The user does not directly use the expanded form. The user deals only with the forms as defined in the knowledge base.

One of the drawbacks of the physical inheritance of slots is that the child form must respect the slot names of its parent. If the parent has used a name for slot, the child cannot use the same name for a different slot, even though the parent's slot is a transition slot and not apparent in the definition of the child.

4.2.5 System modularity

There are persuasive arguments for maintaining the independence of data and process [Date86]; the validity of these arguments is not questioned. Simply put, if data is independent of process, it is possible to change one without changing the other. One way of applying this principle is to create an independent central database with any number of
individual application programs. Communication between the database and the applications is channelled through intermediate processes that extract the required data from the database for use by the applications, or take the application output and relay it back to the database. The application programs have no direct contact with the database and are unaffected by changes to the database organization. The changes affect only the intermediate processes whose job is solely that of go-between.

The mainspring of this argument is the concept of modularity, that a system should be made up of quasi-independent modules that can each be changed without requiring compensating changes in the rest of the system. Although forms inexorably link data and process, they offer their own clear and valuable modularity. Each form is a module. New forms can be added to the system with the one concern that they can compute their own property values. Forms can be defined as specializations of existing forms and thus inherit these necessary property functions.

Although process is included in the data structures, the case is quite different from the case in which external programs are dependent on a fixed database structure. Yet due to the complexity of some applications, there is a need, even in a data-driven system, for independent applications programs. In this case the independence of the database and the applications must be scrupulously maintained.

One aspect of modularity in forms is the consistency of properties within a class; within a class, all sub-classes share the properties of the parent form. To illustrate this point, consider an instance of the form Beam. This instance has the parameter Cross-section which takes as a value any instance of the form Shape-2d or its sub-classes. When Beam computes its volume it queries the filler of the slot Cross-section for its Area. This query will always succeed because all acceptable values of the slot Cross-section have the slot Area by virtue of being in the class Shape-2d. Any new form that is subsequently added to the class of two-dimensional shapes must also have the slot Area, and a method to compute its value. An instance of this new form will be acceptable value for the slot Cross-section in Beam. The representation of Beam is thus extended, without making any change to the form Beam, by adding to the range of acceptable values of its parameter Cross-section. It is the consistency of properties within a class which guarantees the compatibility of new forms with the rest of the system.
While frames are themselves computational modules, the description itself consists of larger scale modules. A set of forms can be collected together as a knowledge base. The forms, free of instance data, stand as a pure definition of the description domain. A set of related instances can be collected together as a single description. While the description is wholly dependent on the forms, the forms are wholly independent of the instances. Thus any number of separate descriptions can be created using the same collection of forms.

The forms define the language of description. Any language is limited in the range of artefacts that it can be used to describe; the vocabulary of description is necessarily finite. One of the features of the independent knowledge base is that the language of description can be revised and extended solely by changing the knowledge base, without having to touch the shell program.

As the forms provide the language of description, so the shell provides a language for defining forms. The shell, so called because it is empty of domain knowledge, is the engine that runs the description system. The shell interprets the forms and instances and performs the actual evaluations required to create and maintain the description.

The shell program in the current implementation is written in Scheme, a dialect of Lisp. One of the features of Lisp is that both data and procedures are represented as lists. This feature allows procedures to be manipulated as data. Thus the slot methods in the forms can be examined and modified as data by the shell program, but can also be incorporated in the shell computations and evaluated as procedures. The attached procedures in the form direct the course of computations by the shell.

The slot method may be any valid Lisp expression. The shell defines a set of Lisp functions that can be used within slot methods. The most important of these is the slot query function. This function allows a slot to query another slot for its value. The actual mechanism of transmitting the query, performing any necessary slot evaluations, and returning the slot value is carried out by the shell.

The forms themselves constitute a program. They define classes of computational objects and their interactions. The shell provides the programming language in which the forms are written. Although the shell sets the format of the form definitions, and is able to interpret the forms, the shell has no knowledge of actual forms. Any number of
knowledge bases can be created to be run by the shell. The shell and the knowledge base are independent entities.

The four apparent levels of program abstraction in the description system are therefore:

1) the Lisp implementation language;
2) the description system shell;
3) the knowledge base (i.e., the collection of forms); and
4) the description (i.e., the collection of instances).

The description system itself consists of the three higher levels. The user has access to the top two levels, the knowledge base and the description. There are in fact two levels of user: (1) the user who takes a hand in creating the knowledge base; and (2) the user who builds descriptions with an existing knowledge base.

4.3 Description using frames

Description is a simple process of filling parameter slots. Each slot value represents a design decision or a constraint that limits the range of possible interpretations of the description. A slot filler may be either a simple value or a frame (an instance of a form). By filling a slot with a frame, the slot is not fully constrained. The slot filler will have parameters that must also be constrained.

A description therefore takes the shape of a hierarchy of nested frames. An example description hierarchy is illustrated in figure 4-5. This figure describes a steel wide-flange shape, defined as the instance C1 of the form Component.

A Component is a homogeneous solid object. A physical solid object may be described in terms of its shape (solid geometry), location and material. In the form Component the attributes of shape and location are combined in the parameter Geometry. The other parameter of the form Component is the slot Material. The acceptable values of parameter slots are constrained by the Menu facet of the slot. The acceptable values for the slot Material are instances of the form Material. An instance of the form Steel, being a subclass of the form Material, is an acceptable value. The value of the slot Geometry must be an instance of the form Shape-3D. The form SLEE (straight-line extruded element) is a subclass of the form Shape-3d.
Figure 4-5: A description of a wide-flange component.

The filler of the slot Material, an instance of Steel, is fully defined, not having parameters to be filled. The slot Material is thus fully constrained. The filler of the slot Geometry, while constraining the shape to be a linear element of constant cross-section, does not fully define the shape. The value of Geometry is thus only partially resolved. It can only be more fully resolved by evaluating the parameters of the nested instance of SLEE.

The form SLEE has parameters Line-vector and Cross-section. The value of the slot Line-vector must be an instance of the form Line-vector-3d. The form Line-vector-3d has parameters End1 and End2, locating the two ends of the line in 3D space. The acceptable values for slots End1 and End2 are instances of the form Vector-3d. The form Vector-3d has parameters X, Y, and Z. These slots accept only numerical values and do not accommodate further frame nesting (sub-classes of Vector-3d may however have parameters that take frames as fillers). The value of the slot Cross-section in the form SLEE must be an instance of the form Shape-2d. An instance of Wshape, which a sub-class of Shape-2d, is an acceptable value. The parameters of Wshape take numerical values.

The description hierarchy is not a strict hierarchy limited to the one-to-many relation. A globally defined instance, for example, may be used to fill more than a single slot.
Figure 4-6: A description of a wide-flange component with aggregation.

The description hierarchy may also include aggregation. An example of this is illustrated in figure 4-6. This description describes the same shape as the previous description but uses the form Built-up-shape in place of the form Wshape.

A Built-up-shape is an aggregation of component shapes. In this case the wide-flange cross-section is built up from three rectangles. Any instance of Shape-2d is an acceptable value for the slot Items. The Built-up-shape could therefore be built up of other Built-up-shapes. A description hierarchy may be an extensive structure. The complexity of objects being described depends on the scope of the frame definitions.
4.4 Summary

The description system is based on Winston's simplified version of the frames language FRL with extension and modifications. The description system, for example, divides the frame into two parts, separating the generic data from the instance data. The form, which is the generic part, consists of the knowledge attached to the frame slots. The instance consists of the slot filler and other instance data. An extra level of association is added to the frame layout to accommodate local forms and the definition of slot methods.

The description system consists of three separate levels: the system shell, the knowledge base, and the description. The shell is the description system program free of any knowledge of forms or instances. The knowledge base is the collection of forms, and defines the description domain. The description is the collection of instances.

A description takes the shape of a hierarchy of nested instances. The hierarchy may include generalization or aggregation branches.
Chapter 5  
Values and Methods

This chapter looks at the manner of computing slot values. The manner of slot evaluation is both object-oriented and data-directed. The description frames satisfy the requirements for computational objects of local state and local process. The evaluations are data-directed in that slot values are computed as the side-effect of a request for the value. The description system incorporates two modes of evaluation: forward chaining and backward chaining. Parameter slots are evaluated in the forward chaining mode, that is, all parameter slots of and instance are evaluated when that instance is created. Property slots are evaluated in the backward chaining mode, that is, the property slots are only computed when their value is requested.

The ability to compute data values within a description system for engineering design is important because design description is “generative” [Rasdorf85], that is, new data (and new types of data) are continually being generated throughout the design process; completion of the description effectively marks the end of the design process. If the knowledge base is to be a thorough representation of knowledge about the objects of description, the procedures for computing design values must be included as a fundamental aspect of this knowledge. For example, the procedure for computing the value of the slot Area in the frame Rectangle is essential knowledge about the slot and in fact may be used to define the slot.

On another level evaluation is important in translating the abstractions within the description to abstractions used by independent application programs. The advantage of data independence is that the description model is not tailored to any one application but achieves a general representation that can be translated to a form suitable for each individual application. It is necessary in translating the description to a form suitable for an application to perform evaluations of data. In the description system these computations are performed, as all computations are, at the slot level. For example, the external program PADL-2, a solids modeller, requires that the solid being modelled be input in a special way—as a constructive solid geometry tree. Because the objects in the description system are not modelled in this way, a translation is required. The constructive geometry representation is

52
computed as the value of the slot Csg. This Csg representation is a general representation—further processing is required to develop a suitable input file for PADL-2—and so can also be used with other constructive geometry applications without altering the slot definitions.

5.1 Evaluation modes

Computation in the description system is based on an object-oriented approach to programming. Abelson introduces object-oriented programming in the following way:

One powerful design strategy, which is particularly appropriate to the construction of programs for modelling physical systems, is to base the structure of our programs on the structure of the system being modelled. For each object in the system, we construct a corresponding computational object. For each system action, we define a symbolic operation in our computational model. Our hope in using this strategy is that extending the model to accommodate new objects or new actions will require no strategic changes to the program, only the addition of new symbolic analogs of those objects or actions. If we have been successful in our system organization, then to add a new feature or debug an old one we will have to work on only a localized part of the system. [Abelson85]

The two essential features of the computational object are local state (i.e., local variables) and local process; the computational object is in fact a separate computational environment. The local process has direct access only to the local variables. Computation in a system of objects is achieved by the interaction of the individual objects. This interaction takes the form of “messages” sent from one object to another, and the replies received. In the foremost object-oriented programming language, Smalltalk [Byte81] [Goldberg84], the message consists of the name of the addressed object, the name of the local process to be run, and any arguments to the process. Both the arguments to the message and the reply are also Smalltalk objects.

Frames fulfill the requirements of a computational object in that they have both local state and local process: local state is provided by the frame slots; and local process is provided by the frame methods. The slot values and methods are accessible only within the frame. To gain access to values in other frames requires, as in Smalltalk, that a message be sent. In the description system however, the message is of a different form. While in Smalltalk the message names a process to be run, in the description system the message names a slot whose value is returned.
Each slot has a value method which is evaluated as a "demon" when the value of the slot is requested. Computations in the description system are strictly data-directed. It is a request for a value that causes an evaluation, rather than a request for an evaluation that causes a value.

The way slots are evaluated in the description system is similar in many respects to the way rules are evaluated in an expert system. Rules can be evaluated either by forward chaining or by backward chaining. Forward chaining starts with an initial set of facts and derives dependent facts. Backward chaining starts off with a goal—a fact that is to be proved—and works backwards, evaluating those rules that contribute to proving the goal.

Whether an expert system implements forward or backward chaining depends upon the application of the system. An expert system will often be able to operate in both modes. Forward chaining requires the user to input basic ingredient data before the system can start its evaluation of dependants. Backward chaining requires that the user set up certain unevaluated data as goals of the evaluation.

In the description system, parameters are evaluated by forward chaining so that the user can easily enter the basic ingredient data. The evaluation of property values on the other hand is delayed until requested by the user—an obvious application of backward chaining.

The implementation of forward chaining in expert systems can be quite simple: scan the rule base, firing all rules that can be fired, and repeat the process until the entire rule base has been tried without success. The description system however does not have the equivalent of the global rule base which it can scan; it does however have frames and frame slots. Starting with an initial frame, the description system can set about evaluating all slots in that frame, the user being asked to enter the values of parameter slots. If the value of a slot is itself a frame, the system can proceed to evaluate each slot of that new frame. By starting with a single frame and evaluating all slots, the entire description can be built up in a forward chaining manner.

While the description system has a good reason for forcing the early evaluation of parameters, it has an equally good reason for delaying the evaluation of properties: as more resources are allocated to maintaining data, less are available for other needs. If there is no immediate need for a piece of data and there is a guarantee of its future availability, it
should not be stored. Since properties are functions of the parameters and can be computed at any time, they should be computed only when they are needed.

This case can most clearly be appreciated in the case of a property that is rarely used. For example, a property of 2D shapes is the polar moment of inertia. Few applications may make use of this property but the slot for polar moment of inertia will be a permanent part of the frame. If this property is computed in cases where it is not needed, not only is the cost of storage lost without gain, but the cost of the computation itself is never recovered. The computation of unused values is undeniably wasteful. There is however no way to know in advance which properties will be used and which will not.

Thus, while a forward chaining mode of slot evaluation is acceptable for the evaluation of parameter slots, it is not acceptable for the evaluation of property slots. Property slots should be evaluated only on request. The description system therefore distinguishes property slots from parameter slots so that the former can be passed over in the initial forward chaining evaluation.

Property slots are evaluated by backward chaining. The evaluation, as in expert systems, starts with a goal. In this case the goal is a requested property value. To compute a property value, its slot method is evaluated. If the slot method itself contains requests for ingredient values, those ingredient slots are also evaluated. The evaluation eventually backward chains to the basic parameter slots. If the user has not already entered a value for these, their evaluation will trigger a query to the user. Computed values are passed back up the chain and result in a value being returned for the initial request.

The query is the principal means of interaction between the user and the system. The system asks the user for parameter values, and the user asks the system for property values. Even the input of values by the user can be based on query. For example, if the user wants to enter a value for the slot Cross-section, this can be done by requesting the slot value. If this slot is a parameter, it will be evaluated by querying the user. The user can then enter a value to this returned query. This indirect way of entering a slot value has definite advantages over using a separate input message. First, it obviates the need for the second type of message, thus reducing the number of commands the user is required to master and simplifying the shell code. More importantly, the query to the user, originating from the
slot whose value is to be entered, can make use of the knowledge attached to the slot in
guiding the user to an appropriate input value.

5.2 Slot evaluation

Slot evaluation is the prime computational mechanism of the description system. Slot
evaluations are data-driven: they are triggered either by a request for the value of a par-
ticular slot, or by the entering of a new slot value by the user.

The description system stores generic knowledge and instance data separately. The
generic knowledge associated with an abstraction is contained in the form, while the instance
data is contained in the instance. Slot values, being instance data, are stored in the instance,
in the Filler facet. The means of getting the slot value is stored in the form in the Value
facet.

In general, the value facet may contain any expression that evaluates to a good slot
filler. The following are all accepted expressions:

1) a numerical or logical value;
2) the name of a global or local form;
3) the name of a method;
4) one of the ask-user functions;
5) any valid Lisp expression (which includes logical and symbolic computations).

If a query to the user or a slot evaluation returns the name of a form, a local instance
of the form is created, and the name of this instance is the the slot filler.

The shell program is equipped to identify the various types of possible Value facet
expressions; their type, therefore, does not have to be designated in the form. The order in
which the shell tries to identify the expressions is as follows: null value, number, ask-user
function, local form, global form, method, other valid Lisp expression. The local form is
ranked above its global counterpart so that if a local form has the same name as a global
form, the local form will be used. An instance, not being generic information, cannot be
put in the Value facet of the form.
A slot filler may be either a single value or a list of values. In practice all slots are treated as potential multiple-value slots. A single-value slot is a slot whose method produces a single value.

Because the ask-user function is so widely used in the description system—all parameter slots must make use of it—special symbols have been associated with them: the ask-user function that queries the user for a single value is represented by a single question mark ("?"), and the ask-user function that queries for multiple values is represented by two question marks together ("?.").

5.2.1 Forward chaining

In creating a description, except for the initial frame which the user provides at the outset, all other data is entered in response to specific queries for slot values. A complete description can be built up by answering the questions posed by the description system; this is possible because the description system knows what questions to ask by virtue of the knowledge contained in the forms.

The user creates new instances with the shell function Create-instance. The syntax of this function is

\( (\text{Create-instance } \langle \text{instance name} \rangle \ (\text{form name})) \)

To create an instance of the form Beam to be named B1, the command is

\( (\text{Create-instance } 'B1 '\text{Beam}) \)

This command causes the description system to create an instance of the form Beam and then proceed to evaluate its parameter slots.

One of the parameter slots in this frame is the slot Cross-section. The method for this slot queries the user for a single slot value. In response to the query the user may enter the name of a previously created instance of a sub-class of the form Shape-2d. If the user enters the name of a valid form, the system creates a local instance of the form and puts its name as the slot value. Before continuing the evaluation of the parameter slots of the Beam, the description system will first forward chain through the newly created frame, querying the user for each of its parameter slots. For example, when asked for
the value of Cross-section for the instance B1, the user may enter "Rectangle," the name of a form. The description system, in this case, creates an instance of Rectangle with an internally generated name and then queries the user for the parameters of Rectangle. When all parameters of Rectangle have been evaluated, the system returns to the instance B1 and continues with the evaluation of its parameters.

The function Create-instance terminates when all parameter slots have been evaluated. At that time the user will have been asked to supply values for all parameter slots in the description. If the reply to a slot query is "Nil" the slot filler will be the "asked-token." The asked-token is interpreted by the system as Nil, but prevents any repeat of the query.

5.2.2 Backward chaining

The basic functions for querying a slot are the shell functions Get-value and Get-values, for querying single- and multiple-value slots respectively. The syntax of these functions is

\[(\text{Get-value} \langle \text{instance-id} \rangle \langle \text{access-path} \rangle \langle \text{querying-slot} \rangle)\]

and

\[(\text{Get-values} \langle \text{instance-id} \rangle \langle \text{access-path} \rangle \langle \text{querying-slot} \rangle)\]

If the queried slot already has a value, the value is returned. If it does not, its method is evaluated and its value is returned.

The first two arguments to the function Get-value identify the queried slot. The third argument identifies the querying slot and is used for establishing the integrity link between the querying and the queried slot. In queries from the user this last argument is "nil."

The slot is identified by an access path rather than its slot name to allow for queries to local instances whose names are not apparent to the user, and to allow for queries within slot methods, where instance names are not known. Description is a hierarchy of nested instances. The access path uniquely identifies a slot by tracing the path through this hierarchy from the querying to the queried slot. The access path is a list of slotname-value pairs. In a description hierarchy the slot value will be an instance name.

The slot value, in addition to the slot name, is needed to uniquely identify a slot if the access path goes through a multiple-value slot. The value specifies which branch of the hierarchy to follow. In the case of single-value slots the slot value is not required in the
access path and may be replaced by a wild-card token ("*"). The wild-card token is also useful in multiple-value queries, in which case the values returned along each branch are combined.

The use of the access path can be demonstrated with reference to the description hierarchy of figure 4-5. A query from the user for the cross-section area of the component C1 would be

\[(\text{Get-value } '\text{C1} ' ((\text{geometry } *) ((\text{cross-section } *) (\text{area } *)) \text{ nil})\]

If the slot Geometry or Cross-section did not have a value, this slot would be evaluated before continuing with the query to the next level in the description hierarchy. A query thus forces evaluation of all slots along the access path.

Queries are a basic part of slot methods. For example, also with reference to figure 4-5, the method for computing the value of the slot Mass in the frame C1 would query for geometry volume and material density. The access paths for these queries relative to the querying slot are

\'[((\text{geometry } *) (\text{cross-section } *))\]

and

\'[((\text{material } *) (\text{density } *))\]

Queries in slot methods are made somewhat differently from user queries. Because the form method has no knowledge of instances, it is not possible to include instance names as arguments to the query. Queries within slot methods therefore must consist solely of the relative access path. Within slots the function names Get-value and Get-values are replaced by the symbols \(\text{G}\) and \(\text{GG}\), respectively. The syntax of the queries within slot methods for single and multiple value slots is

\((\text{G} \langle\text{colon-path}\rangle)\)

and

\((\text{GG} \langle\text{colon-path}\rangle)\)

The "colon-path" is a more concise representation of the access path consisting of the slot names joined by colons. The actual slot queries in the method for the slot Mass are

\((\text{G geometry:volume})\)
and

(O material:density)

It should be noted that \( \ast \) is not a shell function. Before a slot method is evaluated it is scanned, and all lists whose first element is \( \ast \) are expanded to the equivalent Get-value function call—the missing arguments are added to the function call and the colon-path is expanded to a proper access path with wild-card tokens in place of slot values.

The expansion of methods also recognizes the symbols id and \( \ast \text{iller} \). When encountered, these symbols are replaced by the instance name and the slot value, respectively.\(^1\)

When queries go through multiple value slots, the values of each path are combined. For example, with reference to figure 4-6, the cross-section of the shape is an instance of Built-up-shape. The method for computing the Area of Built-up-shape sums the areas of the component items. The query

\[(O items:area)\]

within the method for Area returns a list of the item areas.

Queries can only be made for known properties of forms. For example, for a method in the form Component

\[(O geometry:volume)\]

is a valid query. All values of Geometry have the property Volume. However, the query

\[(O geometry:cross-section)\]

is not a valid query because not all values of Geometry will have the slot Cross-section. The extent of a single query through the description hierarchy is very limited, normally being addressed only to the next level of the hierarchy. The overall effect of the query will extend much further. For example, with reference to figure 4-5, the user query

\[(Get-value 'C1 '((mass *)) nil)\]

---

\(^1\) The expansion also recognizes the syntax \( \text{Gref (access-path)} \). This "function" redirects a query to another slot. It is a carry-over from a previous implementation, and although it is no longer strictly necessary, it is still used in the example knowledge base which will be presented in a later chapter.
Figure 5-1: Hierarchical relation of Rectangle and local form Center.

will cause evaluation of the method for the slot Mass. This method contains the query

\[(0 \text{ geometry:volume})\]

The method for the slot Volume contains the query

\[(0 \text{ cross-section:area})\]

The initial query triggers a cascade of queries that extend right through the description hierarchy, although each query is directed at only the next level in the hierarchy.

A query may extend further than a single level if the slot filler is an instance of a local form, since the parameters of the local form are known. Similarly a slot method in the local form may query a slot in the enclosing form. To indicate that a query is addressed to a superior level of the description hierarchy the symbol \(<\) is used in the access path. For example, consider the form Rectangle and its local form Center. The hierarchical relation of instances of these forms is illustrated in figure 5-1. Rectangle has parameters D and B. Center is a kind of Vector-xy with slots X and Y computed as functions of the parameters of Rectangle. The method for slot X in local form Center contains the query

\[(0 <::b)\]

which is a query directed at the slot B in the next higher level of the description hierarchy.

While the value query is the principal means of interaction between slots, some frame interactions may require that one frame impose a slot value in another frame. This can be done with the shell function Add-value. Slots that accept values from the Add-value function have the symbol \(??\) in the Value facet. The syntax of the Add-value function is

\[(\text{Add-value (id) (path) (value)})\]
The complementary shell function is `Delete-value`.

```
(Delete-value (id) (path) (value))
```

The main use of these functions is in procedures associated with the If-added and If-removed facets.

5.3 Summary

The description system employs two different modes of evaluating slots: forward chaining and backward chaining. The system evaluates parameter slots by forward chaining and property slots by backward chaining. In backward chaining a slot is evaluated only when its value is needed. The parameter slots of a new frame are always evaluated by forward chaining, even if the new frame is entered as a result of a backward chaining query.

Slot fillers are stored in the instance facet “Filler.” A single description may be an extensive hierarchy of nested instances. Any slot in the nested description can be uniquely identified by means of the access path. The access path is a list of the slot-name/slot-value pairs encountered in tracing through the hierarchy to the slot. The access path may include wild-card tokens in place of slot values. In the case of multiple-value slots the wild-card token is used to aggregate slot values.

The procedure to evaluate a slot value is stored in the slot facet “Value.” This facet may specify the form of the filler, or it may give a method for computing the slot value.

The user creates a new description with the shell function Create-instance. The user queries the description using the shell function Get-values. The slot query causes evaluation of the slot. The query can also be used to enter slot values. The asked-token prevents the description system from asking the user repeatedly for the same slot value.

The computations in the description system are data-directed, that is the computations are triggered by queries for the slot values. The computations themselves are based on an object-oriented approach to programming in which the individual frames in the description are computational objects consisting of both local state and local process.
Chapter 6
Integrity Maintenance

This chapter will look at the role of integrity maintenance in the description system. Integrity is “the maintaining of functionally related information so that the relations are satisfied” [Eastman78]. The integrity maintenance facility of the description system maintains pointers between functionally related data. These links form a dependency network along which data updates can be propagated.

Integrity is important to civil engineering design description because the description is continually being updated throughout the process of design. As it is essential that all data be kept accurate and up-to-date, some system for integrity maintenance must be in place. Because procedures are included as an aspect of knowledge within the description system, it is possible to establish integrity relations between individual slots, thus only those slots whose integrity is affected by a change are updated.

6.1 Data dependency

Before going on to consider integrity maintenance in the data-driven computing environment of frames, consider the case of a simple process-driven system.

The simplest case is that of a single application program with separate input and output files, as shown in figure 6-1. The data values in the output file are functionally related to those in the input file as determined by the intermediate program. Thus there is a data dependency between the two files—the output file is dependent on the input file. Unfortunately, it is not possible to be more specific than this. For the purposes of integrity maintenance the application program is a “black box.” Functional relations cannot be established between individual data items in the input and output files. The unit of integrity is the file as a whole. Any change to the input file violates the integrity of
Figure 6-1: An input and output file linked by a program.

the whole output file. The only way to restore integrity is to recompute the output file by running the intermediate program.

A more advanced design system may consist of a series of programs linked by datafiles—or, from the perspective of this discussion, a series of datafiles linked by programs. This arrangement, as illustrated in figure 6-2, is an extended dependency network. Change to any input file requires the recomputation of all dependent datafiles.

Figure 6-2: A system of linked data files and programs.

A yet more advanced system consists of a body of programs each linked to a common database, as shown in figure 6-3. The great advantage of this organization is data independence [Date86]. A further advantage is the opportunity to minimize redundancy and to put in place procedures for maintaining data integrity. Special measures must be taken to establish the dependency relation among individual data items. In particular, this involves additional features in the database for the storage of the dependency pointers [Fenves85]
Figure 6-3: A central database system.

[Lafue82]. Incorporation of integrity maintenance may extend the database beyond being a simple repository of data, to taking an active role in the evaluation of data values.¹

The system of frames shares some of the advantages of the central independent database. The chief of these is modularity. The modularity is achieved by creating independent computational objects that contain both data and process. Because, in the object-oriented world of frames, the unit of computation is the slot, the slot will also be the unit of integrity. The integrity model for frames is basically that of figure 6-2, with the simple change that the data files are in this case individual data items, i.e., frame slots, and the programs are individual slot methods. What might have been seen as a disadvantage of the frames system, that there is no fixed order of computation, becomes instead an advantage. Within the frames system integrity relations can be established between individual data items. When a data value is changed the dependent value can be restored by recomputing the intermediate process, which in this case is the method of the dependent slot. The update is strictly limited to those slots that are affected by the change. Whereas in a process-driven system the user would identify the process to be run again, in the data-driven system the act of

¹ “Possibly the most important objective of the DBMS [database management system] is to emulate, in the database, the processes occurring in design so that the database achieves an accurate representation of the physical structure. In particular, what is sought is a DBMS that can itself perform some of the duties of design, thereby automating to some degree the generation of design data values. To do so requires that the DBMS have knowledge of the constraints that govern the generation of those values. Such an association of properly invoked design constraints with a database results in a DBMS that actively contributes to the design process, thereby achieving a degree of design automation” [Reisner85].
changing a data value will itself activate the necessary methods to restore integrity. The key to integrity maintenance is the storage of the necessary dependency relations between data items which allow updates to be propagated from ingredient to dependent values.

6.2 Dependency links

The first duty of the integrity maintenance facility is to establish links between functionally related data. These dependency links must point both ways: any slot value must know both its ingredients and its dependants. The unit of integrity maintenance is the slot, hence the procedures for integrity maintenance must be implemented at the slot level. As the computations in the description system are based on queries for ingredient slot values, the dependency links are established as a side-effect of the slot query.

The dependency link is implemented as complementary pointers. The ingredient slot stores the name of its dependants in the instance facet “Cards” (so called because the name of the dependant is left like a calling card when a query is made). The dependent slot stores the name of its ingredients in the instance facet “Ingreds.”

In the description system, dependent slots are evaluated only to satisfy a query from the user. Having dependent values unevaluated is not considered a breach of integrity. It may be that Area = b \times d, but until the value of Area is computed, the functional dependence is not recognized, and the values of b and d can be changed without concern for integrity. However, once the functional relation has been established, integrity must be scrupulously maintained.

If a querying slot fails in its self-evaluation and hence has no value, it does not have a functional relation with its ingredients and the integrity link should not be established. The integrity relation links slot values, not slots; if there is no value there can be no link. However, in the present implementation, failed evaluations do have a value—the asked-token—and a dependence relation is created.

The primary use of the asked-token is to prevent repeat queries to the user for parameter values. When the user has been queried for a value and fails to return one, the null filler is changed to an asked-token. This token is for all practical purposes the same as the null value, except that it prevents the user from being asked again for the slot value. The description thus distinguishes between an unasked slot and a null asked slot.
The asked-token is also used by property slots. A queried property slot that fails to get a value takes the asked-token as a filler. The token means "the value of this slot cannot be evaluated." Obviously the truth of this statement depends on the values—or lack of values—of the ingredient slots. Should the ingredient values change and should it consequently become possible to evaluate the slot, this statement would no longer be true. The asked-token is therefore dependent on the ingredient values and a dependence relation is established. The value of the asked slot must be re-evaluated should any of these ingredients have a change in value.

The use of the asked-token in properties is important for efficiency. A failed evaluation may involve considerable backward chaining through many ingredients and so consume considerable resources. The asked-token prevents slot evaluations that are bound to fail.

While, in general, a dependency link is created only between a querying and a queried slot, there are two special cases in which a dependency link must be established with a slot other than the queried slot. The first case is when the queried slot does not exist. The second case is when the queried slot is a slot of a global instance.

Valid queries can only be directed at known properties of an instance. However it is possible that a valid query may be directed at a slot that does not exist. For example, the volume of a beam is computed as the beam's cross-section area multiplied by its length. The method for the slot Volume in the form Beam will contain the query (0 cross-section:area). If the slot Cross-section does not have a value, there is no slot Area to query. The slot Area will only exist when the slot Cross-section is filled. By linking the querying slot to the last existing slot along the access path—in this case the slot Cross-section—the querying slot can be notified when this slot is filled and the as yet non-existent slot—the slot Area—is brought into existence.

Another special case involves the use of a global instance as a slot value. The usual slot value is a local instance, an instance that has no existence separate from the slot it fills. When a slot containing a local instance is deleted the instance is also deleted; all its slots are deleted and all dependants of those slots are notified. If the slot contains a global instance, however, the global instance cannot be deleted, yet any dependants that have queried the instance through the deleted slot must be notified. To take care of this case,
an extra link is established between the querying slot and the slot containing the global instance.

6.3 Reduction

The reduction procedure reduces the level of specialization of an instance, that is, it changes an instance of a form to an instance of the AKO parent of the form. An instance of Square, for example, reduces to an instance of Rectangle. The reduction procedure is a way to relax specialization constraints. In reducing an instance of Square to an instance of Rectangle, the constraint that the depth of the shape must equal its breadth is removed. A description can be created as a highly constrained specialization with subsequent relaxation of the specialization constraints to allow substantial alterations in the description.

Reduction is a simple process of removing specialization constraints. In the current implementation these constraints are expressed functionally as procedures for computing the parameters of the parent form from the parameters of the specialization. If the specialization introduces parameters not shared with the parent form, these must be deleted. However, before they are deleted the content of these slots must be preserved by ensuring that the parameters of the parent form, which are derived from them, are evaluated and their values stored. The definition of the AKO relation states that the parameters of the parent form can be computed from the parameters of the child form. However, if a parameter in the parent depends on more than one parameter of the child, one or more of which have not been given values, there is no way to preserve all the parameter data when the instance is reduced.

Also of concern is the maintenance of integrity links. As all members of a class have consistent properties, dependants of the property slots of the reduced instance are unaffected. The property slots may however have a different set of ingredients. For example, the method for computing the property Area in the form Square may depend only on the parameter D \((\text{Area} = D^2)\), whereas the property method for this same slot in Rectangle will depend of the parameters D and B \((\text{Area} = B \times D)\). Furthermore, the slot B, which was dependent on slot D in the form Square, is an independent parameter in the form Rectangle. Any dependency link between D and B must therefore be severed when the instance is reduced.
An additional concern arises if the instance has a filler that is an instance of a local form which is not defined in the AKO parent. In this case the filler itself can be reduced to a form that is meaningful in the reduced instance. The filler may reduce to a global form or to an acceptable local form. Another case of concern is when a local form is redefined rather than inherited. For example, Rectangle redefines the local form Center, rather than inheriting it from its AKO parent Parallelogram. In reducing Rectangle to Parallelogram, rather than reducing Center to Vector-xy, Center’s AKO parent, the filler is replaced by a new instance of the form Center which is local to Parallelogram. Replacement in such a case better preserves the intent of the description.

6.4 Change propagation

When a slot value is deleted the slot reverts to its original unasked condition. Its many dependants no longer have valid values. There is no need to re-evaluate the dependants as they obviously will evaluate to a null value. Value deletion can easily be propagated along the dependency network. Deletion of a slot value causes the deletion of the whole network of dependent slots.

The user may delete only those values that the user has input. The user cannot directly interfere with the value of property slots. These slot are given values in accord with the slot methods.

Value deletion does not have to trigger slot re-evaluation. A value change however must do so. While a deleted slot will instruct its dependants to delete themselves, a changed slot should instruct its dependants to re-evaluate themselves. If the notification process operates in a depth-first fashion, the top slot will instruct the first of its dependents to update itself, and this dependent slot will in turn instruct its dependents before the initially changed slot moves on to instructing its next dependent. It is quite possible, even likely, that within the dependency network there will be shared dependants and merging branches. A single dependent slot may receive more than one instruction to update. The first update will be performed while some of its ingredients have not yet been notified of their inconsistency. The slot will therefore re-evaluate to an inconsistent value. When the still inconsistent ingredient finally does instructs it to re-evaluate it must go through the process yet again. If a slot has many affected ingredients it will have to re-evaluate on the instruction of
each one but will only attain consistency after the final re-evaluation. This obviously is an
inefficient and undesirable means of propagating value changes.

Instead, what is required is a breadth-first propagation of value changes. The immediate
dependent slots should all be re-evaluated before any one of them passes on the change
notice. As any slot receiving an update notice has no way of knowing when it is okay to
pass on the notice, any slot should actually receive two messages: the first message will tell
it to update its value; the second will tell it to propagate the change. In turn the slot must
first inform all its dependants in turn to update themselves, and then inform them all in
turn to propagate the change.

The value update process as implemented does propagate an inconsistency notice, but
it does not directly propagate the update notice. Rather it makes use of a "bulletin board"
for storing update notices. Instead of sending an update notice to the dependent slot,
the notice is posted on the bulletin board. Only those slots that do not have dependants
need to be posted. By evaluating the last-changed slots posted on the bulletin-board,
the network of inconsistent slots can be re-evaluated using the normal backward-chaining
mode of computing slot values. While the inconsistency noticed is propagated down the
dependency network, the update notice is propagated up the network.

6.4.1 Bulletin board

The idea of the bulletin board is presented by Lafue as a means of delaying integrity
checking. Having in mind the use of constraints for integrity maintenance in a relational
database, Lafue says:

A constraint need not be checked immediately after operations on instances of its depen-
dent variables, if the corresponding instances of its dependent variables are of no interest to the
current user/applications. Instead the checking can be delayed until the dependent instances
become of interest, i.e., are accessed, since they are the only instances to be affected by this
constraint. This approach . . . is called delayed integrity checking (and update propagation),
as opposed to the more traditional immediate integrity checking (and update propagation).

Lafue82]

The argument for delayed integrity checks is that it is not reasonable to insist on the
integrity of the database at all times during the design.
For instance, if one wants to modify or replace the supports of a beam in an engineering drawing, one can simply erase the supports without being obliged at first to erase the beam on the basis that in reality the beam cannot float in the air. What one wants is that only in the final drawing nothing represents a physical element floating in the air. Similarly, an integrity maintenance procedure specializing in checking statics in a spatial database should simply flag statics violations rather than strictly prohibit them. [Lafue78]

The bulletin board system presented by Lafue would post the names of changed data items on the board. When slots are accessed the bulletin board is checked to see if the names of any of the slots ingredients have been posted. If they have been, this may trigger an update of the dependent slot. Unfortunately, this system only triggers an update if the name of a direct ingredient is on the bulletin board. If the ingredient of an ingredient is changed, no update is triggered. In the data-directed computing environment of frames it is essential that the integrity system act dynamically in direct response to data changes. There is no option of scanning the database to check all integrity constraints at one time. Integrity must be performed at the slot level. Change notices must therefore be propagated immediately along the dependency network. However this does not involve “checking” the dependent values for violations. Such checking is in fact re-evaluation, and as such can be delayed, as in Lafue’s proposal, until the value is requested by the user.

Delayed re-evaluation does not have to be included as a special feature, as it is natural to the description system. In general, any value is evaluated only when its value is requested. The bulletin board plays no role in delaying evaluation. Rather, the bulletin board is needed to force early evaluation. The bulletin board contains the names of the slots affected by a change but not themselves having dependants. Re-evaluation of these slots will restore all violated values. The user may instruct the system to query these posted slots. Otherwise, evaluation will wait until a value is requested in the normal way.

One important advantage of not re-evaluating slots immediately is that if the user is performing many changes, re-evaluation will be suspended until all the changes are made. In this way, re-evaluation can be done once for all changes. Thus if the user is making

[2] ... undetected violations may occur due to the fact that records are directly sensitive to changes in their parents only, not in their further ancestors. For example, suppose beams depend on the columns which support them. Suppose further that a column is deleted, and then a pipe supported by the beam which is itself supported by that column, is accessed before the beam. The pipe does not know that its supporting beam is no longer supported. Special provisions for sensitivity to changes in ancestors must be offered if specifically requested. [Lafue82]
many changes to the data, the bulletin board will be cleared only when all these changes are complete. It is not essential even to force an update as this will occur the next time a violated slot is queried.

6.5 Summary

The maintenance of data integrity is essential to the frame-based description system. In this system there is no fixed order of computation. Slot values are computed dynamically in response to queries to the description. The description must handle the continual input of new data values and changes to existing values. To deal with this fluid situation the system keeps track of the dynamic dependency relations between individual data values. As the unit of computation is the slot, so too is the unit of integrity. The procedures for data integrity apply directly to the frame slots. The dependence link is bi-directional, from the ingredient to the dependant, and from the dependant to the ingredient.

The principal integrity operation is the deletion of a parameter value. The deletion is propagated along the dependency network, deleting in turn all dependent values. Deleted slots return to their original "unasked" conditioned. Deleted slot may be re-evaluated as the result of a query for a property value, or by clearing the bulletin board of posted change notices. The description system normally delays re-evaluation of slots until their values are requested. Clearing the bulletin board forces early evaluation of the deleted slots. The integrity maintenance system ensures that all existing values are valid.
Chapter 7
Defaults

This chapter will look at the use of default values in the description system. The description clearly distinguishes default values from values derived from input data. Default values are evaluated by default methods in response to specific default queries.

Default values are important in a civil engineering design description because the description will be incomplete until the end of the design process. It is often useful to anticipate the design completion with reasonable temporary values. These values however cannot be treated as input values. There use is strictly limited to providing a semblance of completeness to the description. Engineering description thus requires that these default values be recorded as a separate system of values. Because they do not have the same requirements for accuracy as the final design values, the default values can be computed using less rigorous procedures. In fact, several different approximate procedures can be used depending on the available ingredient data.

7.1 Soft defaults

A default is “a failure to fulfill an obligation” [Oxford83]. The user of a program has a responsibility to supply all necessary input values. Should the user default on this responsibility, the program might carry on, none the less, with internally generated values; such values are “default values.” If the program proceeds to use the default value in the same way that it would have used the input value, the default value is a “hard” default value.

The alternative to a “hard” default is a “soft” default. A soft default value is a value that is clearly distinguished from an input value or a hard default value.

Dependent values that are derived from default values must also be considered as default values. Values therefore are of two kinds: hard values (derived from input values) and default values (derived in whole or in part from default values). Clearly, any request for a value must indicate whether a hard value or a default value is to be returned. Which kind of value is requested will depend on the intended use of the value.
Consider, as an example, the situation in which the user has created an incomplete description of a structural frame and wishes to use a 3-D modeller to display the structure. In this example, let us say, the description contains all the pertinent information describing the geometry of the frame except the cross-sectional dimensions of the individual frame members. The user might find the display to be of use even if the individual member sizes are inaccurate; unfortunately, the modeller needs a complete input file. This situation is one in which default values can play an important role; the place of the missing member sizes can be taken by reasonable default values, and a complete input file can be constructed for the modeller. The display, in this case, is a default display and must be interpreted as such. The default values supplied by the description system are used to give an illusion of completeness to the description.

As the purpose of the default value is to take the place of a value the user has failed to provide, and as the user provides values only to parameter slots, default values may be reasonably limited to parameter slots. Given default parameter values, default values of all dependent slots could be evaluated using the slot value methods. However, the value method of a property slot will not always be appropriate for computing the default value. As default values do not have to be accurate, default methods may use less rigorous computations than the value methods or rely on fewer ingredient data. There is every reason therefore to include in property slots default methods that are different from the value methods.

In the example of the structural frame, the basic default for member size may be a single fixed value, in which case all members of the default structure will be of the same size. A better default—one that makes full use of available input values and is a closer approximation to the final design value—would be a method that chose member size as a function of member length. This default method uses a rule-of-thumb of a kind that is common in the preliminary stages of design.

It should always be possible to evaluate a default. Since the best defaults depend on input values which might or might not exist at any given time, it will often not be possible to evaluate these defaults. In anticipation of this possibility, the description system accommodates alternative defaults. If a more accurate default method cannot be evaluated, a less accurate one can be used in its place.
The default methods are attached to the form slot in the Default facet. As with the Value facet, there are several ways of indicating the value: the default value may be a numerical value; it may be a global or local form; or it may be a method. All the options available for designating the slot value are also available for the slot default, with one important exception: the default method cannot query the user. The alternative default methods are ordered as they are to be evaluated, the better methods being evaluated first.

Default values must be substantial values—either simple numerical or logical values, or fully instantiated instances. The default value must not result in further queries to the user. It is not difficult to see why this is so. For example, if the user defaults on a query for a beam cross-section and the description uses Rectangle as a default value, the system should not ask the user to supply values for rectangle depth or breadth+1; if the user cannot decide on the shape of the section, the user is unlikely to be able to supply more detailed data values.

The Default facet in the form stores the method for computing the default value. The default value itself must be stored in the instance. Rather than creating a new facet to store default values, the defaults values are stored in the Filler facet.

There are two main reasons for putting defaults in the Filler facet. The first concerns nesting within the description hierarchy and the access path. Instances are nested in the Filler facets. The designation of the access path assumes this, and for that reason the access path does not include a facet key. If defaults were included as a separate facet, this would introduce a second nesting path. Also, as the system must ignore any default value if a hard value exists, either one or the other will be accessed, but never both; one or the other will therefore be redundant at all times.

The second reason involves multiple-value slots. Putting defaults in the Filler facet offers the possibility of mixing hard values and default values within the slot. The Filler facet might contain an entry of the type

\[(\$filler (value) (default) (value))\]

where one of several fillers is a default value. This would not be possible with a separate instance facet for default values.
Default values in the Filler facet are clearly identified as such. The description system recognizes several kinds of "special fillers." These special fillers are simply lists, the first element of which is a recognized key. Default values are identified by putting them in the list with the key "default."

If a value query fails the asked-token is put as the slot value. If a default query is then made on the same slot the default value will displace this asked-token. As the role of the asked-token is to prevent re-evaluation of the slot in any subsequent value queries, the default value must do the same. The asked-token includes a timestamp. At the users option, tokens created before a certain time can be ignored, and so not prevent re-evaluation. The default value must therefore also include this timestamp. This allows the default value to be cleared so that the slot can be re-evaluated for a hard value. The default value is therefore of the type

\[(\text{default \ (value) \ (r)})\]

When the timestamp, \(r\), predates the latest clear command it is treated as a null value. If the default itself cannot be evaluated it must itself take an asked-token as a value. (This situation is undesirable and will not happen if all defaults have substantial values, but in some circumstances it is inevitable.) As the default value already includes the timestamp, there is no need to include another with the asked-token, so the asked default is of the type

\[(\text{default \ asked \ (r)})\]

In summary the various types of values that a slot can take are:

\[(\text{value});\]

\[(\text{asked \ (r)});\]

\[(\text{default \ (value) \ (r)}); \text{ and}\]

\[(\text{default \ asked \ (r)}).\]

Since hard values and default values are clearly identified, the user in making a request for a value, must indicate whether a hard value or a default value is to be returned. To do this the user either uses the shell function Get-value or the shell function Get-default. The shell function Get-value interprets default values as the asked-token and returns nil if a default is encountered anywhere along the access path. The shell function Get-default,
on the other hand, treats hard values and default values the same. The shell function Get-default will return the best value that can be evaluated; if a hard value is available it is returned; if a hard value cannot be returned the best default value is returned.

The shell function Get-default is also responsible for extending the description when slots along the access path have not yet been evaluated. Clearly, these slots must first be evaluated for hard values, and only when such evaluations fail are the default methods to be tried. The shell function Get-default therefore must do all that the shell function Get-value does and handle defaults as well.

The syntax of the default query is similar to that of the value query. The general syntax is:

\[(\text{Get-default} \ (\text{instance-id}) \ (\text{access-path}) \ (\text{querying-slot}))\]

and

\[(\text{Get-defaults} \ (\text{instance-id}) \ (\text{access-path}) \ (\text{querying-slot}))\]

Within slot methods the syntax is:

\[\text{"\@ (colon-path)"}\]

and

\[\text{"\@@ (colon-path)"}\]

These queries within methods are expanded to the general syntax by expanding the colon-path to a proper access path, and splicing on the missing arguments from the context of the query. "\@" is expanded to the shell function Get-default, and "\@@" is expanded to the shell function Get-defaults.

The description system also recognizes another special default method. The default method of a slot will often be identical to the value method. For example, in the form Rectangle the value method for Area is

\[\text{Glet} \ (\ (d \ (@ d)) \ (b \ (@ b)) \ ) \ (\ * \ d \ b \ )\]

The \texttt{GLet} creates local variables \texttt{d} and \texttt{b} that take the values of slots \texttt{D} and \texttt{B} of the form Rectangle.
The default method for the same slot is

```lisp
(\let ((d (\* d))
       (b (\* b)))
       (\* d b))
```

The only difference between this method and the value method is that default queries are used in place of the value queries. Because default methods are often a restatement of the value methods, the description system offers a way to borrow the value method for use in computing default values. In the default facet the token "\*value" signifies that the value method is to be used with the value queries within the method changed to default queries. The slot for Area in Rectangle may therefore be defined as

```lisp
(area
 ($value (\let ((d (\* d)) (b (\* b)))
       (\* d b)))
 ($default \*value))
```

In this example, little is gained by the use of the \*value token, but the more complex the value method, the more useful the \*value token becomes. The \*value token can be used as the first default, as an alternative default, or not at all. The use of the token is simply a shorthand way of writing default methods. If the default method is a restatement of the value method with some, but not all, of the value queries changed to default queries, the default method must be written out in full.

There is no restriction on the use of value and default queries in either value or default methods. Either method may make use of either type of query. The rule of inheritance with regard to defaults is that defaults are inherited unless overwritten. The \*value token, even if inherited, is always an expansion of the current value method, whether inherited or overwritten.

### 7.2 Integrity maintenance

The previous discussion of integrity maintenance (in Chapter 7) did not consider the integrity of default values. The integrity maintenance system for defaults works much the same as it does for hard values, but must handle complications imposed by the use of alternative default methods.
A default query links the querying and queried slot via complementary pointed stored in the Cards and Ingreds facets. The Default facet may contain several alternative default methods. These are stored in the facet in order of use. In processing a default query the system will first try to get a hard value. If a hard value cannot be evaluated, the system will try to evaluate the first default method. If this evaluation succeeds the system will ignore the other default methods. If the first default method fails, the system will go on to try the next method, and so on until all alternative methods are exhausted. As each default method is evaluated, the slot is linked to the ingredient slots of that default method. It may be that the different defaults have different ingredients, in fact this must be so, for if they all had the same ingredients they would all succeed or fail together.

The goal of the integrity maintenance system is to maintain the best possible slot value. The links to ingredients of the failed queries are maintained so that if any of these ingredients should subsequently become available, the dependent slot can be updated. However, when the dependent slot is updated it must remove all links to its previous ingredients. This prevents an update of an ingredient of a lower ranked default causing the update of the value of a higher ranked default. While in most cases, this unwanted update would only result in a redundant evaluation, if the update displaces an input value, this value cannot be recovered. Valid links, removed as part of the update procedure, are re-established in the subsequent re-evaluation of the slot.

The shell function Serve-change-notices, as previously introduced, queries for the values of the slots posted on the bulletin board. If a deleted slot has no dependants and has a default value, the slot is posted, but identified as a default slot, so that in re-evaluation the slot will be queried with the shell function Get-defaults rather than the shell function Get-values. Thus all values in the description are restored, whether hard values or default values.

7.3 Summary

Although the description created by default values will in no sense be an accurate description it still has important uses. The concept of top-down design holds that at all stages of design the description should represent a complete artefact. The description system itself represents the full set of possible complete descriptions; the default system is
a means of choosing one possible description as a representative of the set of possibilities. Although it would be reasonable to restrict defaults to parameter slots, the implementation allows the more general case of defining defaults on property slots as well.

The defaults are implemented as soft defaults, that is, they are clearly identified as defaults. A query for a slot value must indicate whether a hard value is to be returned, or whether the best possible value—which may be a hard value or a default value—is to be returned. Default values are computed by default methods. A default method may query for either hard values or default values.

Because defaults must be substantial fillers, i.e., they must be fully instantiated instances, the system allows alternative default methods. Thus, the slot may contain defaults which depend on parameter values, along with alternative substantial defaults. Default methods are evaluated in the order in which they are listed in the Default facet.

The integrity maintenance system maintains the best default value. Both value methods and default methods create dependency links, even though these methods fail. When a slot is updated these links are removed. Subsequent re-evaluation of the slot restores valid links.
Chapter 8
User Interface

This chapter will look at the user interface of the description system, and an important part of that interface, the explanation facility.

The user interface is an important part of any interactive computer program. The user and the program are partners in a process that requires the exchange of information. The user interface is the medium of this exchange.

To use a program the user must form a conceptual model of the program. The user interface can greatly aid the formation of this conceptual model. Rubinstein calls the model of the program presented by the user interface the "external myth":

To ensure ease of learning, a system must have consistent external behavior—behavior independent of its internal workings. We like to refer to this concept as the presentation of a consistent external myth. We call it a myth to emphasize that what the user actually sees may not relate directly to the internals of the system... If the external myth is familiar, users can make predictions, extrapolations, and educated guesses about the behavior of the system. We must see to it that the external myth meticulously supports the building of simple conceptual models that correctly predict the behavior of the system. [Rubenstein84]

The user interface must be designed therefore to portray the mythological workings of the program. Being unable to compare the external myth to the underlying reality, the user will accept the external myth as the truth, but only as long as it is consistent. A successful user interface will not disillusion the user.

---

1 "A high level 'conceptual model' of the interface is of major importance to the users' ability to learn an interface and to their comfort and efficiency in using it... If the interface either encourages formation of a conceptual model, or at least doesn't discourage formation of such, the users' ability to learn and use the system is enhanced." [Hendler88]

2 "The user of an interactive computer system brings a complex range of psychological skills and attributes to bear on his/her interactions with the system such as knowledge, experience, motivation and personality. Effective interactive communication is seen as requiring the user to learn, assimilate and structure relevant information concerning the system and its uses into a meaningful conceptual model of operation. The question of how such knowledge is acquired and the influence of particular interface styles on this knowledge acquisition are essentially psychological in nature, and of central importance for the successful design of usable or 'user-friendly' systems." [Dillon87]
The user interface, in even the simplest of programs, will provide some guidance and direction to the user in the form of input prompts, error messages, and comments. On-line help is an increasingly popular feature in programs. It is a feature that not only helps the novice user become acquainted with the program but also assists the experienced user in making the most efficient use of the program. On-line help can be implemented in the interface quite separately from the program itself. The help provided by such a facility will be of a general nature and independent of the context of the interaction from which it is invoked. A help facility that is actually built into the program, on the other hand, can offer help that is tailored to the specific context of the current interaction. But whether the help facility is built into the program or not, it is important that the current context of the interaction between the user and the program be preserved while help is being sought [Tesler81].

The conceptual model of the description system that is promoted by the user interface of the description system is the image of the form as a questionnaire with blanks to be filled in. Each instance in a description can be viewed as a separate questionnaire, one that can be used to fill a blank in another. The final description is a sheaf of these questionnaires with the blanks filled in, each a separate instance, but together forming a single description hierarchy.

The interaction between the user and the system is carried on through the exchange of queries: the user queries the system for property values; and the system queries the user for parameter values. The user builds up a description by replying to the queries posed by the system.

8.1 Forms and instances

The first thing that the explanation facility of the description system must do is provide the user with a display of the forms and instances defined in the system. In doing this, the explanation facility of the description system is similar to that of the expert system which provides the user with rule definitions and facts, and it is also similar to the interface of the object-oriented programming language which provides the user with browsers for classes and instances.
The forms present no problem to display as they are adequately defined in the knowledge base "as is." Access is a matter of interactively displaying the definition that already exists in the knowledge base. In fact, any display that did not directly correspond to the user definition would violate the consistency of the external myth. The layout of the form is that of figure 5-3. The external and internal representation of the form are basically the same.

Instances are created cooperatively by the user and the description system. The display of the instances is limited to the slot values. The fillers of the other facets are not displayed. Access to all of the other facets is possible by using the appropriate shell function.

The display of the instance therefore consists of slot names and their corresponding fillers. The slot filler may be a numerical value, an instance, or a special filler. If the filler is a numerical value, the value is shown. If the filler is a global instance, the instance name alone is shown. If the filler is a local instance, the entire instance is shown, appropriately indented. If the filler is a default value it is flagged as a default. If the slot has been asked the filler is the asked-token. If the filler is any one of the other special fillers it is represented by a special symbol—for example, the CSG-list in forms for 3D shapes is represented as "** CSG **".

In adherence to the conceptual model, each instance within the description should be displayed separately. However, it would be awkward to have to call up individually each local instance in a description. The display of an instance therefore includes all the local instances. The nesting of these local instances within the description hierarchy is indicated by progressive indentation. Global instances are displayed on their own. Transition slots, being used for internal computations only, are not included in the display.

8.2 Menus

To fill the parameter slots, and hence to complete the description, the description system queries the user for values. A query consists of a simple prompt for a value. The prompting for parameter values offers an excellent opportunity to include some in-context help to the user.
The prompt itself will necessarily be terse; it may in fact be some unfamiliar abbreviation. A fuller explanation is contained in the Defn facet. This facet holds a textual definition of the slot.

The Menu facet is a list of all the acceptable classes of fillers for the slot. The slot will not accept a filler that does not belong to the class of good fillers specified in the Menu facet. The user has a genuine interest in being able to consult this list of good fillers. This list however is incomplete in the sense that it does not specifically include the sub-classes of the good fillers which are also acceptable fillers. The explanation facility makes good on this shortfall by compiling a complete menu of good fillers based on the AKO hierarchy and presenting it to the user. Included in the menu are the local forms defined in the context of the current slot and global instances. The compiling of the menu of good fillers, owing to the inclusion of local forms, can only be performed in the context of the current interaction.

In the menu of good fillers, the hierarchy of fillers is indicated by progressive indentation, the sub-classes being indented relative to their parents. Each item in the list of good fillers is given a reference number. The user may make a selection from the menu of fillers by entering the number, rather than by typing in the name. Also, each item is identified as a form, a local form or an instance.

As the names of the fillers may themselves be unfamiliar to the user, the user may call up a definition of any of the fillers in the menu of good fillers. This is done by selecting the filler by reference number. Each form and instance has an association that holds a textual definition. This definition is displayed to the user upon request.

To make access to help as easy as possible the ask-user function that prompts the user for the parameter value also handles requests to the explanation facility. The general request to provide more information about the slot is made by entering a single question mark at the prompt. The explanation facility responds by giving the slot definition, the menu of good fillers, and the default value if it is not a process. After providing the user with this information, control passes back to the ask-user function, which repeats the prompt. In general, all requests to the explanation facility that are given at the ask-user prompt are identified by being prefixed by a question mark. Longer requests to the explanation facility can be entered as a list whose first element is a single question mark. After each request the system returns to the prompt so that the context of the interaction is not interrupted.
8.3 Summary

The user interface presents a consistent model of the workings of the description system to the user. Description is based on the idea of abstraction, and is represented as a hierarchy of nested instances. The interaction between the user and the system is conducted through the exchange of queries. This style of interaction suits the data-driven style of computation employed by the description system.

The most practical help the explanation facility can give the user is, first of all, access to the form definitions and instances. This access is essentially free of the context of the interaction between the user and the system. Instances are displayed in their entirety as lists of attributes with values. Values that are "\"resolved\" local instances are also shown in their entirety but indented to indicate their subordinate position in the description hierarchy. Defaults and other special fillers are indicated as such.

The explanation facility assists the user in selecting slot values by providing explanatory text and a complete menu of good fillers. This help is provided in the context of the interaction. The menu of good fillers includes local forms and global instances. The explanation facility also offers explanatory text on each item in the menu. Requests to the explanation facility are made at the system prompt without interfering with the current interaction.
Chapter 9
Demonstration

This chapter presents a demonstration of the features of the description system. The principal shell functions by which the user interacts with the description system—Define-form, Create-instance, Update-value, and Reduce—are demonstrated, and some aspects of the internal representation of forms and instances are examined. The demonstration includes the definition of selected forms which are part of a knowledge base for the description of building structures. Within this knowledge base, the computation of the geometric properties of 2D and 3D shapes is defined as an internal application. An interface to a 3D modeller is defined as an external application.

9.1 Forms

A form is defined using the shell function Define-form. The syntax of this function is

```
(define-form `(form-name)
  (ako <form-name>)
  (defn "<short textual definition>")
  (slots
    (slot-name ($value <filler>)
      ($defn <filler>)
      ($menu <filler>)
      ($default <filler>)
    )
    (slot-name ($value <filler>)
      )
  )
  (methods
    (method-name <procedure>)
    : 
    (method-name <procedure>)
  )
  (local-forms
    (form-name
      (ako <form-name>)
    )
    (form-name
      (ako <form-name>)
    )
  )
)
```

86
Figure 9-1: The parameters of the family of Quadrilaterals.

A form is a definition of a generic object, i.e., a class definition. The form consists of an indefinite number of slots, each of which represents a single attribute of the class. Each slot consists of a number of facets, each of which represents some special knowledge about the slot. Procedures for computing slot values may be collected in the association Methods. Within a form, local forms may be defined in the association Local-forms. The definition of a local form is identical to that of a global form, but may only be accessed through the superior form. Local forms may have their own local forms.

Consider the definition of the form Quadrilateral. Referring to figure 9-1, the parameters of Quadrilateral are b, d1, dr, a and c. The Value facet of each of these parameter slots will contain a single question mark, signifying that each of these slots is a single-value parameter slot. The Menu facet will indicate that the good filler for each of these slots is a number. The Defn facet will contain a short definition of each slot. The typical parameter slot is:

\[
(b \text{ $(\text{value } ?)$)} \\
($\text{menu number}$) \\
($\text{defn "The length of the base"}$)
\]

The value of the property Area is computed from the parameter values. The formula for computing the area of Quadrilateral is:

\[
\frac{d1(a + c) + dr(b - a)}{2}
\]
Or, as it is written in Lisp's prefix notation:

\[
(/ (+ (* dl (+ a c)) (* dr (- b a))) 2)
\]

Unfortunately, the Lisp functions /, *, +, and - do not accept slot names as arguments. Arguments to these functions must be the slot values. The slot values are obtained by querying the slots with the shell function \texttt{Get-value}, represented in methods by the symbol \texttt{\&}:

\[
(/ (+ (* (\& dl) (+ (\& a) (\& c))) (* (\& dr) (- (\& b) (\& a))) 2)
\]

The procedure expressed in this way, however, has an undesirable feature. Should any of the queries be unsuccessful and return a null value, an error will be triggered by the Lisp math functions, which accept only numbers as arguments. To avoid this problem, the queries are evaluated before passing their values on to the mathematical functions, using a special Let construct called @Let. The @Let consists of two parts. The first is an assignment of local variables. The second is the procedure to be evaluated. If any of the local variables evaluates to nil, the procedure is not evaluated and nil is returned as the value of the @Let. Local variables may retain the names of the slots whose value they represent.

An acceptable method for Area is:

\[
(@let ((b (\& b))
(dl (\& dl))
(dr (\& dr))
(a (\& a))
(c (\& c)))
(/ (+ (* dl (+ a c)) (* dr (- b a))) 2))
\]

In queries to multiple-value slots, the @Let construct is not sufficient, as the multiple-value query returns a list of values. For example, the aggregate form \texttt{built-up-shape-2d} computes Area as

\[
(apply + (\& items:area))
\]

The query \texttt{(\& items:area)} returns a list of the areas of the component items. The function + is then applied to this list giving the sum of the areas. If one or more values in
the returned list of areas is nil, an error results. In this case the @Apply construct can be used. The above method for Area would be

```scheme
(apply + (get items:areas))
```

Instead of passing null arguments to the function +, the @Apply will immediately return nil. @Apply may be used in conjunction with @Let.

A partial definition of the global form Quadrilateral may be written using the function Define-form as follows:

```scheme
(define-form 'QUADRILATERAL
  (ako shape-2d)
  (defn "A quadrilateral")
  (slots
   (b ($value ?)
     ($menu number)
     ($defn "The length of the base"))
   (dl ($value ?)
     ($menu number)
     ($defn "The height of the left apex"))
   (dr ($value ?)
     ($menu number)
     ($defn "The height of the right apex"))
   (a ($value ?)
     ($menu number)
     ($defn "The horizontal offset of the left apex"))
   (c ($value ?)
     ($menu number)
     ($defn "The horizontal distance between apexes"))
   (area
     ($value get-area)
     ($menu number)
     ($defn "The area of the shape")))
  (methods
   (get-area
     (get @let ( (b (get b))
        (dl (get dl))
        (dr (get dr))
        (a (get a))
        (c (get c)) )
        (/ (+ (* dl (+ a c)) (* dr (- b a))) 2))))))
```
Quadrilateral is defined as a kind of the form Shape-2d. The form Shape-2d is in turn a kind of the form Entity. Both of the forms, Entity and Shape-2d, are incompletely defined. They are included in the knowledge base to link all forms in a single hierarchy. The form Entity is proposed as the top of the hierarchy of all forms. The form Shape-2d is included in the expectation that 2D shapes other than Quadrilateral will later be added to the hierarchy.

9.2 Instances

Instances are created using the shell function Create-instance as described in chapter 5. An instance of the form Quadrilateral is created as follows. In all following interactions user input is underlined. The numbers in square brackets are the Scheme prompt.

```
[1] (create-instance 'q1 'quadrilateral)
Instance defn (string): "Example quadrilateral"
B: 100
DL: 25
DR: 50
A: 25
C: 50

*************
*    Q1    *
*************
Q1 <aio QUADRILATERAL>
B: 100
DL: 25
DR: 50
A: 25
C: 50
AREA:
()
```

It can be observed in the above interaction that the description system automatically queries for the values of the parameters of the instance. A display of the instance is included at the end of the Create-instance interaction. As the Area slot has no value, it is simply left blank in the display. When a slot value in a display is an instance, the name of its form is displayed beside it in angular brackets along with the key "aio" (an instance of).
Chapter 9: Demonstration

The user can get the value of any slot by addressing a query to the slot. If the slot does not already have a value its value will be computed as directed in the Value facet. For example, to get the value of the slot Area of the above defined instance of Quadrilateral:

```
[2] (get-value 'q1 '((area *)) nil)
2812.5
[3] (show-instance 'q1)
***************
  Q1  *
***************
Q1 <\i> QUADRILATERAL>
B: 100
DL: 25
DR: 50
A: 25
C: 50
AREA: 2812.5
()
```

As the function Get-value does not automatically produce a display of the instance, the user must call up the display with the function Show-instance. As can be seen in the display, once a slot value has been evaluated it is stored for future use. Subsequent queries to the slot do not cause re-evaluation of the slot but simply return the stored value.

### 9.3 Specialization

Given the form Quadrilateral, further forms can be created by specialization, using the AKO relation. For example, Trapezoid is a kind of Quadrilateral. In terms of the parameters of the Quadrilateral, a Trapezoid is a Quadrilateral with dl=dr. Or if a new parameter is created for Trapezoid, a Trapezoid is a Quadrilateral with dl=dr=d:

```
(define-form '(TRAPEZOID
  (ako quadrilateral)
  (defn "A trapezoid")
  (slots
    (d ($value ?)
      ($defn "The height of the shape")
      ($menu number) )
    (dl ($value (\i d))
      ($transition? #T))
    (dr ($value (\i d))
      ($transition? #T)) )))
```
In this form the Transition facet (Transition? #T) indicates a slot that is used in the internal computations of the form but is not displayed by the display function Show-instance. Through the AKO relation, Trapezoid inherits the parameter and property slots of Quadrilateral.

\[[4] \text{ (create-instance } 't1 '\text{trapezoid)}\]
\[\text{Instance defn (string): "Example trapezoid"}
\[\text{D: 50}
\[\text{B: 100}
\[\text{A: 25}
\[\text{C: 50}
\[***************
\[ * \quad T1 \quad *
\[***************
\[T1 <\text{aio TRAPEZOID}>
\[\text{D: 50}
\[\text{B: 100}
\[\text{A: 25}
\[\text{C: 50}
\[\text{AREA:}
\[()\]
\[[5] \text{ (get-value } 't1 '((area *)) nil)}\]
\[3750\]
\[[6] \text{ (show-instance } 't1)\]
\[***************
\[ * \quad 1 \quad *
\[***************
\[T1 <\text{aio TRAPEZOID}>
\[\text{D: 50}
\[\text{B: 100}
\[\text{A: 25}
\[\text{C: 50}
\[\text{AREA: 3750}
\[()\]

Further specializations of Trapezoid are likewise possible. For example, Parallelogram is a Trapezoid with \(c=b\), Rectangle is a Parallelogram with \(a=0\), and Square is a Rectangle with \(b=d\). (The definitions of these forms are included in Appendix A.)

Using the AKO relation, a new form can be created by constraining some parameter of an existing form. These new forms can rely entirely on the property methods of their parent forms. For example, the entire family of specializations of Quadrilateral, as defined above, is dependent on the single inherited method for computing area.
9.4 Aggregation

While specialization is important in creating new forms it is, by its nature, limited. Aggregation, on the other hand, has no such limits. Consider an aggregation form for 2D shapes. This form, Built-up-shape-2d, will combine any number of existing shapes to create a new shape. To specify the aggregate shape requires a shape and location for each component. The slots for shape and location will be linked together by using the local form Item. Leaving location out of consideration for the time being, for the sake of clarity, the form Built-up-shape-2d can be partially defined as:

```
(define-form 'BUILT-UP-SHAPE-2D
  (ako shape-2d)
  (defn "A 2d shape built up from other 2d shapes")
  (slots
    (item
      ($value ??)
      ($defn "A component of the built up shape")
      ($menu item))
    (area
      ($value get-area)
      ($menu number)))
  (methods
    (get-area
      (0let ( (areas (00 item:cross-section:area)))
        (0apply + areas)))
  (local-forms
    (item
      (ako entity)
      (slots
        (cross-section
          ($value ??)
          ($menu shape-2d)
          ($defn "The cross-section of the part")))
    )))
```

The value "??" in the Item slot indicates that this is a multiple-value parameter. The fillers must be instances of the local form Item. The local form Item consists of the single slot Cross-section. The area of the built-up shape is the sum of the areas of the item cross-sections. Including location of the component items permits computation of many other section properties, such as the location of the centroid and moments of area.
Figure 9-2: The parameters of the form Wshape.

Adding Built-up-shape-2D to the existing 2D shape forms allows the building of any shape made up of Quadrilaterals. Consider the aggregate shape Wshape (wide-flange shape). The parameters of the form Wshape are shown in figure 9-2.

(define-form
  '(WSHAPE
    (ako built-up-shape-2d)
    (slots
      (d ($value ?)
        ($menu number)
        ($defn "depth of section")
      )
      (b ($value ?)
        ($menu number)
        ($defn "flange width")
      )
      (w ($value ?)
        ($menu number)
        ($defn "web thickness")
      )
      (th ($value ?)
        ($menu number)
        ($defn "flange thickness")
      )
    ))
This form inherits the slots Item and Area from its parent form Built-up-shape-2d, and introduces the new parameters d, b, w, and th. The slot Item, which is a multiple-value parameter slot in the parent form, is given fixed values in the form Wshape; it is filled with instances of the local forms Top-flange, Bottom-flange and Web. Each of these local forms defines its Cross-section as a Rectangle, with dimensions computed from the newly introduced parameters of Wshape.
This definition of the form Wshape is somewhat involved since the parameters of the component shapes, nested in two levels of local forms, must be defined in terms of the parameters of Wshape. The symbol "<" in the slot queries is an indication that the query is addressed to the superior level of nesting in the description hierarchy. As Wshape has local forms within local forms it is possible for a query to be addressed to a slot two levels of nesting above the querying slot. The first component item of the Wshape is an instance of the local form Top-flange. The Cross-section of Top-flange is another local form Rectangle1. The depth of this rectangle is the flange thickness of the Wshape. The computation for slot D in local form Rectangle1 is therefore directed at the slot th of Wshape, two levels of nesting above: (d (value (0 <:: th))).

The Area of a Wshape is computed by the property method inherited from Built-up-shape-2d as the sum of the Areas of the component Rectangles. Each Rectangle, in turn, inherits its Area method from the parent form Quadrilateral. It should be noted that Quadrilateral itself can be defined as a Built-up-shape-2d consisting of two Triangles—it is defined in this way in Appendix A. Redefining Quadrilateral as a built-up shape in no way alters the definition of its specializations or of the form Wshape. Definition of the two forms Triangle and Built-up-shape-2d allows the creation of any shape that can be defined as a union of triangles. All computations of the properties of these shapes can be inherited.

9.5 External applications

The computation of the areas and volumes of planar and solid shapes has been developed in the knowledge base as an internal application. There is also a need for higher level applications that cannot be directly incorporated in the description system. The description system must be able to work with external applications.

One application that is particularly useful and illustrative in the description of physical artefacts is the three-dimensional graphic display. Due to the complexity of this application, no attempt has been made to include 3D modelling as an internal computation of the description system; rather, the description system relies on the services of an external modelling program.
9.5.1 Constructive solid geometry

The solids modeller chosen to work with the knowledge base is PADL-2 [Hartquist83], a constructive solids geometry (CSG) modeller. In PADL-2, complex solids are defined as combinations of a small set of primitive solids consisting of Block, Wedge, Cylinder, Cone, Sphere and Torus. Three of these primitives are shown in figure 9-3. There are two kinds of operators that can act on defined solids: the movement operator allows the positioning of solids; the set operators—union, difference and intersection—make possible the combination of solids into more complex shapes. A CSG definition takes the form of an ordered binary tree; this is illustrated in figure 9-4.

The coordinate axes defined in the knowledge base follow the PADL-2 convention: axes are arranged with the Y-axis vertical and signs for rotations follow the right hand rule.

Because, in the description system, computations are data-directed, an internal representation of the CSG-tree is computed as the value of a property slot. This property slot, Csg, is included in the form Shape-3d. The internal CSG-tree is represented as a nest of lists. A displayable CSG representation, the CSG-list, is specified as follows:

\[
<\text{csg list} >= (\text{CSG} <\text{primitive solid} | \text{set operation} | \text{csg list} \text{ AT <motion arguments>})
\]

The representation of the primitive solids is similar to that of PADL-2:

\[
<\text{primitive solid} >= (\text{BLO x y z})
\]

\[
(WED x y z)
\]

\[
(CYL h d)
\]
Figure 9-4: A constructive solid geometry tree (after Requicha)

The set operators are represented by:

\[
\begin{align*}
<\text{set operation}> & = (\text{UN} \; <\text{csg list}> \; <\text{csg list}> ) \\
& \quad (\text{DIF} \; <\text{csg list}> \; <\text{csg list}> ) \\
& \quad (\text{INT} \; <\text{csg list}> \; <\text{csg list}> )
\end{align*}
\]

It should be noted that the primitive solids by themselves cannot be displayed. The only displayable representation is the CSG-list which includes a motion operator. Because the description system has its own default mechanism it does not make use of default values in the PADL-2 program. The default in PADL-2 is that primitives are displayed at the global origin. In the CSG-list this must be stated explicitly.

PADL-2 allows motion operators to be executed in any order, but in the internal representation the order is fixed as \{movx, movy, movz, degx, degy, degz\}. The effect of changing the order of motions can be achieved by nesting the CSG-list. For example, the internal equivalent to "(DEGX=30, MOVX=5)" is "(CSG (CSG <csg list> AT (0 0 0 30 0 0)) AT (5 0 0 0 0 0))." The more deeply nested motions are carried out first.
Figure 9-5: Two-dimensional constructive geometry primitives.

9.5.2 Constructive plane geometry

The "extrusion" abstraction requires some representation of plane shapes as cross-sections of the extruded solids. In keeping with the conventions of PADL-2, a two-dimensional counterpart to constructive solid geometry has been devised, called constructive plane geometry (CPG).

The form Shape-2d is given the property slot Cpg. The filler for this slot is a CPG-list. The specification for the CPG-list is similar to that of the CSG-list.

\[
\langle\text{cpg list}\rangle = (\text{CPG} \ \langle\text{primitive shape} \ | \ \text{set operation} \ | \ \langle\text{cpg list}\rangle \ \text{AT} \ \langle\text{motion arguments}\rangle)
\]

In two-dimensional space there are only three motion arguments, i.e., two translations and a single rotation: \{movx, movy, degz\}. The specifications for the two-dimensional primitives, as illustrated in figure 9-5, are:

\[
\langle\text{primitive shape}\rangle = (\text{RECT} \ x \ y) \\
(\text{TRI} \ x \ y) \\
(\text{CIRC} \ d)
\]

The set operations are:

\[
\langle\text{set operation}\rangle = (\text{UN} \ \langle\text{cpg list}\rangle \ \langle\text{cpg list}\rangle) \\
(\text{DIF} \ \langle\text{cpg list}\rangle \ \langle\text{cpg list}\rangle) \\
(\text{INT} \ \langle\text{cpg list}\rangle \ \langle\text{cpg list}\rangle)
\]

A three-dimensional solid can be developed from these two-dimensional shapes by adding the Z-axis dimension. Thus an element with cross-section "(RECT 2 4)" with a
Figure 9-6: Triangle built up from primitive right triangles.

length of 10 becomes "(BLO 2 4 10)." Similarly CIRC into CYL. The Wedge is developed along the X-axis in PADL-2. A simple rotation however brings this into line with the other primitives. A CPG-list can be converted into a CSG-list by developing each of the primitive shapes in the list.

9.5.3 Example

In the knowledge base presented in Appendix A, the properties of all defined 2D shapes are inherited from the two forms Triangle and Built-up-shape-2d. To compute a constructive plane geometry representation of the family of 2D shapes, only these two forms need to be considered.

Consider first the form Triangle. The CPG primitive is the right triangle. The CPG representation of a general triangle must therefore consider five cases of combining the primitive. These cases are shown in figure 9-6. Cases (a) and (c) are represented as a difference of primitive right triangles; cases (b) and (d), themselves right triangles, are represented as the simple primitive and the primitive rotated; the case (c) is a union of primitive right triangles.

The other form that must compute its own CPG representation is the form Built-up-shape-2d. This form uses the union operator to combine component shapes.

As PADL-2 is a solids modeller, 2D shapes cannot be displayed. However, the function "Plane-to-solid" can be used to extrude a 2D shape along the Z-axis. The arguments to Plane-to-solid are a CPG-list and a length. The function "Padl2-display" converts a CSG-list into a series of commands which can be used by the modelling program. The following interaction creates a Quadrilateral shape and produces the appropriate PADL-2 commands.
[1] (create-instance 'q1 'quadrilateral)
Instance defn (string): "Example quadrilateral 1"
B: 100
DL: 25
DR: 50
A: 25
C: 50
REFERENCE-POINT: center
ROTATION: 0

[5] (padl2-display (plane-to-solid (get-value 'q1 ')'((cpg *)) nil) 1))
ERASE
C17 = WED (X= 1 , Y= 50 , Z= 25 ) \nAT (MOVX= -1 )
...  
C13 = (C14 U C20 ) \nAT (MOVX= -55.55555555555555 , MOVY= -17.5925925925926 )
DISP C13
()

The wireframe display of the above interaction is shown in figure 9-7. Included in the display is the coordinate origin. As the Reference-point was given as Center, the origin is at the center-of-gravity of the Quadrilateral. The computation for center-of-gravity is inherited from the parent form Built-up-shape 2d.

It should be noted that the definition of Quadrilateral in this example is that defined in Appendix A, that is, the Quadrilateral is defined as a kind of Built-up-shape-2d consisting of two Triangles. Given the forms Triangle and Built-up-shape-2d, any 2D shape that can be described as a union of Triangles can be displayed, and the locations of a number of reference points, including the center of gravity, can be automatically computed.

The efficiency of some computations can be improved by giving specializations their own methods. For example, it is far more efficient for the form Rectangle to compute its own area, rather than computing the area by methods inherited through Quadrilateral, Built-up-shape-2d and Triangle. This is also true for the computation of the CSG representation; it more efficient for the form Rectangle to use the "Rect" primitive instead of using the inherited representation as a union of "Tri" primitives.

The discussion, so far, has been limited to two-dimensional shapes. To bring in the third dimension, the form SLEE (straight-line extruded element) will be introduced. This
Figure 9-7: The wireframe display of a quadrilateral.

form is shown in Appendix A. The method for computing the CSG-list uses the function Plane-to-solid to extrude the 2D shape along the Z-axis. The appropriate motions are then applied to the extruded shape to get it to its proper location in 3D space.

Using the forms Slee and Wshape, a three-dimensional wide-flange shape can be described.

[1] (create-instance 's1 'slee)
Instance defn (string): "An example wshape beam"
LINE-VECTOR: line-vector-3d
END1: origin-xy
END2: vector-3d
X: 0
Y: 0
Z: 1000
CROSS-SECTION: wshape
D: 310
B: 165
W: 6
TH: 10
REFERENCE-POINT: center
ROTATION: 0

[2] (get-value 's1 '((csg *)) nil)
(CSG (CSG (UN (CSG (CSG (BLO 165 10 1000.) AT (-82.5 -10 0 0 0 0)) AT (0
Figure 9-8: Wireframe image of Wshape beam.

155 0 0 0 0)) (CSG (UN (CSG (CSG (BLO 165 10 1000.) AT (-82.5 0 0 0 0)) AT (0. 0 0 0 0)) (CSG (CSG (BLO 165 10 1000.) AT (-145 0 0 0 0)) AT (0 0 0 0 0 0)))) AT (0 0 0 0 0)) AT (0 0 0 0 0)) AT (0 0 0 0 0) AT (0 0 0 0 0) AT (0 0 0 0 0) AT (0 0 0 0 0)) [3] (padl2-display (get-value 'm1 '((csg *) nil))

ERASE
C12 = BLO (X= 165 ,Y= 10 ,Z= 1000. ) \\
AT (MOVX= -82.5 ,MOVY= -10 )
C11 = C12 \\
AT (MOVY= 155 )
C15 = BLO (X= 165 ,Y= 10 ,Z= 1000. ) \\
AT (MOVX= -82.5 )
C14 = C15 \\
AT (MOVY= -155 )
C17 = BLO (X= 6 ,Y= 290 ,Z= 1000. ) \\
AT (MOVX= -145 ,MOVY= -145 )
C16 = C17
C13 = C14 UN C16
C10 = C11 UN C13
C9 = C10
C8 = C9
DISP C8
()

The display of these PADL-2 commands is shown in figure 9-8.
9.6 Integrity maintenance

Integrity is maintained by the use of pointers between functionally related data. When a data value is changed, appropriate measures are taken to update dependent values. The maintenance of the functional links is crucial: as there is no procedure for integrity checking, it is essential that integrity be actively maintained at all times.

Integrity principally deals with the functional links between data values. Another class of links among forms and instances is also important. These are the semantic links, which are concerned with meaning rather than computation.

9.6.1 Semantic links

The first of these semantic links is the AKO (a kind of) link. All forms are related within the AKO network. This link is specified in the form definition when a new form is defined as a specialization of an existing form. The AKO relation is important because it allows the inheritance of property slots from the parent form. The link is represented by storing the name of the parent in the association “AKO.” To make the link bi-directional, the parent must also store the name of the child. The child’s name is stored in the association “Subclasses.” According to the rules of single inheritance, a form may have any number of children but only one parent.

Another link is the AIO (an instance of) link. Because an instance, which contains only instance data, must be coupled with its form to be interpreted, the AIO link is essential. The instance stores the name of its form in the association “AIO.” The form stores the name of its instances in the association “Instances.” An instance can be the instance of only one form, but a form may have any number of instances.

Local forms are not actually nested within the superior form but are given internally generated unique names and stored in the same manner as their global counterparts. The user can only access the local forms through the enclosing form. A form stores the names of its local forms in the association “Locals.” Because the local forms have local “given names” the Locals are stored as pairs associating the given name with the unique global name—this is necessary if the user is to be able to call local forms by their given names. The complementary pointer in the local form is stored in the association “Supers.” A form may have any number of local forms but a local form may have only one superior. A local form
Figure 9-9: Semantic links among forms and instances.

differs from a global form only in having a superior form, and local forms may themselves have local forms.

In the instance, the association “Locals” stores the name of slot fillers which are themselves instances. These filler instances represent an inferior level in the description hierarchy. The inferior instance stores the name of its superior in the association “Supers.” If the local instance is globally defined, it may have any number of superiors. A locally defined instance can have only one superior. The locally defined instance is distinguished from its global counterpart with the association (local? #t). While the association Locals contains the name of the inferior instance, the association Supers contains the address of the slot the instance fills. A local instance may be an instance of a global or local form.

In the following example, the global instance R1 is an instance of the global form Rectangle. The filler of the slot Reference-point is a local instance of the local form Center which is defined in Rectangle as a sub-class of the global form Vector-xy.

[14] (create-instance 'r1 'rectangle)
Instance defn (string): "An example rectangle"
D: 20
B: 10
REFERENCE-POINT: center
ROTATION: 0
;

The semantic links involved in this example are illustrated in figure 9-9.
9.6.2 Functional links

In the second category of links are the functional links between data items. It is these links that are the true concern of the integrity maintenance system.

The addresses of dependent slots are stored in the slot facet Cards while the addresses of ingredient slots are stored in the slot facet Ingreds. A slot address is simply a pair consisting of the instance name and the slot name.

In the following example an instance of the form Component is described. A Component is a homogeneous solid object with parameters Geometry (a kind of 3D shape) and Material. In this case the geometry is defined as an SLEE (straight line extruded element) and the material as Steel. The parameters of SLEE are Line-vector, which defines the length and location of the element, and Cross-section. The Line-vector is defined by giving the locations of two ends of the line-vector. The Cross-section is a Rectangle. The development of this description was illustrated in figure 4-5, where the cross-section was defined as a Wshape. Figure 9-10 presents essentially the same description, showing the dependency network. The arrows in this figure point in the direction of the internal slot queries, i.e., from the dependent to the ingredient slot. It is understood that the link is bi-directional.

[1] (create-instance 'c1 'component)
Instance defn (string): "An example solid object"
GEOMETRY: slee
LINE-VECTOR: line-vector-3d
    4D1: origin-xy
    END2: vector-zz
    X: 300
    Z: 400
CROSS-SECTION: rectangle
    D: 50
    B: 20
REFERENCE-POINT: center
    ROTATION: 0
    MATERIAL: ectoplasm

[2] (pp (get-instance 'c1))
(c1 (AIO COMPONENT)
    (SUPERS)
    (LOCALE INST-1 INST-30)
Figure 9-10: Integrity links in an instance of Component.

(SLOTS
  (GEOMETRY (FILLER INST-1))
  (MATERIAL (FILLER INST-30))
  (VOLUME)
  (WEIGHT))

(DEFP "An example solid object")

\[3\] (get-value 'c1 '((weight *)) nil)

5.

\[4\] (pp (get-instance 'c1))
(C1 (AIO COMPONENT)
  (SUPERS)
  (LOCALS INST-1 INST-30)
  (SLOTS
    (GEOMETRY (FILLER INST-1))
    (MATERIAL (FILLER INST-30))
    (WEIGHT
(INGREDS (INST-30 DENSITY) (INST-1 VOLUME))
(FILLER 5.))
(DEFN "An example solid object")

Recognizing the global names INST-1 and INST-30 as the local instances of Slec and Material, the above interaction shows that the ingredients of the slot Weight of the instance of Component, C1, are Material:density and Geometry:volume. Correspondingly, these ingredient slots store the name of the Weight slot in their Cards facets. This is seen in internal representations of the instances INST-1 and INST-30.

(INST-1
 (AIO SLEC)
 (SUPERS (C1 GEOMETRY))
 (LOCALS INST-2 INST-6)
 (SLOTS
  (LINE-VECTOR (FILLER INST-2))
  (CROSS-SECTION (FILLER INST-6))
  ...
  (VOLUME
   (INGREDS (INST-6 AREA) (INST-1 LENGTH))
   (FILLER 500000.)
   (CARDS (C1 WEIGHT'))))
 (LOCAL? #T))

(INST-30
 (AIO ECTOPLASH)
 (SUPERS (C1 MATERIAL))
 (LOCALS)
 (SLOTS
  (DENSITY (FILLER 1.e-5) (CARDS (C1 WEIGHT'))))
 (LOCAL? #T))

This example illustrates the integrity pointers in the case in which the slot value is a local instance, which is considered the usual case. Several special cases must also be considered. The special cases concern (1) the case of the null ingredient, (2) the case of the non-existent ingredient, and (3) the case of the global instance as filler.

When a queried slot evaluates to nil, that is, a value cannot be computed, it is filled with the asked-token. The asked-token is treated as nil, except that it prevents any attempt at recomputing the slot value. As it is just as important to maintain the integrity of the asked-token as any other value, the pointers are established as in the usual case.
When an ingredient slot does not exist it is obviously impossible to include it in the dependency network. Yet all the same, it is necessary that the querying slot be notified if and when this slot comes into existence. The solution to this problem is to link to the last node of the access path of the query. With reference to figure 9-10, if the slot Material in instance C1 had no value, the slot Weight, querying Material:density, would leave its address with the slot Material and register Material as one of its ingredients. This done, any change to the slot Material would result in the update of the slot Weight. This is desirable as the slot Density cannot be created without changing the value of Material.

The final special case involves the global instance as filler. Referring to figure 9-10, this case can be visualized by naming the filler of Cross-section the global instance R1 instead of an local instance of Rectangle. The problem introduced by the global instance is that changing the value of the slot Cross-section, that is, changing the filler from R1 to something else, will not change the slot values of R1 and hence the dependants of R1 will not be notified, even though the functional relation has been severed. This problem does not exist with the local instance because the local instance has no independent existence and the system can simply notify the dependants of all the slots of the filler. A global filler cannot be deleted because its existence is not dependent upon being a slot filler, and all dependants should not be notified because the instance may be part of other networks, and there is no way to tell which dependants are affected. One solution to the problem is to store a copy of the pointers to slots within a global instance with the slot the instance fills. By doing this, deleting the instance from the slot will cause all affected dependants to be re-evaluated. This extra pointer does not interfere with the rest of the dependency network. The dependency network is shown in figure 9-11.

All queried slots within a description become part of a dependency network. Each slot keeps a record of its ingredients and dependants. It is the knowledge of these dependencies which allows the integrity system to maintain the description data.

9.6.3 Change propagation

The primary way of changing a description is to delete a slot value. This is easily done by simply replacing the current slot value with nil. Integrity requirements demand that dependent values either be deleted or marked as no longer valid. In either case the change must be propagated immediately to all dependent slots. The description system has no way
of checking integrity. Instead the integrity maintenance system actively maintains data in
direct response to changes.

[32] (show-instance 'c1)
************
*  C1  *
************
An example component with un-named fillers
C1 <aio COMPONENT>
GEOMETRY: <aio SLEE>
  LINE-VECTOR: <aio LINE-VECTOR-3D>
  :
  CROSS-SECTION: <aio RECTANGLE>
    D: 50
    B: 20
    :
    AREA: 1000
    :
    LENGTH: 500.
  CSG:
  VOLUME: 500000.
  MATERIAL: <aio ECTOPLASM>
    DENSITY: 1.e-5
  WEIGHT: 5.
()
[33] (delete-slot3 'c1 '(geometry cross-section d))
"T
Chapter 9: Demonstration

[34] (show-instance 'c1)
************
* C1 *
************

An example component with un-named fillers
C1 <ai COMPONENT>
GEOMETRY: <aio SLEEP>
   LINE-VECTOR: <aio LINE-VECTOR-3D>
   ...
   CROSS-SECTION: <aio RECTANGLE>
      D:
      B: 20
      ...;
      AREA:
      ...
      LENGTH: 500.
      CSG:
      VOLUME:
      MATERIAL: <aio ECTOPLASM>
      DENSITY: 1.e-5
      WEIGHT:
      ()

All of the dependent slot values can be restored, so nothing is actually lost in deleting dependants except the cost of recomputation. Upon deletion the integrity links between slots are also deleted. This is done because the functional relation is between values, not slots.

In making changes to data values, the user only has the right to change values that the user has entered, that is, the parameter values. All property values are maintained by the integrity maintenance system.

Values are restored by querying for the value. To make restoration easier the names of deleted slots that have no dependants are posted on a "bulletin board," the list Change-notices. Querying for the value of these slots causes queries to be made to all of the other deleted slots including the parameter slot that the user has deleted. The function Serve-change-notices will query all slots in the Change-notices list.

[35] change-notices
((C1 WEIGHT))
[36] (serve-change-notices)
D: 40
Change notices served

[37] (show-instance 'c1)

********** ...
* C1 *
**********

An example component with un-named fillers
C1 <asio COMPONENT>
GEOMETRY: <asio SLEE>
     LINE-VECTOR: <asio LINE-VECTOR-3D>
      :
CROSS-SECTION: <asio RECTANGLE>
     D: 40
     B: 20
     AREA: 800
     LENGTH: 500.
     CSG:
     VOLUME: 400000.
MATERIAL: <asio ECTOPLASM>
       DENSITY: 1.e-5
WEIGHT: 4.
()

The function Change-slot is a combination of Delete-slot and a query to the deleted slot. The user upon deleting the slot will immediately be asked for a replacement value.

[39] (change-slot3 'c1 '(geometry cross-section d))
D: 30
(30)

The function Update-slot is similar to Change-slot except that it queries the slots in Change-notices rather than the deleted slot. Update-slot also results in the user being queried for a replacement value for the deleted slot. It will be undesirable to use Update-slot if more than a single change is planned, as recomputation is performed for each change; in this case it is much better to use Change-slot. Delete-slot is used if there is no replacement value.

[43] (update-slot3 'c1 '(geometry cross-section d))
D: 50
Change notices served

#T
9.7 Reduction

A specialization is a copy of its parent form with one or more of the parent's parameter values constrained. Being more constrained and having a richer implicit meaning, the specialization is a higher level of abstraction. Reducing the level of abstraction of an instance is a process of expressing an instance of a specialization as an instance of the parent form. The specialization inherits all the slot of its parent but may have additional parameter slots of its own. Reducing the level of abstraction can be easily accomplished by removing these additional slots so that only those slot common with the parent are retained. If this is done, care must be taken to update the integrity links and to preserve, as much as possible, all input data.

Input data is preserved by making sure that the parameters of the parent form, those that are not also parameters of the child, are evaluated—it is to be understood that this evaluation has the many usual side-effects including the storage of the value, the creation of integrity links, etc.

A slot that is computed in the child but which is a parameter in the parent must break its integrity links with its former ingredients. The dependants of the slot are unaffected by the reduction. If a property slot is computed by a different method in the parent form, that is, the property method is not inherited, the integrity links must be revised to accord with the parent's method. As integrity links are established as a side effect of value queries, the way to revise these links is to evaluate the property method. The property values themselves are unaltered so there is no need to replace the value with the recomputed value. The value will be replaced as a matter of course should any of its new ingredients be updated.

In the following example an instance of Square is created and then reduced. In order to create some integrity links, the values of the property slots Area, C-of-g (center of area) and Cpg (the 2D constructive geometry representation) are queried. The parameter Reference-point is filled with an instance of the local form Topcenter. The integrity links established by these queries are illustrated in figure 9-12. In this figure the arrows represent the internal value queries and so point from the dependant to the ingredient. Square inherits its property methods and local forms Center and Topcenter from its AKO parent, Rectangle.
Figure 9-12: Integrity links in an instance of Square.

```
[1] (create-instance 's1 'square)
Instance defn (string): "a two-dimensional shape"
D: 50
REFERENCE-POINT: topcenter
ROTATION: 0

[5] (get-value 's1 '((cpg *)) nil)
(CPG (RECT 50 50) AT (-25 -50 0))

[6] (show-instance 's1)
************
* S1 *
************
A two-dimensional shape
S1 <aio SQUARE>
D: 50
REFERENCE-POINT: <aio TOPCENTER>
  X: 25
  Y: 50
  LENGTH:
  ROTATION: 0
AREA: 2500
C-OF-G: <aio CENTER>
  X: 25
  Y: 25
  LENGTH:
  CPG: **CIF**
```

()`
The instance of Square is reduced to an instance of Rectangle using the shell function Reduce.

[7] (reduce 's1)
Instance s1 being reduced from SQUARE to RECTANGLE
s1
[8] (show-instance 's1)

************
*  s1  *
************
A two-dimensional shape
s1 <sio RECTANGLE>
D: 50
B: 50
REFERENCE-POINT: <sio TOPCENTER>
  X: 25
  Y: 50
  LENGTH:
  ROTATION: 0
  AREA: 2500
  C-OF-G: <sio CENTER>
    X: 25
    Y: 25
    LENGTH:
    CPG: **CPG**

() 

The instance S1 is now an instance of Rectangle and as such has parameters D and B. Slot B, which had previously been dependent on the value of slot D, is now an independent parameter; this is illustrated in figure 9-13 by the absence of an arc from slot B to slot D. If slot B had not been evaluated before the reduction it would have been evaluated as part of the reduction process. As property methods are common between Square and Rectangle, the property slots are unaffected by the reduction, and it not necessary to revise their integrity links.

The instance of Rectangle can be reduced further to an instance of Parallelogram.

[9] (reduce 's1)
Instance s1 being reduced from RECTANGLE to PARALLELOGRAM
Instance inst-20 being reduced from TOPCENTER to VECTOR-XY
s1
[10] (show-instance 's1)

************
*  s1  *
************
Figure 9-13: Integrity links in an instance of Rectangle.

A two-dimensional shape
S1 <aio PARALLELOGRAM>
D: 50
B: 50
A: 0.
REFERENCE-POINT: <aio VECTOR-XY>
  X: 25
  Y: 50
  LENGTH:
ROTATION: 0
AREA: 2500
C-OF-G: <aio CENTER>
  X:
  Y:
  LENGTH:
CPG: **CPG**
()

In the above transaction, the local form Topcenter is not defined in the form Parallelogram and thus cannot be kept as a filler of the slot Reference-point. Topcenter is therefore itself reduced to an acceptable form. In this case the parent of Topcenter is the global form Vector-xy. The slots X and Y in Vector-xy are independent parameters.

The local form Center presents a different case. Both Rectangle and Parallelogram have a local form called Center, but Rectangle redefines the local form rather than inheriting it.
Figure 9-14: Integrity links in an instance of Parallelogram.

In this case the local form is not reduced but is replaced by a new instance of the parent's local form of the same name. The slots of this new instance will remain unvalued until queried.

The integrity links for Parallelogram are illustrated in figure 9-14. Parallelogram inherits its property methods from Built-up-shape-2d. Area and Cpg therefore are computed as aggregate properties of component items. In this figure the instances of Item are left un-detailed for clarity.

Parallelogram can further be reduced to Trapezoid and Quadrilateral.

Throughout this process of reduction the property values of the instance remained uncharged. Although the property methods have been re-evaluated to re-establish the integrity links the values are still those computed by the methods in Square. Thus, Cpg is still (CPG (RECT 50 50) AT (-25 -50 0)). Although the instance is now described as a Quadrilateral, its shape has not changed. Changes can be introduced to the instance using the shell functions Update-slot or Change-slot.

The shape can be further reduced until it is finally reduced to the class head. In the case of the original instance of Square, the final reduction is to the class Shape-2d.

[20] (reduce 's1)
Instance S1 being reduced from BUILT-UP-SHAPE-2D to SHAPE-2D
Instance inst-24 being reduced from CENTER to VECTOR-XY
S1
[21] (show-instance 's1)
**************
  S1  *
**************
A two-dimensional shape
S1 <aio SHAPE-2D>
REFERENCE-POINT: <aio VECTOR-XY>
  X: 26
  Y: 50
  LENGTH:
  ROTATION: 0
  AREA: 2700.
C-OF-C: <aio VECTOR-XY>
  X: 30.7407407407408
  Y: 27.4074074074074
  LENGTH:
  CFG: **CFG**
()
[22] (reduce 's1)
Instance S1 cannot be reduced
()

In the final reduction to Shape-2D, this incomplete form consists of a list of class properties as parameters. This final reduction therefore reduces the instance to a collection of property values. The local form Center is reduced to an instance of its AKO parent Vector-xy.

9.8 Defaults

In the description system default values are clearly identified as such. The default value is stored in the filler facet as a "special filler," that is, as a list consisting of three items, (default <filler> <r>). The first item is the special-filler key. The second item is the actual default value. The last item is the time stamp. This last is needed because when querying for hard values the default filler acts as the asked-token and so must assume the time-stamp of the asked-token it replaces. A failed default computation takes the special filler (default asked <r>).

The mechanism for computing default values is similar to that for computing hard values. The default may be a fixed value, a form or a method. The default cannot be
an instance, as forms have no knowledge of instances. In addition, the default method cannot call on the ask-user function. A default method is similar to the value method and may query for hard values or default values. The Default facet may contain a number of alternative defaults. The alternative defaults are evaluated in the order of appearance until a valid value is produced.

Consider the definition of Rectangle.

(RECTANGLE
 (AKO PARALLELOGRAM)
 (DEFN "A rectangle. Local origin at bottom left corner.
"
 (SLOTS
  (A ($VALUE 0.) ($TRANSITION? #T))
  (D ($DEFAULT ($LET ((B (@ B)) (+ B 1.5))
           100))
  (B ($DEFAULT ($LET ((D (@ D))) (/ D 1.5))
       75))))

(LOCAL-FORMS
 
)

(METHODS
  (GET-AREA ($LET ((B (@ B)) (D (@ D))) (+ D B))
  
))

Rectangle is a kind of Parallelogram and thus inherits most of its slots. Slots D and B are parameter slots in Parallelogram and remain parameter slots in Rectangle. However their default values are changed. Each of these slots has two defaults. The first default is based on another hard value: the default value for D is computed as 1.5 times the hard value of B; similarly, the first default value for B is computed as 2/3 the hard value for D. As these defaults depend on a Get-value query that may fail, a second default is also included in each slot: for D the second default value is 100 and for B it is 75. The slot for Area is inherited through the AKO hierarchy from Shape-2d and Built-up-shape-2d:

(area
  ($value get-area)
  ($menu number)
  ($default "value"
    ($defn "area of shape")))
The token "value in the Default facet indicates that the default method is computed using the value method, but with all value queries interpreted as default queries. The definition of Rectangle redefines the slot method Get-area without altering the slot itself.

Consider a rectangle with one of the parameters unknown. In this case the value of D is computed from the known value of B.

[1] (create-instance 'r1 'rectangle)
Instance defn (string): "An example rectangle"
D: nil
B: 50
:
[4] (get-default 'r1 '((area *)) nil)
3750.
[5] (show-instance 'r1)
**************
 * R1  *
**************
An example rectangle
R1 <aio RECTANGLE>
D: **default= 75.
B: 50
:
AREA: **default= 3750.
:
()

The display of the instance clearly identifies the default values.

If D is known and B unknown, the default value of B can be similarly computed from the known value of D.

If both D and B are unknown, the first defaults fail and the alternative defaults are used.

[9] (create-instance 'r3 'rectangle)
Instance defn (string): "Third example rectangle"
D: nil
B: nil
:
[10] (get-default 'r3 '((area *)) nil)
7500
[11] (show-instance 'r3)
*************
* R3 *
*************
Third example rectangle
R3 <aio RECTANGLE>
D: **default= 100
B: **default= 75
... 
AREA: **default= 7500
...
()

Finally, it should be noted that if it is possible to compute a hard value, it will be computed and stored as a hard value even if it is computed in response to a Get-default query.

9.8.1 Default integrity

The integrity of default values must be maintained no less than the integrity of hard values. The basis of the integrity maintenance system is the record of dependants and ingredients for each slot. Names of dependants are stored in the facet "Cards" and names of ingredients are stored in the facet "Ingreds." The situation is mildly complicated by the existence of alternative defaults—both failed and successful default methods will create dependency links. In addition, the computation of a default value is always preceded by a failed computation of a hard value; this too leaves pointers. Consider the instance of Rectangle, R2.

[3] (create-instance 'r2 'rectangle)
Instance defn (string): "Another example rectangle"
D: nil
B: nil
...
[4] (get-value 'r2 '((area *)) nil)
()
[5] (pp (get-instance 'r2))
(R2 (AIO RECTANGLE)
  
  (SLOTS
   (D (FILLER (ASKED 0)) (CARDS (R2 AREA)))
   (B (FILLER (ASKED 0)) (CARDS (R2 AREA)))
)
(AREA (INGREDS (R2 D) (R2 B)) (FILLER (ASKED 0)))

(CPG))

(DEFPN "Another example rectangle")

In this example it can be seen that slots D and B have calling cards left by the dependent slot Area, which in turn stores the addresses of slots D and B as ingredients. These integrity links are created as a side-effect of the failed evaluation of slot Area. These links are retained when the default query is made.

[6] (GET-DEFAULT 'r2 '((area *)) nil)
7500

[10] (PP (GET-INSTANCE 'r2))
(R2 (AIO RECTANGLE)

(SLOTS
  (D (FILLER (DEFAULT 100 0)))
  (CARDS (R2 B) (R2 AREA))
  (INGREDS (R2 B)))
  (B (FILLFR (DEFAULT 75 0)))
  (CARDS (R2 D) (R2 AREA))
  (INGREDS (R2 D)))

  (AREA (INGREDS (R2 D) (R2 B))
    (FILLER (DEFAULT 7500 0)))

  (CPG))

(DEFPN "Another example rectangle")

In the above example the first default for B calls on the value of D, and the first default for D calls on the value of B. Thus each of these slot is a failed ingredient and dependant of the other. Updating one of these slots will cause the other to be updated as well.

[15] (UPDATE-SLOT 'r2 'd)
D: 45
Change notices served
#T
[16] (PP (GET-INSTANCE 'r2))
(R2 (AIO RECTANGLE)

  (SLOTS
Changing the value of slot D notifies dependants B and Area. Slot B discards the default value 75 and recomputes its value with the first default method which calls on D. The dependency links are revised. The links associated with inferior defaults are discarded to ensure that only a better default or a hard value can displace the current default value.

It may be noted that the value of the slot Area is now a hard value, as its ingredients are hard values. Because inferior dependency links are discarded there is no danger of a default ingredient displacing the hard value.
9.9 Explanation facility

The role of the explanation facility is to present the user with information as and when it is needed. The shell functions Show-instance, Show-form and Show-local-form display instances, forms and local forms. Show-instance shows slot fillers only as the other facets are mainly used for internal computation. Local instances are included in the display. To examine other facets the user may display the internal representation.

The most important thing that the explanation facility does is provide a menu of good fillers when the user is asked to fill a slot. Menu is based on the contents of the Menu facet but is expanded to include all sub-classes of the menu items, all instances of these subclasses and any local forms.

Every form and instance has a Defn association which holds a textual explanation of the form or instance. Every slot has a Defn facet which holds a textual explanation of the slot.

When queried for a slot filler the user may enter a single question mark to get the definition of the slot and the complete menu of good fillers. Each item in the menu is numbered. To get a definition of a menu item the user may enter "?<n>\n", where <n> is the item number. To select an item the user may type in the name of the item or "@<n>\n."

Consider the creation of an instance of Quadrilateral.

[1] (create-instance 'ql 'quadrilateral)
Instance defn (string): "Example quadrilateral 1"
B: ?
***DEFN: The length of the base
***MENU*
  1>NUMBER
  B: 100
  DL: 25
  DR: 50
  A: 25
  C: 60
REFERENCE-POINT: ?
***DEFN: ()
***MENU***
  1>VECTOR-XY form
  2> ORIGIN-XY form
  3> BUTTONCENTER local form
4> CENTER local form

++DEFAULT: CENTER

REFERENCE-POINT: center

ROTATION: 0

************

1> Q1

************

Q1 <a1 QUADRILATERAL>

B: 100

DL: 25

DR: 50

A: 25

C: 50

REFERENCE-POINT: <a1 CENTER>

X:

Y:

LENGTH:

ROTATION: 0

AREA:

C-OF-G:

REF-COPG:

REF-COPG-ROT:

CPG:

()

The inclusion of instances in the menu of good fillers is seen in the selection of shapes available as the filler of Cross-section in the following example. Instances Sq1, R1, P1, T1 and Q1 have already been created.

[16] (create-instance 'a1 'a1)

Instance defn (string): “Example solid shape”

CROSS-SECTION: ?

++DEFN: Cross-section shape of element

+++MENU+++  

1> SHAPE-2D form

2> QUADRILATERAL form

3> TRAPEZOID form

4> PARALLELOGRAM form

5> RECTANGLE form

6> R100x75 form

7> SQUARE form

8> SQ1 instance

9> R1 instance

10> P1 instance

11> T1 instance
12> Q1 instance
CROSS-SECTION: 71
**DEFN: A two-dimensional shape defined on the XY plane
CROSS-SECTION: 72
**DEFN: A quadrilateral
CROSS-SECTION: 712
**DEFN: Example quadrilaterals
CROSS-SECTION: 912
= Q1
LENGTH: 1000
*************
* S1  *
*************
S1 <aio SLEE>
CROSS-SECTION: Q1
LENGTH: 1000
VOLUME:
()

If the user selects a form at the prompt for a slot filler, the slot is filled with a local instance of the form. The user may choose to fill the slot with a new global instance by entering "new-instance" at the prompt. The new instance must be an instance of a global form contained in the menu.

9.9.1 Menu specialization

The Menu facet is an important slot constraint; only fillers contained in the slot menu are accepted by the description system. As a value constraint, the menu facet can be used to create specializations of a form. Specialization does not necessarily involve changing the Value facet of a slot. Consider this definition of the form Rectangular-slee:

(define-form 'RECTANGULAR-SLEE 
  (ako slee)
  (defn "A straight line element with rectangular cross-section")
  (slots 
    (cross-section 
      ($menu rectangle))) ))

Changing the Menu facet has the effect of constraining the choice of fillers for the slot Cross-section, thus creating a value restriction. By changing the menu of good fillers rather than dictating that the value will be Rectangle, the user is given the option of selecting from the valid sub-classes of rectangle and their instances.
[19] (create-instance 'rs1 'rectangular-slee)
Instance defn (string): "A rectangular sleeve"
LINE-VECTOR: 91
   = LINE-VECTOR-3D
END1: origin-zx
END2: vector-zx
X: 300
Z: 400
CROSS-SECTION: triangle
TRIANGLE is not a valid filler for this slot
CROSS-SECTION: wshape
WSHAPE is not a valid filler for this slot
CROSS-SECTION: ?
**DEFN: ()
***MENU***
   1>RECTANGLE form
   2> SQUARE form
   3> R100X75 form
**DEFAULT: R100X75
CROSS-SECTION: square
D: 50
REFERENCE-POINT: center
ROTATION: 0
Chapter 10
Building Description

This chapter will look at an application of the description system to the description of building structures. Two example descriptions will be presented. The first description will treat the structural frame as an assembly of independent elements with an explicit representation of member connectivity. The second example will consider the use of higher level abstractions in conjunction with the Reduce function.

10.1 Assembly and connectivity

A building is composed of four major sub-systems [Rush86]:

1) structure;
2) envelope;
3) mechanical; and
4) interior.

Each of these sub-systems falls within the domain of a separate design discipline. Design of the structural system falls within the domain of the Civil Engineer. It is this system which will be considered.

There are many types of structural systems. These systems involve different construction materials and analysis methods. Steel construction tends to use discrete components; concrete construction tends to be monolithic. Considering a structure as an assembly of discrete components, there is a wide variety of component types. Structural components can be classified as line, plane or solid elements—straight, folded or curved [Schue1er83].

The scope of the knowledge base in the following examples is limited to the description of the main gravity load resisting elements of a common type of steel-framed industrial structure. This effectively limits the description to steel beams and columns, which both can be represented as straight-line elements of constant cross-section.
The need for the explicit representation of connectivity is apparent in the computation of building loads due to gravity. While a beam is dependent on a column for its location, the column is dependent on the beam for its load.

10.1 Knowledge base

The forms comprising the knowledge base used in the following example are listed in full in Appendices A and B. Appendix A contains general purpose forms mainly concerned with the description of solids. Appendix B introduces forms specifically intended for the description of building structures.

In the knowledge base the class of physical objects is represented by the form Object. The form Object has two sub-classes, Component and Assembly. Component is the class of discrete Objects. A Component has parameters of Geometry—encompassing both shape and location—and Material. Assembly is the class of assembled objects. The parts of an Assembly may be discrete Components, or may themselves be Assemblies.

Appendix B introduces two important specializations of Component, Column and Beam. These new forms constrain the parameter slots of the parent form, slots Geometry and Material, in terms of their own parameters. In both cases the slot Geometry is constrained as a specialization of the global form SLEE (straight-line extruded element). An SLEE has parameter slots Line-vector and Cross-section. The slot Line-vector defines a vector located in 3D space; this is the vector along which the cross-section of the shape is swept.

Beam, like Column, is a kind of Component, and like Column its geometry is fixed as a kind of the form SLEE. Beam is constrained to be a horizontal element.

An assembly is defined as an aggregation of Objects. The Volume and Weight of the Assembly are computed as the sum of the Volumes and Weights of the Parts, respectively. The Csg representation is a compound of the Csg representations of the elements. The function "Padl2-assemble" takes a list of individual Csg representations and returns the representation of the assembly.

To make full use of the integrity maintenance system to manage changes in the description, the locations of the parts of the assembly are given in terms of relative geometry. Beams, for example, are located relative to the columns or beams by which they are
supported. These relative locations can be maintained when the locations of objects are changed. Thus, moving a column causes the elements supported by that column to be moved correspondingly.

The form Elevation is used to locate both Beams and Columns. Elevation specifies a horizontal plane in terms of its Y-axis intercept.

In building description, columns are usually located relative to a set of building column lines. The form Column-line is defined as a kind of Horizontal-line.

The relative locations of the Parts of the Assembly are given in terms of specializations of the form Vector-3d. These include the forms Column-line-intersection, Column-location, Beam-relative-location, etc. The location of any element can be defined relative to its supporting element.

The knowledge base defines connectivity in such a way that a component does not directly connect to another component; rather, the connection is mediated by a separate entity called a Connection; components are supported by connections, and connections are supported by components. For example, if beam B1 is supported by column C1, a single connection will support B1 and be supported by C1. A component has two kinds of slots that link the component to connections: the component has slots for the supporting connections and those for supported connections. The supporting connections of a component are fixed both in number and location. For example, a column is supported by a single connection at its base—all of the gravity load supported by the column is transferred through this single connection. A simply supported beam is supported by exactly two connections, one at either end of the beam—the gravity load on the beam is distributed between these two supporting connections. The number of supported connections that a component has is unlimited; a beam or column may support any number of other components through supported connections. Because each supporting connection is special, the forms Beam and Column have individual slots for each supporting connection. In the form Column the single supporting connection is stored in the slot Base-connection. In the form Beam the supporting connections are stored in the slots Connection1 and Connection2. Supported connections, being undifferentiated, are stored together in a single multiple-value slot. Both forms Beam and Column store their supported connections in a slot called Connections.

The parameters of the form Connection are the slots Component and Location:
(slots
  (component
    ($value ?)
    ($menu beam column foundation)
    ($defn "Host component")
    ($if-added add-component)
    ($if-removed remove-component))
  (location
    ($value select-rel-loc)
    ($force? #t))
)

The slot facet ($force? #t) is used here to force the evaluation of the slot Location in forward-chaining mode when an instance of Connection is created—because the Value facet holds a method, the slot is not interpreted as a parameter slot.

The slot Component stores the name of the component that supports the Connection. When the value of this slot is entered the If-added method puts the name of the connection in the slot Connections of the supporting component. The If-added method is:

(add-component
 (add-value filler 'connections id)))

The symbols filler and .d are replaced in the expanded method by the slot value and the instance name, respectively. The If-removed method uses the complementary shell function Delete-value.

The locations of components are determined from the locations of their supporting connections, and the location of a connection is computed relative to its supporting component. As the parameters of the forms which define locations relative to components take different parameters depending on the component type—for example, the location relative to a beam is measured from End1 of the beam, and the location relative to a Column uses Elevation—the appropriate form of the relative location, in the form Connection, is selected by the method Select-rel-loc:

(select-rel-loc
  (let ( (conn (get component)) )
    (cond ( (instance-of? 'column conn)
      'crl)
      ( (instance-of? 'beam conn)
        'brl)
      ( (instance-of? 'foundation conn)
        'frl) )
 )
(else nil)))

The local forms Crl, Brl and Frl are defined as specializations of Column-relative-location, Beam-relative-location and Foundation-location, respectively. The local form Crl is defined as:

(crl (ako column-relative-location)
(slots
 (column (value (oref <:component)))))

That is, the local form is a relative location with the reference component specified. The method functionoref forwards a query to another slot; in this case any query to the slot Column is forwarded to the slot Component of the superior instance. The local forms Brl and Frl are similarly defined.

The knowledge base forms include methods for computing gravity loads. These loads are transferred from the supported component to the supporting component through the shared connection. External loads are applied only to beams.

The global form Connection has no method for load transfer. The load transfer method is defined in the local specializations of Connection in Beam and Column. The load transferred by the base connection of a column is the sum of the loads of its supported connections. The method Get-load in the local form Base-conn in the form Column is:

(get-load
 (apply +(00 <:connections:load)))

The form Beam includes parameters for the application of external loads. Both point loads and uniformly distributed loads are accepted. The total load on the Beam is computed by the method Get-total-load:

(get-total-load
 (+ (apply +(00 point-loads:load))
 (apply +(00 distributed-loads:load))
 (apply +(00 connections:load))))

This load is distributed between the two end connections. The local specialization of Connection, Conn2, has the following method for computing its load:

(get-load
 (let ((moment (0 <:load-moment))
 (length (0 <:length)))
 (/ moment length)))
The current implementation of connectivity allows any component to connect to any other. Thus, a beam may connect to either a column or another beam, and a column may connect to another column or a beam. Because a Column, as defined, must connect to another component, a new component, Foundation, is defined to support the main building columns. A Foundation is a non-physical component that accepts load without transferring it. The Foundation component computes its load as the sum of the loads of its supported connections. In effect, the Foundation component is a simple reaction for gravity loads.

Another form that is introduced is the form Structure. A Structure is an assembly of Column-lines, Elevations, Foundations, Columns and Beams.

10.1.2 Description

To illustrate the load transfer procedures in the knowledge base, a two-storey structure will be described in this example. The floor plans of the structure are shown in figure 10-1. The structure is defined by first establishing the elevations and column-lines, then locating the foundation elements at column-line intersections, then putting in the columns and beams.

The first step in describing the structure is to define the floor elevations:

[6] (create-instances 'elevation)
Instance id: base-el
Instance defn (string): "Elevation of base of columns"
Y: 5000
***************
• BASE-EL
***************
Elevation of base of columns
BASE-EL < aio ELEVATION>
Y: 5000

Instance id: floor1-el
Instance defn (string): "Elevation of Floor1"
Y: 10000
***************
• FLOOR1-EL
***************
Elevation of Floor1
FLOOR1-EL < aio ELEVATION>
Y: 10000

Instance id: roof-el
Figure 10-1: Floor plans of two-storey structure.
Instance defn (string): "Elevation of roof"
Y: 15000

**************
• ROOF-EL •
**************
Elevation of roof
ROOF-EL <aio ELEVATION>
Y: 15000

Instance id: nil
#

Next, the column-lines are defined:

[?]
(create-instances 'column-line)
Instance id: la
Instance defn (string): "Column line LA"
POINT: origin-zx
ROTATION: 90
**************
• LA •
**************
Column line LA
LA <aio COLUMN-LINE>
POINT: <aio ORIGIN-ZX>
  X:
  Y:
  Z:
  LENGTH:
  ROTATION: 90

:

Instance id: l3
Instance defn (string): "Column line L3"
POINT: vector-zx
X: 15000
Z: 0
ROTATION: 0
**************
• L3 •
**************
L3 <aio COLUMN-LINE>
POINT: <aio VECTOR-ZX>
  X: 1
  Y:
  Z: 0
As the locations of the building columns will be defined relative to their supporting Foundation elements, it is the Foundation elements that are located at the intersection of column-lines:

[8] (create-instances 'foundation)
Instance id: f1
Instance defn (string): "Foundation at base of column C1"
ELEVATION: base-el
LOCATION-ZX: column-line-intersection
COLUMN-LINE-1: la
COLUMN-LINE-2: li
*************
  F1  *
*************
Foundation at base of column C1
F1 <aio FOUNDATION>
ELEVATION: BASE-EL
LOCATION-ZX: <aio COLUMN-LINE-INTERSECTION>
    COLUMN-LINE-1: LA
    COLUMN-LINE-2: L1
    X:                    
    Y:                    
    Z:                    
    LENGTH:              
CONNECTIONS: LOCATION: <aio LOCN>
    X:                    
    Y:                    
    Z:                    
    LENGTH:              
LOAD:
  
The columns are connected to the Foundation elements:

[9] (create-instances 'column)
Instance id: c1
Instance defn (string): "Column C1"
COMPONENT: ?
**DEFN: Host component
***MENU***
Chapter 10: Building Description

1>   BEAM form
2>   COLUMN form
3>   FOUNDATION form
4>   F1 instance
5>   F2 instance
6>   F3 instance
7>   F4 instance
8>   F5 instance
9>   F6 instance

COMPONENT: f1
TOP-ELEVATION: roof-el
CROSS-SECTION: mil
***************
  C1  *
***************
C1 <asio COLUMN>
BASE-CONNECTION: <asio BASE-CONN>
  CROSS-SECTION:
  LENGTH:
  CSG:
  VOLUME:

The Beams are located relative to their supporting Columns and Beams. The description of Beams includes the beam loading. The beam loading has been computed manually based on tributary areas and applied to the beams as uniformly distributed linear loads. Point loads are directly applied to the beam.

[13] (create-instances 'beam)
  Instance id: b1
  Instance defn (string): "Beam B1"
  COMPONENT: c1
  ELEVATION: ?
  **DEFN: Elevation
  ***MENU***
1>ELEVATION form
  2>  BASE-EL instance
  3>  FLOOR1-EL instance
  4>  ROOF-EL instance
  ELEVATION: 03
  = FLOOR1-EL
  COMPONENT: c2
  ELEVATION: 03
Chapter 10: Building Description

- FLOOR1-FL

POINT-LOADS: nil
DISTRIBUTED-LOADS: ?
**DEFN: ()
***MENU***

1> DIST-LOAD local form
2> FULL-DIST-LOAD local form

DISTRIBUTED-LOADS: T1
**DEFN: Uniform load distributed on portion of beam
DISTRIBUTED-LOADS: T2
**DEFN: Uniform load distributed on full length of beam
DISTRIBUTED-LOADS: Q1
= 1f-55
L1: 0
L2: 2000
DIST-LOAD: 16.67
DISTRIBUTED-LOADS: Q1
= 1f-55
L1: 4000
L2: 7000
DIST-LOAD: 17.5
DISTRIBUTED-LOADS: nil
CROSS-SECTION: nil
MATERIAL: nil
***************
= B1 =
***************
B1 <aio BEAM>

CONNECTION1: <aio CONN1>
  COMPONENT: C1
  :
CONNECTION2: <aio CONN2>
  COMPONENT: C2
  :

CONNECTIONS:
POINT-LOADS: **ASKED T=0
DISTRIBUTED-LOADS: <aio DIST-LOAD>
  L1: 4000
  L2: 7000
  DIST-LOAD: 17.5
  LOAD:
  LOAD-MOMENT:
  <aio DIST-LOAD>
  L1: 0
L2: 2000  
DIST-LOAD: 16.67  
LOAD:  
LOAD-MONENT:  
CROSS-SECTION: **ASKED T=O  
MATERIAL: **ASKED T=O  
LINE-VECTOR: <aio LV>  

LENGTH:  
CSG:  
VOLUME:  
WEIGHT:  

Finally, all the structure components are combined in an instance of the form Structure:

[49] (create-instance 'bldg 'structure)  
Instance defn (string): "Assembled structure"  
COLUMN-LINES: la  
  
COLUMN-LINES: nil  
FOUNDATIONS: f1  
  
FOUNDATIONS: nil  
COLUMNS: c1  
  
COLUMNS: nil  
BEAMS: b1  
  
BEAMS: nil  
***************  
*   BLDG    *  
***************  
Assembled structure  
BLDG <aio STRUCTURE>  
COLUMN-LINES: L3  
  
L3  
FOUNDATIONS: F6  
  
F1  
COLUMNS: C6
The above description is incomplete as the cross-sections of the structure components have not been input. Nevertheless, default values of some important properties can be computed. As the computation of gravity loads does not depend on the properties of the component cross-sections, the structure loads can be computed as hard values.

It should be noted that although the Csg representation has not been defined explicitly in the forms of Appendix B, the necessary procedures are inherited and a 3D model of the structure can be produced.

\[[26] \text{(get-default 'bldg '((csg *)) nil)}\]
\((\text{Csg (ASB (Csg (Csg (Csg (BLO 75 100 7000.) AT (-37.5 -50 0 0 0 0.) AT (0 0 0 0. 90. 0)) AT (0. 10000 0. 0 0 0)) (Csg (ASB (Csg (Csg (Csg (BLO 75 100 8000.)}}
\):
\[[27] \text{(padl2-display (get-default 'bldg '((csg *)) nil))}\]
\text{ERASE}\n\text{C6 = BLO (X= 75 ,Y= 100 ,Z= 7000. )}
\text{AT (NOVX= -37.5 ,NOVY= -50 )}
\text{C5 = C6 \}
\text{DISP CO}
\)

The display of these PADL-2 commands is shown in figure 10-2.

Because connectivity is explicitly represented in the description, the loading on any component or connection can also be computed. The loads supported by the structure foundations are obtained as follows:

\[[28] \text{(get-value 'f1 '((load *)) nil)}\]
248 9411224489796
\[[29] \text{(get-value 'f2 '((load *)) nil)}\]
728.088775510204
Figure 10-2: PADL-2 image of two-storey structure.

[30] (get-value 'f3 '((load *)) nil)

[31] (get-value 'f4 '((load *)) nil)

284.996918367347

[32] (get-value 'f5 '((load *)) nil)

739.654081632653

[33] (get-value 'f6 '((load *)) nil)

430.

The integrity maintenance system dynamically updates the properties of this description (including the loading data) whenever an input value is changed by the user.

10.1.3 Summary

The example application of the description system to building description uses a set of basic forms defining solid shapes. Added to these are more specialized forms that apply specifically to structures. Thus the form Column-line is defined as a specialization of the form Horizontal-line, and the forms Beam and Column are specializations of the form Component. These specializations make use of the properties of their AKO parents and
thereby have access to default values and useful computations such as the interface to the solids modeller.

10.2 Higher level abstractions

The previous examples required that each component of the structure be created individually by the user. This next example will consider the usefulness of higher level abstractions in which some instances are created by methods included in the knowledge base. Specifically, it will consider the form Structure2, a specialization of the form Structure. The form Structure2 describes a class of regular structures—orthogonal structures with regular bay and floor spacing. This example adds the forms of Appendix C to those of Appendices A and B.

10.2.1 Knowledge base

The methods in the form Structure2 create instances of the structural components using the shell function Make-instance, a non-interactive version of the shell function Create-instance. This shell function creates a global instance with specified slot values. For clarity, special care is taken in assigning names to these internally created instances: the instances of a form are sequentially numbered; for example, the columns are named C1, C2, C3, etc., and the beams, B1-1, B1-2, B1-3, etc., the first number here indicating the floor level.

The methods that create the structure components rely on a definite ordering of components. For example, the method which defines the beams spanning between columns requires that the column names be stored in a matrix representative of the actual column layout. The column names therefore cannot be stored in a multiple-value slot, as in these slots the order of entries is not fixed. Instead, the columns are stored as a matrix in the single-value slot Column-matrix. The multiple-value slot Columns queries the slot Column-matrix to create the unordered list of columns.

The function Make-instance creates a named instance with specified parameter slot values. Parameter slots that are not given values in the function call, are not evaluated. A slot that is to be evaluated by the description system is indicated by giving it the value nil, which is not itself a valid slot value. The syntax of the function is:

(make-instance
 <new-id>
As an example of the use of the function Make-instance consider the creation of an instance of Elevation. The building elevations are created by the function Make-elevations. The arguments of this function are the number of elevations and the regular vertical spacing between elevations. Each elevation is given a name, EL1, EL2, etc. The heart of the function is the call on the shell function Make-instance:

\[
\text{(make-instance (new-id) 'elevation '(((y ,(* i s)))))}
\]

The first argument to this call on the shell function Make-instance is the instance name, computed by the function New-id. Next, is the form of the instance, Elevation. Lastly, is the list of slots and their values. In this case, the slot Y is given a value computed by the procedure (* i s), where s is the elevation spacing, and i is the increment, indicating the individual elevation.

The parameter slots of the form Structure2 are:

\[
\text{(define-form ' (structure2}
\text{ (ako structure)}
\text{ (defn "A regular structure")}
\text{ (slots)}
\text{ (no-of-bays-:\#'S
\text{ ($value \,)?})
\text{ ($menu number)}
\text{ ($defn "Number of bays North-South")))}
\text{ (bay-spacing-NS
\text{ ($value \,)?})
\text{ ($menu number)}
\text{ ($defn "Bay spacing North-South")))}
\text{ (no-of-bays-EW
\text{ ($value \,)?})
\text{ ($menu number)}
\text{ ($defn "Number of bays East-West")))}
\text{ (bay-spacing-EW
\text{ ($value \,)?})
\text{ ($menu number)}
\text{ ($defn "Bay spacing East-West")))
\text{ (no-of-floors
\text{ (la:))})}
\]
An example of a property slot is the slot Elevations, which is a class property inherited from the parent form Structure. Because the values in a multiple-value slots are unordered, another slot is defined to hold the ordered list of structure elevations. This slot is Elevation-1st

\[(\text{elevation-1st} \ (\text{value get-el-1st}))\]

The method Get-el-1st uses the function Make-elevations to create the required instances of the form Elevation:

\[(\text{get-el-1st})\]
\[\text{\quad (let \ ( (n (\text{no-of-floors}))}
\[\text{\quad \ (s (\text{floor-spacing}))}
\[\text{\quad \ (\text{cons '1st (make-elevations n s))))})}\]

By prefixing the special-filler key "1st" to the list of newly-created elevations, the ordered list of elevations can be stored as a single value. The method for slot Elevations takes the ordered list of the slot Elevation-1st and creates a multiple-value filler.

The methods for creating the structure Column-lines, Foundations, Column and Beams work in a similar manner.

To make description more convenient, higher level abstractions of previously defined forms are also added to the knowledge base. Thus the forms Column-line-ns and Column-line-ew are kinds of Column-line with fixed orientations (north-south and east-west respectively, that is, parallel to the global Z and X axes); Column2 is a kind of Column that connects only to a Foundation element; Beam2 is a kind of Beam that connects only to Columns; Joist is a kind of Beam that connects only to Beams; and Joist-ew is a kind of Joist
that has a fixed orientation in the east-west direction (i.e., parallel to the global X-axis)—
the parameters of this form are the names of the supporting Beams and the Z-coördinate
of the line-vector (i.e., the perpendicular distance to the global X-axis).

In the regular structure all Columns rest on Foundations. In creating the structure,
first the Beams connecting between columns are created (instances of form Beam2) then
the Joists spanning between these are created (instances of form Joist-ew). Using these
higher level abstractions, a regular structure can be described concisely. All properties of
Structure, including the PADL-2 representation, are available to Structure2 through the
inheritance hierarchy.

While providing a concise way to describe a regular structure, the specialization is too
constrained for most structures; for example, neither of the two earlier example structures
can be described as an instance of Structure2. However, the higher level abstraction is use-
ful even in cases where the structure to be described does not satisfy all of the specialization
constraints; an irregular structured can be described, at the outset, as a regular structure,
and then the specialization constraints can be relaxed to allow changes to be made to the
description reflecting the irregularities. Specialization constraints are relaxed by reduc-
ing the level of abstraction of an instance. Thus a structure described as an instance of
Structure2 can be reduced to an instance of the less constrained form Structure.

10.2.2 Description

As an example of this manner of describing a structure using higher level abstractions
and the Reduce function, consider the description of the structure shown in figure 10-3.
This structure has much regularity that can be exploited in creating its description. A
regular structure which is similar to this is shown in figure 10-4 and is created as follows.

```
[5] (create-instance 'bldg 'structure2)
Instance defn (string): "A building structure"
NO-OF-BAYS-NS: 3
BAY-SPACING-NS: 8230
NO-OF-BAYS-EW: 2
BAY-SPACING-EW: 9145
NO-OF-FLOORS: 1
FLOOR-SPACING: 6000
NO-OF-JOISTS-PER-BAY: 4
**************
* BLDG *
```
Figure 10-3: Plan of a single-storey steel-framed structure.

**************
A building structure
BLDG <aio STRUCTURE2>
NO-OF-BAYS-NS: 3
BAY-SPACING-NS: 8230
NO-OF-BAYS-EW: 2
BAY-SPACING-EW: 9145
NO-OF-FLOORS: 1
FLOOR-SPACING: 6900
NO-OF-JOISTS-PER-BAY: 4

BLDG
[6] (pad12-display (get-default 'bldg '((csg *) nil)))
ERASE
C3 = BLD (X= 75.0, Y= 100.0, Z= 6900.0) AT (NOVX= -37.5, NOVY= -50)

CO = C1 ASS C4
DISP CO

Description using the higher level abstraction Structure2 is obviously much more concise than the equivalent description would be using the less specialized form Structure.

The PADL-2 display of the regular structure is shown in figure 10-5. Because the component names are generated internally, one will note a difference in names between corresponding components in figures 10-3 and 10-4.

Defined as an instance of the form Structure2, changes to the description can only be made within the strict constraints of this form; the structure must remain a regular structure. To introduce irregularities in the structure, the structure must be reduced from an instance of Structure2 to an instance of its AKO parent Structure. This is done using the shell function Reduce.

[7] (reduce 'bldg)
Instance BLDG being reduced from STRUCTURE2 to STRUCTURE
BLDG
[8] (show-instance 'bldg)
***************
* BLDG *
***************
A building structure
BLDG <sio Structure>
COLUMN-LINES: L3

LA
FOUNDATIONS: F12

F1
COLUMNS: C11

C1
BEAMS: B1-37

B1-1
PARTS: B1-1


Figure 10-4: Plan of regular structure.

Figure 10-5: PADL-2 image of regular structure.
B1-37
C1
:
C11
WEIGHT:
VOLUME:
CSG:
()

The process of reduction removes the parameters of the form Structure2 from the
description and leaves the parameters of Structure. Structure is defined by the unordered
multiple-value parameter slots Columns-lines, Foundations, Columns and Beams. Although
the properties of the description are not changed, the description is no longer constrained
as a regular structure; for example, unwanted individual components can be deleted from
the reduced description.

[9] (delete-instances '(b1-6 b1-9 b1-38 b1-39 b1-40 b1-41 b1-17 c12))
#T
[10] (padl2-display (get-default 'bldg '((csg *)) nil))
ERASE
C214 = BLD (X= 75 , Y= 100 , Z= 5900. )
:
C211 = C212 ASB C215
DISP C211

The plan of the structure resulting from these deletions is shown in Figure 10-6. The
PADL-2 display is shown in figure 10-7.

The reduced structure also no longer has to be orthogonal. The next step in revising
the structure towards that of figure 10-3 is to move the existing column lines to their correct
locations, introduce new column lines, and change the locations of foundation elements F7
and F11.

First, the column lines are moved:

[11] (change-slot 'lc 'x)
X: 19810
(19810)
[12] (change-slot 'ld 'x)
X: 25810
(25810)

And new column-lines created:
Figure 10-6: Plan of reduced and updated structure.

Figure 10-7: PADL-2 image of reduced and updated structure.
[13] (create-instance 'lb2 'column-line-ns)
Instance defn (string): "Column line LB2"
 X: 14760
 :
[14] (create-instance 'le 'column-line-2pt)
Instance defn (string): "Skewed column line LE"
P1: column-line-intersection
COLUMN-LINE-1: ld
COLUMN-LINE-2: li
P2: column-line-intersection
COLUMN-LINE-1: lb2
COLUMN-LINE-2: l3
 :
These new column-lines must be added to the structure:

[15] (add-fillers 'bldg 'column-lines)
COLUMN-LINES: le
COLUMN-LINES: lb2
COLUMN-LINES: nil
()

The foundation components F11 and F7 must be moved to new column-line intersections:

[16] (change-slot3 'f11 '(location-zz column-line-2))
COLUMN-LINE-2: lb2
(LB2)
[17] (change-slot3 'f7 '(location-zz column-line-1))
COLUMN-LINE-1: le
(LE)

The result of the above changes is the structure shown in plan in figure 10-8. The PADL-2 display is shown in figure 10-9.

The final update must change the connections of beams B1-5 and B1-16. Beam B1-5 must be detached from column C7 and attached to Beam B1-14. As Beam B1-5 was defined as a kind of Beam2 that connects only to columns, it must first be reduced to a kind of its parent form Beam before the update can be made. Once Beam B1-16 is detached from Column C8, this Column is no longer needed and is deleted.

[19] (reduce 'b1-5)
Instance B1-5 being reduced from BEAM2 to BEAM
Instance INST-53 being reduced from CONN-1-2 to CONN1
Figure 10-8: Plan of further updated structure.

Figure 10-9: PADL-2 image of further updated structure.
Figure 10-10: Plan of final structure.

Figure 10-11: PADL-2 image of final structure.
Instance INST-54 being reduced from LOCN to COLUMN-RELATIVE-LOCATION
Instance INST-55 being reduced from CONN2-2 to CONN2
Instance INST-56 being reduced from LOCN to COLUMN-RELATIVE-LOCATION
B1-5
[20] (change-slot3 'b1-5 '(connectica2 component))
COMPONENT: b1-14
(B1-14)
[21] (get-value 'b1-5 '((connection2 *) (location *)) nil)
L: 9145
(9145)
[22] (change-slot 'b1-16 'column-2)
COLUMN-2: c7
(C7)
[23] (delete-instance 'c8)
#T
[24] (pad12-display (get-default 'bldg '((csg *)) nil))
ERASE
C182 = BLO (X= 75 , Y= 100 , Z= 6900. ) AT (WX: X= -37.5 , WOY= -50 )
:
DISP C179
()

These few changes result in the structure shown in figures 10-10 and 10-11. These figures, except for the difference in component names, accurately describe the structure of figure 10-3.

10.2.3 Summary

This final example extends the knowledge base to include higher level specializations of previously defined forms. Thus the form Column-line-ns is a kind a form Column-line with a fixed orientation; Joist is a kind of Beam that spans between other Beams; and Structure2 is a kind of Structure that has regular bay and elevation spacings. The usefulness of these highly specialized abstractions is greatly increased by the use of the Reduce function. The Reduce function serves to relax the constraints of a specialization so that irregularities can be introduced. In this way, the basic regularity of a structure can be made use of in description, with exceptions being introduced at a reduced level of abstraction. The Reduce function is made possible by the strict definition of the AKO relation and the maintenance of dependency pointers in the reduced description by the integrity maintenance facility.
Chapter 11
Conclusion

This final chapter presents a summary of the thesis, recommendations for further development and the conclusions drawn from this work.

11.1 Summary

Engineering deals with the descriptions of physical artefacts: engineering analysis predicts the behaviour of an artefact based on its description; engineering design creates descriptions of artefacts which are to behave in a desired manner. Design is a top-down process in which the general shape of the description is determined before the details are resolved. Description deals in abstractions, that is, in views of the artefact with detail selectively omitted. An abstraction can be represented as a set of essential attributes, this set of attributes being called a "form." A form describes a class of similar objects. Individual objects are singular instances of the form or class. Attributes are of two kinds: parameters are those attributes which together suffice to fully describe an instance; properties are those attributes whose values can be derived from the values of the parameters. An instance is defined by giving values to the form parameters.

Abstraction is of two kinds: generalization considers a set of similar objects as a generic object; aggregation considers a group of objects as a single object. Specialization is a process of creating a new form by applying constraints on the parameters of an existing form, thus creating a sub-class of an existing class. The relation between the sub-class and its super-class is the AKO (a kind of) relation. Within a class all sub-classes share common properties. These properties are inherited by the sub-class from its super-class. New forms can be defined as specializations of existing forms without concern for the computation of form properties, as these are inherited. The relation of an instance to a form is the AIO (an instance of) relation.

The description system is implemented using the knowledge representation technique of frames. A frame is a multi-level association which consists of a number of slots, each
of which in turn consists of a number of facets. Each facet holds a different aspect of knowledge about the slot. Both the form and the instance are implemented as frames with slots corresponding to the form attributes. The slots of the form have facets which hold generic knowledge of the slot: the Value facet specifies the means to compute the value of the slot; the Menu facet constrains the value to a specified class; the Default facet specifies the means to compute alternative default values; the Defn facet holds a textual definition of the slot. The slots of the instance have facets which hold instance data: the Filler facet holds the instance value of the slot; the Cards and Ingreds facets hold integrity pointers. The form and the instance each have the same set of slots with complementary facets, in the one case containing purely generic, and in the other, purely instance information. To interpret an instance, the instance must be combined with the form, thus creating a complete frame containing both the generic and instance data. Each complete frame is a separate computational environment, having both local state in the slot values, and local process in the slot methods. A form is a discrete unit of knowledge. This modularity of forms allows changes to be made to an individual form without requiring compensating changes in the rest of the system.

The system program—called the "shell" because it contains no application knowledge—is entirely separate from the form definitions. The form definitions together form a knowledge base which is an independent definition of the description domain. Related instances together form a description. As the value of a slot may itself be an instance, a description takes the shape of a hierarchy of nested instances. This description hierarchy may extend many levels, the more extended levels of the hierarchy representing greater detail in the description.

The frame representation allows data or procedures to be attached to the slots of the form. The most important of these is the slot method—the procedure for computing the slot value. The slot evaluations are object-oriented in that the evaluation procedure is attached directly to the object of the evaluation. In addition the computations are data-directed, the value of the slot being computed as a side-effect of some other action. In the description system two actions result in slot evaluations: the action of creating an instance causes all parameter slots of the instance to be evaluated; the action of requesting the value of a slot causes that particular slot to be evaluated. In data-directed programming it is a request for a data value that causes an evaluation, rather than a request for an evaluation.
that causes a data value—data is explicit while process is implicit. Instances interact by querying for the slot values of other instances; evaluation of a slot method activates queries to ingredient slot values. The dependency relations created by these queries are recorded so that the integrity of all values can be actively maintained when ingredient values are updated. Updates in the description are implemented as a two-step process, a deletion of dependent values followed by a re-evaluation. The dependency relations form a network along which the value deletions are propagated. A bulletin board is used to record deleted slots which are terminal nodes of the dependency network; a query for the values of these key slots causes re-evaluation of the whole network of deleted values.

Default values give the description an illusion of completeness but do not substitute for hard values. Defaults are implemented as soft defaults, that is, default values are clearly identified as such. Each form slot contains the seed of its instantiation in the Menu facet, which indicates the good fillers of the slot. A default value is an arbitrary selection from among these possible values. Default values may be applied to both parameter and property slots. As the quality of the default value varies with the degree of dependence on other hard values, alternative defaults are accommodated. Default values must be fully instantiated to be accepted. Integrity maintenance preserves the best default value.

The description system offers an interactive explanation facility. The user may at any time call forth a display of the forms in the knowledge base or a display of any instance. The explanation facility also offers some in-context help during the creation of a description. For the most part, interaction between the user and the system is channelled through queries—the description system queries the user for parameter values, while the user queries the system for property values. During the creation of a description the user will be queried for the parameter values of the instances being created. At the prompt for a parameter value, the user may request some explanatory material concerning the slot value. This explanatory material includes a short textual definition of the slot, a menu of permissible slot values, and the default value of the slot. The complete menu of permissible slot values—which includes global forms, local forms and global instances—is compiled from information contained in the Menu facet which strictly constrains the slot value. The user may obtain the definition of any item presented in the menu and may select the slot value directly from the menu. The slot explanations are provided without altering the context of the current interaction.
Chapter 11: Conclusion

An example knowledge base has been presented in the domain of building structures. The knowledge base includes knowledge about the properties of simple building components and the properties of an assembly of such components. Properties considered include geometric properties (volume, mass, cross-section properties of linear components), loads and load transfer, and an interface to a solids modelling program. A series of examples demonstrates the ease with which the knowledge base can be incrementally modified to extend and enrich the description domain. The power of expressing description in terms of higher level abstractions permits concise description, while reduction of these higher level abstractions to more general abstractions, can be used to free the description from the rigid specialization constraints. The description system provides a way of describing an artefact in terms of explicitly defined abstractions. The knowledge base is an independent representation of the description domain. The aspects of knowledge presently implemented in the description system include the generation of property values, integrity maintenance, default values, value constraint and explanation.

As the domain knowledge of the description system is contained entirely in the knowledge base, the system has applicability to other disciplines besides structural engineering. The flexibility of the system is in being able to explicitly define the objects in the application domain in terms of their parameters and properties. Thus, in applying the system to the design of VLSI, one would establish forms for the various components and their relationships. Similarly, in the design of mechanical systems, one would define classes of objects which can be combined to create the design artefact. By supplying the description system with the necessary parameters of the design artefact, the system can make use of the explicitly defined knowledge to provide the user with the design properties, i.e., the design performance, of the artefact.

11.2 Recommendations

The knowledge representation technique of frames has proved valuable in investigating several aspects of design and description. Its potential has, however, has been only partially explored. There are many directions in which the frames representation can be extended and improved to increase the capabilities of the description system.
The description system comprises several levels of program abstraction, each of which is subject to further development. The following discussion will consider three levels: (i) the program code of the description system; (ii) the definition of individual slot facets and their features; and (iii) the knowledge base, which defines the application of the description system to a single description domain. The first two of these levels are currently combined in the description system shell, but an argument will be presented for their separation.

Previous discussion has purposely omitted all but the most general reference to the program code of the implementation. While the existing program algorithms accurately realize the conceptual model of the system, the efficiency of the implementation can be improved. The development of suitably efficient algorithms is properly a task for the dedicated programmer and falls outside the scope of this work.

One might nevertheless consider some aspects of the implementation, particularly, the advantages of implementing the description system in an object-oriented language. Currently, the description system program is written in Scheme, a modern dialect of Lisp. This language was chosen for its suitability for symbolic computing (for this same reason, Lisp is a popular language for Artificial Intelligence applications, which includes frames and expert systems). The advantages of an implementation in an object-oriented language are a more consistent use of object-oriented computation throughout the description system and a user interface more in keeping with the "external myth" of the system, based on the inherent features of object-oriented languages, such as browsers for classes and instances. The description system has many similarities with other object-oriented languages which could be exploited in the implementation.

Although the interactions between the user and system, and interactions between slots, are both object-oriented and data-directed, the actual slot evaluations are Lisp expressions which take as arguments other Lisp expressions. This inconsistency in the fundamental programming approach to internal computations exists because the many essential primitive evaluation operations have not been translated to consistent frame operations—for example, to be consistent, the basic operation of addition should be implemented by defining a form called "Sum" which is a kind of number which takes two numbers as parameters and has the value of their sum as a property. As it is, the description system is an embedded language which depends on the host language for its basic evaluations. The system benefits from this relationship by having access to the wide-ranging computing powers of Lisp. It also benefits
from the efficiency of evaluations carried out in the host language as these are not subject to the extra overhead costs of the evaluations performed within the description system. The disadvantage of this relationship is that the representation of the knowledge base is tied to the implementation language of the system shell, an undesirable circumstance. Ideally, the knowledge base should be independent of the shell implementation.

There are also advantages to introducing an additional level of program abstraction between the implementation language and the system shell. This programming level would define a general purpose frames language. This language would set the general layout of forms and instances, slots and facets, and evaluation modes, but would not assign individual facets to the slots. The system shell would be implemented on top of this general frames language, defining the individual facets and their functions as they apply to the needs of description. The addition of this intermediate language would make it easier to extend the features of the system shell by refining the features of individual facets or by introducing new facets.

With regard to the slot facets, there are many aspects of knowledge that have not been fully addressed in the present system. Among these one can consider value constraints, dependency-directed backtracking for resolution of constraint violations, frame matching, and certainty factors.

Currently, slot values are constrained by the Menu facet. This constraint is absolute. Design constraints however must have some flexibility. While design is in progress values that violate constraints should be tolerated subject to later resolution of the constraints violations. Constraints themselves are not often absolute and resolution of constraint violations may involve trade-offs between inconsistent constraints. The frames representation should be able to comfortably accommodate value constraints as a facet of knowledge attached to individual slots.

The dependency network, which propagates value updates in the description system, can be used to support another important feature called “dependency-directed backtracking.” This feature allows the testing of hypothetical ingredient values. If dependant values based on these hypothetical ingredients violate design constraints, the system can backtrack along the dependency network to change the responsible ingredient value and propagate a new set of dependent values. This process fits in well with the definition of design: the
designer proposes a hypothetical solution to the design problem, then checks whether this solution satisfies the design constraints; should the design fail to satisfy these constraints the proposed solution is revised. This process of design revision is precisely that of backtracking. The design values (the performance of the designed artefact) must satisfy the design constraints (the performance criteria). Knowledge of the dependency network allows the backtracking facility to zero in on the ingredient values that contribute to the constraint violations.¹

Currently, the description system shell supports a process for reducing an instance of a form to an instance of that form's AKO parent form, i.e., reducing the level of abstraction of the instance. The complementary process of increasing the level of abstraction of an instance is not supported. That it is not is due to the way specializations are created. A specialization is based on slot methods which permit the evaluation of the parameters of the AKO parent. Specialization therefore is closely linked to the concept of reduction. To support the reverse process, the specialization must be expressed as a true constraint. For example, currently a Square is defined as a Rectangle whose breadth is computed as a function of its depth. Specialization in this case is a functional relationship. In terms of a true constraint relationship, a Square would be defined as a Rectangle whose breadth equals its depth. It is possible, given this second definition to determine if an instance of Rectangle is also an instance of Square by checking if the constraint is satisfied. The value of being able to increase the level of specialization of an instance would be in taking advantage of the more efficient property methods and the specialized knowledge that is available in the specialization but not in the more general case. The process of increasing the level of specialization is one of frame matching; an instance of a known class is matched to the various possible sub-classes by evaluating the specialization constraints.

Another aspect of knowledge that may be added to the description system is the use of approximate methods for computing design values. All engineering computations are approximate to a greater or lesser extent; all involve modelling of natural processes by means

¹ An application of backtracking in design systems is demonstrated by the expert system "VT" which designs elevators. "VT solves its problems by constructing an approximate elevator design and successively refining it. As it extends the design, VT also builds a dependency network that records for each value which other values are used to obtain it. The dependency network ... is enough to identify all contributors to a violated constraint and the value it constrains. These contributors represent potential points to backtrack to in order to revise the proposed design" [Marcus87].
of simplifying assumptions. The accuracy of any design value depends on the accuracy of the ingredient values and the accuracy of the computation. Not all design values are required to be accurate at all stages of design. For example, in the sketch-plan stage of design it is common to use approximate methods to get a rough description of the design artefact. This description is later refined with more accurate ingredient data and more rigorous computations.

In accommodating approximate values it is important to indicate some measure of the accuracy of these values. In expert systems, a similar measure is commonly called the "certainty factor." Use of certainty factors would obviate the need for default values—default values would simply be values with a low certainty factor. Selection of the proper method to use in computing a value would depend on the certainty required for the value and the certainty of the available ingredient values. If only a low certainty is required, or if ingredients values of high certainty cannot be obtained, a less rigorous computation can be used.

In Civil Engineering design the major concern is reliability. Accuracy of the design computations is only one aspect of reliability. The important design constraints are inequality constraints; for example a beam must be strong enough to support the applied load, but may exceed that strength. It is possible therefore to compensate for the inaccuracy of a computation by adjusting other factors that contribute to the reliability; for example, by increasing the factor of safety. The description system should therefore include a measure of reliability for the design values and the factors by which it is determined. Although implementation of a measure of reliability presents many difficulties, the frames representation is a promising vehicle for exploring this interesting area.

In suggesting improvements to the current system one must also consider extensions to the knowledge base. The example knowledge base contains only a very small part of the knowledge that can be associated with the domain of building structures. The domain can be further developed to include both more general knowledge and more specific knowledge: general knowledge is associated with the most general forms; specific knowledge is associated with the appropriate sub-classes of these forms. Thus, for example, the form for solids incorporates general knowledge about the behaviour of solids, while more specialized forms, such as the forms for beam and column, incorporate the local knowledge associated directly with these special types. Specializations are important to the overall scope of
the knowledge base as engineering knowledge is often limited to the study of special cases and cannot always, because of its empirical nature, be applied to the general case. The specialization hierarchy allows knowledge to be associated with the appropriate level of abstraction, with knowledge of special cases supplementing general knowledge.

11.3 Conclusion

The unique contribution of this thesis is in applying the frames model of knowledge representation to represent engineering design artefacts in terms of the knowledge that is associated with them. The knowledge of objects is thus inseparable from their representation.

As description is the focus of design, attaching knowledge to the objects of description is a means of integrating the design process. This simple idea, that knowledge should be attached to the objects of knowledge, is the basic tenet of the frames approach to knowledge representation. Thus, knowledge of beams is attached to an object called “beam” and knowledge of structures is attached to an object called “structure.” A description, which is a collection of objects, is also a collection of knowledge about those objects. Endowed with knowledge, the description becomes a model of the described artefact; instead of the description having to be interpreted by outside processes, the description is able to interpret itself. While the description system is presently empowered only to answer queries regarding some basic but important properties of the described artefact, the potential exists for much more sophisticated self-interpretation.

To be of practical use, the implementation of the description system requires refinement in several important areas: The efficiency of the implementation of the description system shell must be increased to speed evaluation. The user interface must be improved to more graphically represent the relation of forms within the specialization hierarchy and the hierarchical relation of instances within a description. The user interface should also allow the user to freely browse among the forms and instances. The extent of knowledge in the knowledge base must be developed to express more of the wide range of general and specific knowledge associated with the description domain; this domain may also be extended. The description system will have to relate to a greater number of independent
application programs as internal computations are better suited to data-intensive, rather than computation-intensive, applications.

Given these necessary refinements in the implementation, the description system has much to offer in a working environment.

- The system offers ease of description. Description is a matter of selecting appropriate abstractions. Higher level abstractions can be used to describe structures of a standard type with minimum input. These standard views can be refined by relaxing the specialization constraints associated with high level abstractions and introducing exceptions. The manner of description follows the top-down approach to design; the overall shape of the design is established before the details are worked out.

- Abstractions can be customized to the current project. Because the abstractions are explicitly defined in the knowledge base they can be customized either by updating existing forms or by adding new ones. The inheritance mechanism offers an easy way to define specializations. Access to description properties is not complicated by the increased levels of abstractions as properties are shared throughout a class and remain constant.

- The description system is data-directed. The description data is explicit while the processes computing the data are implicit. The user therefore can deal with the description data directly without knowledge of the data processes. The processes themselves can be updated without affecting other processes as their interaction is channelled solely through the description data.

- The description system integrates design knowledge into a single system. Knowledge is associated with particular classes of objects and these classes are linked in a single hierarchy of knowledge. Knowledge is explicitly represented and can be incrementally modified to encompass an ever greater domain. Knowledge is represented simultaneously at a number of levels of abstraction. Specific knowledge associated with specializations complements knowledge associated with more general classes. The description system recognizes that there are many aspects of knowledge and provides facets for dealing with each of these aspects. The system can be modified to handle new facets.

Engineering knowledge is diffuse and extensive. To encapsulate the knowledge pertaining to even a very limited description domain presents a considerable challenge. Practically,
one must recognize the inherent difficulties of representing such knowledge in a form that can be processed by a computer. In practice, the design process strongly tends to produce artefacts that are easy to describe and analyse. Knowledge is the limiting factor. Only an increase in knowledge can lead to a wider variety of practical design solutions.

This work has been motivated by a desire to throw new light on some familiar issues in engineering design by applying unfamiliar computing techniques. The frames approach offers a model for design integration that recognizes that there are many aspects of knowledge, and one that is able to accommodate the inevitable need for modification and extension of the richness of the representation and the scope of the domain of knowledge.
Appendix A
Knowledge Base KB0

This appendix presents the set of basic form definitions which are the basis for further
extensions presented in Appendices B and C.

(define-form `(ENTITY
  `(defn "A class of all things defined as forms"))

(define-form `(OBJECT
  `(ako entity)
  `(defn "A physical object--WARNING: THIS FORM IS INCOMPLETE!"))

(define-form `(COMPONENT
  `(ako object)
  `(defn "A discrete three-dimensional object")
  `(slots
    `(geometry
      (value ?)
      `(defn "3d shape of the component")
      `(menu shape-3d))
  `(material
    (value ?)
    `(defn "what it's made of")
    `(menu material))
  `(volume
    (value (geometry:volume))
    `(weight
      (value get-weight))
    `(methods
      `(get-weight
        `(let
          `((volume (geometry:volume))
            `(density (material:density))
            (* volume density))))

(define-form `(ASSEMBLY
  `(ako object)
  `(defn "An assembly of physical objects")
  `(slots
    `(parts
      `(defn "The components of the: assembly")
      (value ??)
      `(menu object))

166
(weight
  ($value (apply + (00 parts:weight)))
  ($default "value")
(volume
  ($value (apply + (00 parts:volume))))
(csg
  ($value (pad12-assemble (00 parts:geometry:csg)))
  ($default "value")))

(define-form '((SHAPE-2D
  (ako entity)
  (defn "A 2d shape lying in the xy plane")
  (slots
   (reference-point
     ($value ?)
     ($menu vector-xy)
     ($defn "location of reference point w.r.t. shape origin")
     ($default origin-xy))
   (rotation
     ($value ?)
     ($menu number)
     ($default 0.0)
     ($defn "rotation about reference point in degrees c.c.w.")
   (area
     ($value ?)
     ($menu number)
     ($default "value")
     ($defn "area of shape")
   (c-of-g
     ($value ?)
     ($menu vector-xy)
     ($defn "center of gravity of shape")
   ref-cofg
     ($defn "location of c-of-g w.r.t. reference point before rotation")
     ($value ref-vector)
     ($transition? #T)
     ($menu vector-xy))
   (ref-cofg-r
     ($defn "location of c-of-g w.r.t. reference point after rotation")
     ($value rot-vector)
     ($transition? #T)
     ($menu vector-xy))
   (cpg
     ($value ?)
     ($default "value")
     ($menu cpg)))

(local-forms
 (ref-vector
  (ako vector-xy)
  (protected? #t)
  (slots
   (x (value get-x))
   (y (value get-y)))
  (methods
   (get-x
    (let ( (cx (0 <:c-of-g:x)) (rx (0 <:reference-point:x)))
       (- cx rx)))
   (get-y
    (let ( (cy (0 <:c-of-g:y)) (ry (0 <:reference-point:y)))
       (- cy ry)))
   (rot-vector
    (ako vector-xy)
    (protected? #t)
    (slots
     (x (value get-x))
     (y (value get-y)))
    (methods
     (get-x
      (let ( (rx (0 <:ref-cfg:x))
            (ry (0 <:ref-cfg:y))
            (rot (0 <:rotation)))
       (rotate-vector-x rx ry rot)))
     (get-y
      (let ( (rx (0 <:ref-cfg:x))
            (ry (0 <:ref-cfg:y))
            (rot (0 <:rotation)))
       (rotate-vector-y rx ry rot))))))

(define-form ' (TRIANGLE
 (ako shape-2d)
 (defn "A triangle")
 (slots
  (d (value ?))
  ($default 100)
  ($defn "Height of the apex above base")
  ($menu number))
 (b (value ?))
  ($default 100)
  ($defn "Length of base")
  ($menu number))
(a (\$value ?)
  (\$default 50)
  (\$defn "Horizontal location of apex relative to left corner of base")
  (\$menu number))
(c (\$value (\$let ( (b (\$0 b)) (a (\$0 a)) )
  (- b a)))
  (\$default "value")
  (\$defn "Horizontal location of apex relative to right corner of base")
  (\$transition? #t))
(area (\$value get-area))
(c-of-g (\$value center))
(cpg (\$value get-cpg)
  (\$default "value")))
(local-forms
(belowapex
  (ako origin-xy)
  (defn "The point on base direct y below apex") )
  :definition refpoint
center
  (ako vector-xy)
  ('defn "The center of area")
  (slots
    (x (\$value get-x))
    (y (\$value get-y))
  )
  (methods
    (get-x
      (\$let ( (c (\$0 c)) (a (\$0 a)) )
        (/ (- c a) 3.0)))
    (get-y
      (\$let ( (d (\$0 d)) )
        (/ d 3.0)))
  )
apex
  (ako vector-xy)
  (defn "The apex of the triangle")
  (slots
    (x (\$value 0.0))
    (y (\$value (\$0 < :d)))
  )
(bottomleft
  (ako vector-xy)
  (defn "The left corner of the base")
  (slots
    (x (\$value (\$let ( (a (\$0 a)) ) (+ a -1))))
    (y (\$value 0.0)))
(bottomright
  (ako vector-xy)
  (defn "The right corner of the base")
  (slots
    (x (\$value (\$let ( (a (\$0 a)) ) (+ a 1))))
    (y (\$value 0.0)))

(define-form '(BUILT-UP-SHAPE-2D
  (ako shape-2d)
  (defn "A 2d shape built up from other 2d shapes")
  (slots
    (item ($value ??)
      ($defn "A component of the built up shape")
      ($menu item))
    (area ($value get-area)
      ($menu number))
    (c-of-g ($value center)
      ($menu vector-xy))
  (get-area
    ($let ( (b (* a b)) (d (* d d)) )
      (/ (* b d) 2.0 ))
    (get-cpg
      ($let ( (a (* a a)) (b (* b b)) (c (* c c)) (d (* d d))
        (refx (0 reference-point:x))
        (refy (0 reference-point:y))
        (rot (0 rotation)) )
      (cond
        ( (= a 0.0)
          (list 'cpg (list 'tri b d) 'at (list (* -1 refx) (* -1 refy) rot)) )
        ( (= a b)
          (list 'cpg (list 'cpg (list 'tri d b) 'at ' (0.0 0.0 90.0))
                    ' at (list (* -1 refx) (* -1 refy) rot) ))
        ( (and (> a 0.0)
                     (< a b))
          (list 'cpg (list 'un (list 'cpg (list 'tri c d) 'at ' (0.0 0.0 90.0))
                        (list 'cpg (list 'tri d a) 'at ' (0.0 0.0 90.0)) )
                     ' at (list (* -1 refx) (* -1 refy) rot) ))
        ( (< a 0.0)
          (list 'cpg (list 'dif (list 'cpg (list 'tri c d) 'at ' (0.0 0.0 0.0))
                        (list 'cpg (list 'tri (minus a) d) 'at ' (0.0 0.0 0.0)) )
                     ' at (list (* -1 refx) (* -1 refy) rot) ))
        ( (> a b)
          (list 'cpg (list 'dif (list 'cpg (list 'tri d a) 'at ' (0.0 0.0 90 0))
                        (list 'cpg (list 'tri d (minus c)) 'at ' (0.0 0.0 90.0))
                     ' at (list (* -1 refx) (* -1 refy) rot) )) ) ) ) )

Appendix A: Knowledge Base KB0  170
(reference-point
  (default center))
(cpg
  (value get-cpg)))
(methods
  (get-area
    (let ( (areas (0 item:cross-section:area)))
      (apply + areas)))
  (get-cpg
    (let ( (cpg (0 item:cpg))
      (refx (0 reference-point:x))
      (refy (0 reference-point:y)))
      (cpg-union cpg (list (* -1 refx) (* -1 refy) 0))))))
(local-forms
  (center
    (ako vector-xy)
  (slots
    (x (value get-x))
    (y (value get-y)))
  (methods
    (get-x (let ( (xs (0 item:moments-of-area:x))
      (areas (0 item:cross-section:area)))
      (/ (apply + xs) (apply + areas))))
    (get-y (let ( (ys (0 item:moments-of-area:y))
      (areas (0 item:cross-section:area)))
      (/ (apply + ys) (apply + areas))))))
(item
  (ako entity)
  (slots
    (cross-section
      (value ?)
      (menu shape-2d)
      (defn "The cross-section of the part")
    (location
      (value ?)
      (menu vector-xy)
      (defn "The loc'n of the part's ref-point wrt the origin of b-u-s")
    (c-of-g
      (value cofg-vector)
      (menu vector-xy))
    (moments-of-area
      (value moments-vector)
      (menu vector-xy))
    (cpg
      (value get-cpg)))

(methods
get-cpg
  (let ( (cpg (@ cross-section:cpg))
         (locx (@ location:x))
         (locy (@ location:y)))
  (move-cpg cpg (list locx locy 0))))
(local-forms
cfg-vector
  (ako vector-xy)
  (protected? #t)
  (slots
   (x ($value get-x))
   (y ($value get-y)))
  (methods
   (get-x
    (let ( (locx (@ <:location:x))
           (refx (@ <:cross-section:ref-cfg-rot:x)))
         (+ locx refx)))
   (get-y
    (let ( (locy (@ <:location:y))
           (refy (@ <:cross-section:ref-cfg-rot.y)))
         (+ locy refy))))
(moments-vector
  (*\^ vector-xy)
  (protected? #t)
  (slots
   (x ($value get-x))
   (y ($value get-y)))
  (methods
   (get-x
    (let ( (x (@ <:c-of-g:x))
           (area (@ <:cross-section:area)))
         (+ x area)))
   (get-y
    (let ( (y (@ <:c-of-g:y))
           (area (@ <:cross-section:area)))
         (+ y area))))
(define-form '(QUADRILATERAL
  (ako built-up-shape-2d)
  (defn "A quadrilateral built up from two triangles")
  (slots
   (b ($value .)
    ($menu number)
    ($defn "The length of the base")))
(defn "The height of the left apex")
(defn "The height of the right apex")
(defn "The horizontal offset of the left apex")
(defn "The horizontal distance between apexes")
(defn "Vector location of point C of the quad")
(defn "The vector location of right apex of quad")
(defn "The point on the base below the center of area")
(defn "The cross-section of triangle")

defn "The location of the origin of the triangle"

defn "Reference point of the triangle"

defn "Rotation of the triangle"
(methods
  (get-d (0 d:dr))
  (get-b (0 b:b))
  (get-a
    (0 let ( (a (0 a:a)) (c (0 c:c)))
    (+ a c))))))

(item2 (ako item)

(slots
  (cross-section ($value triangle2))
  (location ($value origin-xy))

(local-forms
  (triangle2 (ako triangle)
    (slots
      (d ($value get-d))
      (b ($value get-b))
      (a ($value get-a))
      (reference-point ($value bottomleft))
      (rotation ($value get-rot))

    (methods
      (get-b (0 get:c:length))
      (get-d
        (0 let ( (dl (0 dl:dl))
          (dr (0 dr:dr))
          (a (0 a:a))
          (c (0 c:c))
          (length (0 get:c:length)))
        (/ (- (* dl (+ a c)) (+ dr a)) length)))
      (get-a
        (0 let ( (a (0 a:a))
          (c (0 c:c))
          (dl (0 dl:dl))
          (dr (0 dr:dr))
          (length (0 get:c:length)))
        (/ (+ (* a (+ a c)) (* dr dl)) length))
      (get-rot
        (0 let ( (a (0 a:a))
          (c (0 c:c))
          (dr (0 dr:dr)))
        (stand (/ dr (+ a c)))))))))

(defn "A trapezoid")

(ako quadrilateral)
(slots
  (d ($value ?)
      ($defn "The height of the shape")
      ($menu number) )
  (dl ($value (0 d))
      ($default "value")
      ($transition? #T))
  (dr ($value (0 d))
      ($default "value")
      ($transition? #T)) )))

(define-form '(PARALLELOGRAM
  (ako trapezoid)
  (defn "A parallelogram")
  (slots
   (c ($value (0 b))
      ($default "value")
      ($transition? #T)) )))

(define-form '(RECTANGLE
  (ako parallelogram)
  (defn "A rectangle. Local origin at bottom left corner.")
  (slots
   (a ($value 0.0)
      ($transition? #T))
   (d ($default (let ( (b (0 b)) ) (* b 1.5))
      100 ))
   (b ($default (let ( (d (0 d)) ) (/ d 1.5))
      75))
   (reference-point ($default center)))
  (local-forms
   (topcenter
    (ako vector-xy)
    (defn "The point on the top above the center of area")
    (slots
     (x ($value (let ( (b (0 <:b)) ) (/ b 2))))
     (y ($value (0 <:d))))
   (bottomcenter
    (ako vector-xy)
    (defn "The point of the bottom below the center of area")
    (slots
     (x ($value (let ( (b (0 <:b)) ) (/ b 2))))
     (y ($value 0))))
   (center
    (ako vector-xy)
    (defn "The center of area of the shape")
(slots
  (x (value (let ( (b (0 <: b)) ) (/ b 2)))
  (y (value (let ( (d (0 <: d)) ) (/ d 2)))))))

(methods
  (get-area (let ( (b (0 b)) (d (0 d)) )
             (* d b)))
  (get-cpg
   (let ( (b (0 b))
           (d (0 d))
           (x (0 reference-point:x))
           (y (0 reference-point:y))
           (rotation (0 rotation)))
     '(cpg (rect .b ,d) at ( ,minus x) ,minus y ,rotation))))))

(define-form '(SQUARE
   (ako rectangle)
   (defn "a square")
   (slots
    (b (value (0 d))
     (default -value)
     (transition? #T))
    (d (default "The length of each side")))))

(define-form '(R100x75
   (ako rectangle)
   (defn "A rectangle depth=100 breadth=75 mm")
   (slots
    (d (value 100))
    (b (value 75))
    (reference-point (default center)))))

(define-form '(WSHAPE
   (ako built-up-shape-2d)
   (slots
    (d (value ?)
     (menu number)
     (defn "depth of section")
    (b (value ?)
     (menu number)
     (defn "flange width")
    (w (value ?)
     (menu number)
     (defn "web thickness")
    (th (value ?)
     (menu number)
     (defn "flange thickness"))

(...)
(reference-point
  ($value ?)
  ($menu vector-xy)
  ($default center))
(item
  ($transition? #T)
  ($value top-flange bottom-flange web)))
(local-forms
  (top-flange
    (ako item)
    (slots
      (cross-section ($value rectangle1))
      (location ($value location1))))
(local-forms
  (rectangle1
    (ako rectangle)
    (slots
      (d ($value (0 <=: th)))
      (b ($value (0 <=: b)))
      (rotation ($value 0))
      (reference-point ($value topcenter))))
(location1
  (ako vector-xy)
  (slots
    (x ($value 0))
    (y ($value (0let ( (d (0 <=: d))
                         (/ d 2)))))))
(web
  (ako item)
  (slots
    (cross-section ($value rectangle2))
    (location ($value location2))))
(local-forms
  (rectangle2
    (ako rectangle)
    (slots
      (d ($value (0let ( (d (0 <=: d)) (th (0 <=: th))
                          (- d (* th 2))))))
      (b ($value (0 <=: w)))
      (rotation ($value 0))
      (reference-point ($value center))))
(location2
  (ako vector-xy)
  (slots
    (x ($value 0))
    (y ($value 0))))
(bottom-flange
 (ako item)
 (slots
  (cross-section ($value rectangle3)))
  (location ($value location3)))
 (local-forms
  (rectangle3
   (ako rectangle)
   (slots
    (d ($value (0 <= $:th)))
    (b ($value (0 <= $:b)))
    (rotation ($value 0))
    (reference-point ($value bottomcenter))))
 (location3
  (ako vector xy)
  (slots
   (x ($value 0))
   (y ($value (0let ( (d (0 <= $:d))
                     (/ d -2)))))
 (center
  (ako vector-xy)
  (slots
   (x ($value 0))
   (y ($value 0)) ))
 (topcenter
  (ako vector-xy)
  (slots
   (x ($value 0))
   (y ($value (0let ( (d (0 <= $:d))
                     (/ d 2))))))
 (bottomcenter
  (ako vector-xy)
  (slots
   (x ($value 0))
   (y ($value (0let ( (d (0 <= $:d))
                     (/ $:d -2)))) ))
 (define-form 'SHAPE-3D
  (ako entity)
  (defn "A 3d shape defined in cartesian space")
  (slots
   (volume
    ($value ?)
    ($default *value))))
(define-form 'SLEE
  (ako shape-3d)
  (defn "A straight line extruded element")
  (slots
    (line-vector
      ($value ?)
      ($defn "Characteristic line vector of element")
      ($menu line-vector-3d))
  (cross-section
    ($value ?)
    ($default R100x75)
    ($defn "Cross-section shape of element")
    ($menu shape-2d))
  (length
    ($value (line-vector:length))
    (volume
      ($value get-volume))
      (csg
        ($value get-csg)
        ($default -value))
      (methods
        (get-csg
          (let ( (len (leng 'b))
            (x (line-vector:end1:x))
            (y (line-vector:end1:y))
            (z (line-vector:end1:z))
            (dx (line-vector:vector:x))
            (dy (line-vector:vector:y))
            (dz (line-vector:vector:z))
            (cpg (cross-section:cpg)))
            (let ( (c1 (plane-to-solid cpg len)))
              (cond ( (null? c1) nil)
                ( (and (= dx 0) (= dx 0))
                  (cond ( (> dy 0)
                    `(csg (csg ,c1 at (0 0 0 -90 90 0))
                      at ,(x ,y ,z 0 0 0))))
                  ( else
                    `(csg (csg ,c1 at (0 0 0 90 90 0))
                      at ,(x ,y ,z 0 0 0))))
                ( else
                  (let* ( (degx (* -1 (asind (/ dy len))))
                      (degy (asind (/ dx (* (cosd degx) len))))
                      `(csg (csg ,c1 at (0 0 0 ,degx ,degy 0))
                        at ,(x ,y ,z 0 0 0)))))))
    )
  )
)
(get-volume
  (let ((length 0 length))
    (area (0 cross-section:area))
    (* length area))))

(define-form '(MATERIAL
  (ako entity)
  (defn "A homogeneous solid material")
  (slots
    (density
      ($value ?)
      ($menu number)
      ($defn "density is N/mm"^3")))))

(define-form '(STEEL
  (ako material)
  (defn "Normal structural steel")
  (slots
    (density
      ($value 76.9e-6))))) ;N/mm^3

(define-form '(CONCRETE
  (ako material)
  (defn "Normal density concrete")
  (slots
    (density
      ($value 23.5e-6)))))) ;N-mm^3

(define-form '(ECTOPLASM
  (ako material)
  (defn "A fictitious material of density 10e-6 N/mm^3")
  (slots
    (density
      ($value 10e-6))))) ;N/mm3

(define-form '(VECTOR-3D
  (ako entity)
  (defn "A 3d vector in cartesian space")
  (slots
    (x ($value ?)
      ($menu number))
    (y ($value ?)
      ($menu number))
    (z ($value ?)
      ($menu number))
    (length ($value get-length)
      ($menu number))))
(methods
  (get-length
    (let ((x (0 x)) (y (0 y)) (z (0 z)))
      (sqrt (+ (* x x) (+ (* y y) (- z z)))))))
(defin-form 'VECTOR-DIFFERENCE-3D
  (ako vector-3d)
  (defn "The difference of two 3d local-forms")
  (slots
    (vector1 ($value ?)
      ($menu vector-3d)
      ($defn "The first vector in difference")
    (vector2 ($value ?)
      ($menu vector-3d)
      ($defn "The second vector in difference")
    (x ($value get-x))
    (y ($value get-y))
    (z ($value get-z)))
  (methods
    (get-x
      (let ((x1 (0 vector1.x)) (x2 (0 vector2:x)))
        (- x2 x1)))
    (get-y
      (let ((y1 (0 vector1:y)) (y2 (0 vector2:y)))
        (- y2 y1)))
    (get-z
      (let ((z1 (0 vector1 z)) (z2 (0 vector2:z)))
        (- z2 z1))))
  (define-form 'VECTOR-XY
    (ako vector-3d)
    (defn "A 2D vector in the X: plane")
    (instances)
    (subclasses)
    (slots
      (x ($value ?)
        ($menu number))
      (y ($value ?)
        ($menu number))
      (z ($value 0)
        ($transition? #T))
      (length ($value get-length)))
    (methods
      (get-length
        (let ((x (0 x))
          (y (0 y)))
          (sqrt (+ (* x x) (+ (* y y)))))))
(define-form '(ORIGIN-XY
  (ako vector-XY)
  (defn "The vector (0 0) in the XY plane")
  (slots
    (x ($value 0))
    (y ($value 0)))))

(define-form '(VECTOR-ZX
  (ako vector-3d)
  (defn "A vector in the ZX plane")
  (slots
    (y ($value 0)
      (transition? #T))))))

(define-form '(ORIGIN-ZX
  (ako vector-zx)
  (defn "The origin in the ZX plane, i.e. x=0, z=0"
  (slots
    (x ($value 0))
    (Z ($value 0)))))

(define-form '(HORIZONTAL-LINE_INTERSECTION
  (ako vector-zx)
  (defn "Column defining location in ZX plane")
  (slots
    (line-1
      ($value ?)
      ($menu horizontal-line .column-line column-line-2pt)
      ($defn "First of two column line defining location")
    (line-2
      ($value ?)
      ($menu horizontal-line .column-line column-line-2pt)
      ($defn "Second of two column line defining location")
      (x ($value get-x))
      (z ($value get-z)))
  (-methods
    (get-x
      (let ( (x1 (e line-1:point:x))
      (z1 (e line-1:point:z))
      (r1 (e line-1:rotation))
      (x2 (e line-2:point:x))
      (z2 (e line-2:point:z))
      (r2 (e line-2:rotation))
      (cadr (line_intersection z1 x1 r1 z2 x2 r2))))
    (get-z
      (let ( (x1 (e line-1:point:x))
      (z1 (e line-1:point:z)))
(define-form '(LINE-VECTOR-3D
  (ako entity)
  (slots
    (end1
      ($value ?)
      ($menu vector-3d))
    (end2
      ($value ?)
      ($menu vector-3d))
    (vector
      ($value vector-dif)
      ($menu vector-3d))
    (length ($value (vector: length))))
  (local-forms
    (vector-dif (ako vector-difference-3d)
      (protected? #T)
      (slots
        (vector1 ($value (0ref <:end1)))
        (vector2 ($value (0ref <:end2)))))))

(define-form '(HORIZONTAL-LINE-VECTOR-3D
  (ako line-vector-3d)
  (defn "A horizontal line vector, ie p1l to xz plane")
  (slots
    (elevation
      ($value ?)
      ($defn "Elevation of hz plane of line vector")
      ($menu elevation))
    (end1-xz
      ($value ?)
      ($defn "Location of end1 of line vector")
      ($menu vector-xz))
    (end2-xz
      ($value ?)
      ($defn "Location of end2 of line-vertex")
      ($menu vector-xz))
    (end1
      ($value vector1)
      ($transition? #T)))
(end2
  ($value vector2)
  ($transition? #T))
)
(local-forms
  (vector1
    (ako vector-3d)
    (protected? #T)
    (slots
      (x ($value (0 <:end1-zx:x)))
      (y ($value (0 <:elevation:y)))
      (z ($value (0 <:end1-zx:z)))
    )
  )
  (vector2
    (ako vector-3d)
    (protected? #T)
    (slots
      (x ($value (0 <:end2-zx:x)))
      (y ($value (0 <:elevation:y)))
      (z ($value (0 <:end2-zx:z)))
    )
  )
)
(define-form ' (VERTICAL-LINE-VECTOR-3D
  (ako line-vector-3d)
  (defn "A vertical line-vector")
  (slots
    (end2-y
      ($value ?)
      ($defn "y-coordinate of end2")
      ($menu number))
    (end2
      ($value vector2)
      ($transition? #T))
  )
)
(local-forms
  (vector2
    (ako vector-3d)
    (protected? #T)
    (slots
      (x ($value (0 <:end1:x)))
      (y ($value (0 <:end2-y)))
      (z ($value (0 <:end1-z)))
    )
  )
)
(define-form ' (HORIZONTAL-LINE
  (ako entity)
  (defn "A horizontal line defined by a single point and a rotation
about the Y axis relative to the Z axis")
  (slots
    (point
      ($value ?)
    )
  )
)
(define-form '(HORIZONTAL-LINE-2PT
  (ako horizontal-line)
  (defn "A horizontal line define by two points")
  (slots
    (p1 ($value ?)
      ($menu vector-zx)
      ($defn "A point on the line")
    (p2 ($value ?)
      ($menu vector-zx)
      ($defn "A point on the line")
    (point
      ($value (0 p1)))
    (rotation
      ($value get-rotation)))
  (methods
    (get-rotation
      (0let ((x1 (0 p1:x))
        (z1 (0 p1:z))
        (x2 (0 p2:x))
        (z2 (0 p2:z))
            (polar-angle (- z2 z1) (- x2 x1))))))
)
Appendix B
Knowledge Base KB1

This appendix presents the form definitions which, combined with those of Appendix A, are used in performing the first example description of Chapter 10.

(define-form '(ELEVATION
  (ako entity)
  (defn "Elevation above datum in mm")
  (slots
    (y ($value ?)
      ($defn "Elevation above datum in mm")
      ($menu number))))

(define-form '(COLUMN-LINE
  (ako horizontal-line)
  (defn "A horizontal line used to define column locations")))

(define-form '(COLUMN-LINE-2PT
  (ako horizontal-line 2pt)
  (defn "A horizontal line used to define column locations")))

(define-form '(COLUMN-LINE-INTERSECTION
  (ako vector-zx)
  (defn "Column defining location in ZX plane")
  (slots
    (column-line-1
      ($value ?)
      ($menu column-line column-line-2pt)
      ($defn "First of two column line defining location")
    column-line-2
    ($value ?)
    ($menu column-line column-line-2pt)
    ($defn "Second of two column line defining location")
    (x ($value get-x))
    (x ($value get-z)))
  (methods
    (get-x
      (@let ((x1 (0 column-line-1:point:x))
      (x1 (0 column-line-1:point:x))
      (r1 (0 column-line-1:rotation))
      (x2 (0 column-line-2:point:x)))))

186
(z2 (o column-line-2:point:x))
(r2 (o column-line-2:rotation))
(cadr (line-intersection z1 x1 r1 z2 x2 r2)))

(get-x
  (@let (  (x1 (o column-line-1:point:x))
            (z1 (o column-line-1:point:x))
            (r1 (o column-line-1:rotation))
            (x2 (o column-line-2:point:x))
            (z2 (o column-line-2:point:x))
            (r2 (o column-line-2:rotation))
          )
          (car (line-intersection z1 x1 r1 z2 x2 r2))))

(define-form '(CONNECTION
  (ako nil)
  (defn "A connection attached to a beam or column")
  (slots
    (component
      ($value ?)
      ($menu beam column foundation)
      ($defn "Host component")
      ($if-added add-component)
      ($if-removed remove-component))
    (location
      ($value select-rel-loc)
      ($force? #t))
    (load ($value get-load))
    (load-moment ($value get-load-moment))
  )
  (methods
    (get-load nil)
    (get-load-moment
      (if (instance-of? 'beam (o component))
        (@let (  (l (o location:1))
                (p (o load))
                (* p l))
                nil))
      (select-rel-loc
        (@let (  (conn (o component))
        (cond ( (instance-of? 'column conn)
               'crl)
               ( (instance-of? 'beam conn)
                  'brl)
               ( (instance-of? 'foundation conn)
                  'frl)
               ( else nil )))))))
(define-form '(COLUMN
(ako nil)
(defn "A vertical building column")
(slots
  (base-connection
    ($value base-conn)
    ($menu connection))
  (top-elevation
    ($value ?)
    ($menu elevation))
  (connections
    ($value ???)
    ($menu connection))
  (cross-section
    ($value ?)
    ($menu shape-2d)
    ($default r100x75))
  (material
    ($value ?)
    ($menu material))
  (geometry ($value column-geometry) ($menu shape-3d))
)(local-forms
  (column-geometry
    (ako sles)
    (slots
      (line-vector ($value lv1))
      (cross-section ($value @ref <:cross-section)))))
)(local-forms
  (lv1
    (ako vertical-line-vector-3d))}
(slots
  (endi ($value v1))
  (end2-y ($value (0 <=:top-elevation:y))))
(local-forms
  (vi
    (ako vector-3d)
    (slots
      (x ($value (0 <=:base-connection:location:x)))
      (y ($value (0 <=:base-connection:location:y)))
      (z ($value (0 <=:base-connection:location:z))))))

(base-conn
  (ako connection)
  (methods
    (get-load
      (apply + (0 <=:connections:load))) ))
)

(define-form '(COLUMN-RELATIVE-LOCATION
  (ako vector-3d)
  (defn "Location in 3space given by column and elevation")
  (slots
    (column ($value ?))
    ($menu column)
    ($defn "Column id")
    (elevation ($value ?))
    ($menu elevation)
    ($defn "Elevation")
    (col-loc ($value (ref column:base-connection:location)))
    (x ($value (0 column:base-connection:location:x)))
    (y ($value (0 elevation:y)))
    (z ($value (0 column:base-connection:location:z))))
(local-forms
  (column-location1
    (ako column-location)
    (slots
      (column ($value (ref <=:column)))))
)

(define-form '(BEAN
  (ako nil)
  (defn "A beam")
  (slots
    (connection1
      ($value conn1)
      ($menu connection)
      ($defn "Connection that supports end1 of the beam")
    (connection2
      ($value conn2)
(connections
  ($value 0)
  ($menu connection))
(point-loads
  ($value 0)
  ($menu point-load))
(distributed-loads
  ($value 0)
  ($menu dist-load))
cross-section
  ($value 0)
  ($defn "Cross-section shape of beam")
  ($default r100x75)
  ($menu shape-2d))
(material
  ($value 0)
  ($menu material)
  ($defn "Beam material"))

(line-vector
  ($value lv)
  ($menu line-vector-3d))
(length
  ($value get-length))
(csg ($value (oref geometry:csg)))
(volume ($value (oref geometry:volume)))
(weight ($value get-weight))
(load-moment
  ($value get-load-moment)
  ($menu number))
total-load
  ($value get-total-load))
geometry ($value beam-geometry)
  ($menu shape-3d))
(methods
  (get-length
    (0 line-vector:length))
  (get-weight
    (0let ( (density (0 material:density))
                (volume (0 geometry:volume)))
          (* volume:density)))
  (get-load-moment
    (0let ( (c-moments (apply + (0 connections:load-moment))))
(d-moments (@apply + (00 distributed-loads:load-moment)))  
(pl-moments (@apply + (00 point-loads:load-moment)))  
(+ c-moments pl-moments d-moments))

(get-total-load  
(+ (@apply + (00 point-loads:load))  
   (@apply + (00 distributed-loads:load))  
   (@apply + (00 connections:load))) ) )

(local-forms

(beam-geometry
  (ako slots
    (slots
      (cross-section ($value (eref <:cross-section)))
      (line-vector ($value (eref <:line-vector))))))

(lv
  (ako line-vector-3d)
  (slots
    (end1 ($value (eref <:connection1:location)))
    (end2 ($value (eref <:connection2:location)))))

(conn1
  (ako connection)
  (methods
    (get-load
      (@let ( (moment (@ <:load-moment))
                (total-load (@ <:total-load))
                (length (@ <:length))
                (- total-load (/ moment length)))))))

(conn2
  (ako connection)
  (methods
    (get-load
      (@let ( (moment (@ <:load-moment))
                (length (@ <:length)))
            (/ moment length)))))))

(point-load
  (ako nil)
  (defn "Point load")
  (slots
    (load ($value ?)
      ($menu number)
      ($defn "Magnitude of point load in kN")
    (location ($value ?)
      ($menu number)
      ($defn "Distance from end of beam")
    (load-moment ($value get-load-moment)
      ($menu number)) )

)
(methods
  (get-load-moment
    (@let ( (p (0 load))
           (d (# location)))
       (* p d)))
  (dist-load
    (ako nil)
    (defn "A uniform load distributed on portion of beam")
  (slots
    (11 ($value ?)
        ($menu number)
        ($defn "Location along beam of start of dist load")
    (12 ($value ?)
        ($menu number)
        ($defn "Location along beam of end of dist load")
    (dist-load ($value ?)
        ($menu number)
        ($defn "Value of load in kN/m")
    (load ($value get-load))
    (load-moment ($value get-load-moment))
  (methods
    (get-load
      (@let ( (11 (0 11))
           (12 (0 12))
           (w (0 dist-load)))
             (/ (* w (- 12 11)) 1000.))
    (get-load-moment
      (@let ( (11 (0 11))
             (12 (0 12))
             (w (0 dist-load)))
             (/ (* w (- (* - 12 12) (0 11))) 2000)))
  (full-dist-load
    (ako dist-load)
    (defn "Uniform load distributed on full length of beam")
  (slots
    (11 ($value 0))
    (12 ($value (0 < length)))))
  (define-form 'BEAM-RELATIVE-LOCATION
    (ako vector-3d)
    (defn "Location in 3-space relative to beam")
  (slots
    (beam
      ($value ?)
      ($menu beam)
      ($defn "The beam")))
(1
  ($value ?)
  ($menu number)
  ($defn "Distance along from endi of the beam")
  (x ($value get-x))
  (y ($value get-y))
  (z ($value get-z)))
(methods
  (get-x
    (def (get-x)
      (let ((xi (0 beam:line-vector:end1:x)))
        (x (0 beam:line-vector:vector:x))
        (len (0 beam:line-vector:length))
        (l (0 l))
        (+ xi (* x (/ l len))))))
  (get-y
    (def (get-y)
      (let ((yi (0 beam:line-vector:end1:y)))
        (y (0 beam:line-vector:vector:y))
        (len (0 beam:line-vector:length))
        (l (0 l))
        (+ yi (* y (/ l len))))))
  (get-z
    (def (get-z)
      (let ((zi (0 beam:line-vector:end1:z)))
        (z (0 beam:line-vector:vector:z))
        (len (0 beam:line-vector:length))
        (l (0 l))
        (+ zi (* z (/ l len))))))))
(define-form '(FOUNDATION
  (ako nil)
  (slots
    (elevation ($value ?) ($menu elevation))
    (location-zx ($value ?) ($menu vector-zx))
    (connections ($value @??) ($menu connection))
    (location ($value locn) ($menu vector-3d))
    (load ($value (@apply + (@@ connections:load)))))
  (local-forms
    (locn
      (ako vector-3d)
      (slots
        (x ($value (0 <:location-zx:x))))
        (y ($value (0 <:elevation:y))))
        (z ($value (0 <:location-zx:z))))
  )
  (define-form '(FOUNDATION-LOCATION
    (ako vector-3d)
    (defn "Location of foundation")
(slots
  (foundation
    ($value ?)
    ($menu foundation))
  (x ($value (0 foundation:location:x)))
  (y ($value (0 foundation:location:y)))
  (z ($value (0 foundation:location:z))))

(define-form 'STRUCTURE
  (ako assembly)
  (defn "A building structure frame consisting of columns and beams")
  (slots
    (column-lines ($value ??)
      ($menu column-line)
      ($defn "The structure column lines")
    (foundations ($value ??)
      ($menu foundation)
      ($defn "The column supports")
    (columns ($value ??)
      ($menu column)
      ($defn "The structure columns")
    (beams ($value ??)
      ($menu beam)
      ($defn "The structure beams")
    (parts ($value (cons 'mf (append (; columns) (; beams))))) ))))
Appendix C
Knowledge Base KB3

This appendix presents the form definitions which, combined with those of Appendices A and B, are used in performing the second example description of Chapter 10.

```
(define-form '(Beam2

(ako beam)

(defn "A beam simply supported by two columns")

(slots

  (column-1
   ($value ?)
   ($menu column)
   ($if-added (get-value id '((connection1 *) (component *)) nil))
   ($defn "Column supporting end-1 of beam")

  (column-2
   ($value ?)
   ($menu column)
   ($if-added (get-value id '((connection2 *) (component *)) nil))
   ($defn "Column supporting end-2 of beam")

  (elevation
   ($value ?)
   ($menu elevation)
   ($defn "Beam elevation")

  (connection1
   ($value conn1-2)
   ($menu conn1-2))

  (connection2
   ($value conn2-2)
   ($menu conn2-2))

  (local-forms

    (conn1-2

      (ako conn1)

      (slots

        (component ($value (0 <: column-1))

        ($force? #T))

        (location ($value locn)

        ($menu locn)))

    (local-forms

```
Appendix C: Knowledge Base KB

(locn
  (ako column-relative-location)
  (slots
    (column ($value (0 :<: column-1)))
    (elevation ($value (0 :<: elevation)))
  ))

( Conn2-2
  (ako conn2)
  (slots
    (component ($value (0 :<: column-2))
      ($force? #T))
    (location ($value locn)
      ($menu locn))
  )
  (local-forms
    (locn
      (ako column-relative-location)
      (slots
        (column ($value (0 :<: column-2)))
        (elevation ($value (0 :<: elevation))))
    ))

(define-form 'Joist
  (ako beam)
  (defn "A beam simply supported on two beams")
  (slots
    (beam-1
      ($value ?)
      ($menu beam)
      ($if-added (get-value id '((connection1 *)) nil))
      ($defn "Beam supporting end1 of joist")
    )
    (1-1
      ($value ?)
      ($menu number)
      ($defn "Location of connection along supporting beam-1")
    )
    (beam-2
      ($value ?)
      ($menu beam)
      ($if-added (get-value id '((connection2 *)) nil))
      ($defn "Beam supporting end2 of joist")
    )
    (1-2
      ($value ?)
      ($menu number)
      ($defn "Location of connection along supporting beam-2")
    )
    (connection1
      ($value conn1-2)
      ($menu conn1-2)
    )
    (connection2
      ($value conn2-2)
      ($menu conn2-2)
    )
  )
(local-forms
  (conn1-2
   (ako conn1)
   (slots
    (component
     ($value (0 <: beam-1))
     ($force? #T))
    (location
     ($value locn)
     ($menu locn)))
   (local-forms
    (locn
     (ako beam-relative-location)
     (slots
      (beam ($value (0 <: beam-1))
      (l ($value (0 <: 1-1)))))
    (conn2-2
     (ako conn2)
     (slots
      (component
       ($value (0 <: beam-2))
       ($force? #T))
      (location
       ($value locn)
       ($menu locn)))
      (local-forms
       (locn
        (ako beam-relative-location)
        (slots
         (beam ($value (0 <: beam-2))
         (l ($value (0 <: 1-2)))))
      (define-form 'Joist-ev
        (ako joist)
        (defn "A joist parallel to the global X-axis")
        (slots
         (z ($value ?)
          ($menu number)
          ($defn "Perpendicular distance of joist from global X-axis")
          (l-1 ($value get-11))
          (l-2 ($value get-12)))
          (methods
           (get-11
            ($let (z (z (? z)))
             (l (0 beam-1:length))
(z1 (beam-1:line-vector:end1:z))
(z2 (beam-1:line-vector:end2:z))
(if (zero? (- z2 z1))
  #f
  (+ 1 (/ (- z z1) (- z2 z1))))

(get-12
  (let ((z (0 z))
        (l (beam-2:length))
        (z1 (beam-2:line-vector:end1:z))
        (z2 (beam-2:line-vector:end2:z))
        (if (zero? (- z2 z1))
            #f
            (+ 1 (/ (- z z1) (- z2 z1)))))

(define-form 'Column2
  (ako column)
  (defn "Column with a column")
  (slots
   (foundation
    ($value ?)
    ($menu foundation))
   (base-connection
    ($value base-conn2)
    ($force? #t)
    ($menu base-conn2))
  (local-forms
   (base-conn2
    (ako base-conn)
    (slots
     (component ($value (0 0 foundation))
      ($force #t)))))

(define-form 'Column-line-nn
  (ako column-line)
  (defn "A column line")
  (slots
   (x ($value ?) ($menu number))
   (point ($value pt))
   (rotation ($value 0)))
  (local-forms
   (pt (ako vector-zx)
    (slots
     (x ($value (0 0 0)))
     (z ($value 0))))))
(define-form 'Column-line-ew
  (ako column-line)
  (defn "A east pointing column line")
  (slots
    (x ($value ?) ($menu number))
    (point ($value pt))
    (rotation ($value 90)))
  (local-forms
    (pt (ako vector-zx)
      (slots
        (x ($value 0 ))
        (z ($value (0 <:z ))))))
)

(define-form 'Structure2
  (ako structure)
  (defn "A regular structure")
  (slots
    (no-of-bays-NS
      ($value ?)
      ($menu number)
      ($defn "Number of bays North-South")
    )
    (bay-spacing-NS
      ($value ?)
      ($menu number)
      ($defn "Bay spacing North-South")
    )
    (no-of-bays-EW
      ($value ?)
      ($menu number)
      ($defn "Number of bays East-West")
    )
    (bay-spacing-EW
      ($value ?)
      ($menu number)
      ($defn "Bay spacing East-West")
    )
    (no-of-floors
      ($value ?)
      ($menu number)
      ($defn "Number of floors")
    )
    (floor-spacing
      ($value ?)
      ($menu number)
      ($defn "Spacing of floors")
    )
    (no-of-joists-per-bay
      ($value ?)
      ($menu number)
      ($defn "Number of joists per bay")
    )
  )
(elevation-lst ($value get-el-lst))
(base-elevation
  ($value get-base-el)
  ($menu elevation))
(top-elevation
  ($value get-top-el)
  ($menu elevation))
(colines-ns-lst
  ($value get-column-lines-NS))
(colines-ew-lst
  ($value get-column-lines-EW))
(foundation-matrix
  ($value get-fndn-mat))
(column-matrix
  ($value get-col-mat))
elevations
  ($value (multi-filler (squash (@ elevation-lst)))))
colines-lines
  ($value (multi-filler (append (@ colines-NS-lst)
                               (@ colines-EW-lst)))))
foundations
  ($value (multi-filler (squash (@ foundation-matrix)))))
columns
  ($value (multi-filler (squash (@ column-matrix)))))
beams
  ($value get-beams))

methods
(get-el-lst
  (@let ( (n (@ no-of-floors))
           (s (@ floor-spacing)))
        (cons '1st (make-elevations n s))))
(get-top-el
  (@let ( (lst (@ elevation-lst)))
         (last lst)))
(get-base-el
  (@let ( (lst (@ elevation-lst)))
         (car lst)))
(get-column-lines-NS
  (@let ( (s (@ bay-spacing-NS))
           (n (@ no-of-bays-NS)))
        (cons '1st (make-colines-ns n s))))
(get-column-lines-EW
  (@let ( (s (@ bay-spacing-EW))
           (n (@ no-of-bays-EW)))
        (cons '1st (make-colines-ew n s))))
(get-fndn-mat
  (let ( (colines-ew (0 colines-ew-1st))
        (colines-ns (0 colines-ns-1st))
        (base-el (0 base-elevation)))
      (cons '1st (make-fndns colines-ew colines-ns base-el))))

(get-foundations
  (let ( (fndn-mat (0 foundation-matrix)))
      (squash fndn-mat)))

(get-col-mat
  (let ( (fndn-mat (0 foundation-matrix))
         (top-el (0 top-elevation))
         (cons '1st (make-columns fndn-mat top-el)))))

(get-beams
  (let ( (col-mat (0 column-matrix))
         (floor-els (cdr (0 elevation-1st)))
         (no-of-joists (0 no-of-joists-per-bay))
         (bay-spacing (0 bay-spacing-ew))
         (multi-filler (make-floors-ew col-mat
                                          floor-els no-of-joists bay-spacing))))'
)
Appendix D
A Note on the Program Code and Computing Resources

The description system shell is written in PCScheme [Scheme87a] [Scheme87b], a lexically scoped dialect of Lisp, and is run on an IBM-PC/AT-compatible computer with extended memory. The program makes use of the extended memory version of Scheme. Interface to the PADL-2 solids modelling program [Hartquist83] is by manual file transfer. The PADL-2 program is run on an Apollo workstation with the Aegis operating system.

The description system shell and example knowledge base occupy 165 kbytes on a 5½-inch diskette. The shell program and knowledge base are available from:

Department of Civil Engineering
Carleton University
Ottawa, Canada
K1S 5B6
Glossary

This glossary gives a brief definition of terms used in the thesis. The numbers in parentheses indicate the pages where the terms are introduced.

Abstraction (24), a view of an object with detail omitted. Abstraction hierarchy (26), a hierarchy of views of an object which are progressively more detailed.

Aggregation (26), a type of abstraction in which a group of objects is treated as a single object.

AIO (38), the “an instance of” relation of instance to form.

AKO (28), the “a kind of” relation of sub-class to super-class.

Asked-token (66), a special slot filler, equivalent to a null value, which prevents re-evaluation of the slot.

Class (27), a set of similar objects. Class definition (27), the generic description of a class; a form. Sub-class (28), a class defined as a specialization of another class in relation to that class. Super-class (28), a class in relation to its sub-classes.

CPG (99), constructive plane geometry; the corresponding 2D implementation of CSG (constructive solid geometry). CPG-list (99), the corresponding 2D implementation of CSG-list.

CSG (97), constructive solid geometry; a system of modelling solids as combinations of primitive shapes. CSG-list (97), the internal representation of CSG solids in the description system.

Default (73), a value which is used in place of an absent input value. Hard default (73), a default value which is used as though it were an input value. Soft default (73), a default value which is used differently from an input value.

Demon (21), a process which is performed as a side-effect of another process.

Dependant (66), a slot whose value depends on the value of another slot. Dependency link (66), the link between functionally related slots; integrity pointer.
Description (47), a collection of related instances. Description hierarchy (48), a hierarchy of nested instances which describes a single artefact.

Facet (35), the sub-association of a slot which contains aspects of knowledge about the slot. Form facet (41), a facet containing generic knowledge which is a sub-association of the slots of a form. Instance facet (41), a facet containing instance knowledge which is a sub-association of the slots of an instance. Cards facet (66), the instance facet which holds integrity pointers to dependent slots. Default facet (44), the form facet which specifies the means of computing the slot default value, with alternatives. Defn facet (44), the form facet which holds a short textual definition of the slot for use by the explanation facility. Filler facet (43), the instance facet which holds the value of the slot. Force facet (131), the form facet which indicates a slot, not obviously a parameter slot, that is to be evaluated in forward-chaining mode. If-added facet (35), the form facet which holds a procedure to be evaluated when a value is added to the slot. If-removed facet (35), the form facet which holds a procedure which is evaluated when the slot value is deleted. Ingreds facet (66), the instance facet which holds integrity pointers to ingredient slots. Menu facet (44), the form facet which specifies the class of permissible fillers of the slot. Transition facet (92), the form facet which indicates a slot that is used for internal computations only and which is not displayed to the user. Value facet (43), the form facet which specifies the means of computing the slot value.

Filler (slot filler) (35), the value of a slot. Good fillers (84), the set of fillers that satisfy the Menu facet constraint. Special filler (76), an association with a key recognized by the description system, e.g., 'default' for default values, 'asked' for the asked-token, 'csg' for a csg-list.

Form (27), the essential attributes of a class of objects; a class description implemented as a frame consisting of slots which represent class attributes, containing generic knowledge only. Form definition (86), a definition of a form in the knowledge base. Global form (42), a form which is defined and accessed globally. Local form (42), a form which is defined within another form and only accessible within that form.

Frame (18), a data structure for representing stereotypical objects, situations or events, consisting of a number of associations called 'slots' each of which consists of a number
of associations called ‘facets;’ each slot represents an attribute of the stereotypical situation; each facets represents an aspect of knowledge associated with the attribute.

Frames (18), a knowledge representation technique.

Generalization (26), a type of abstraction in which similar objects are regarded as a generic object, ignoring many individual differences between objects.

Ingredient (66), a slot whose value contributes to the computation of the value of another slot.

Inheritance (44), the process by which a sub-class shares the properties and associated knowledge of its super-class.

Instance (27), an instance of a form, implemented as a frame containing only instance data—to be interpreted an instance must be combined with its form. Global instance (105), an instance which is globally defined and accessed. Local instance (105), an instance which is created local to another instance, and accessible only within that instance.

Integrity (63), the correspondence of functionally related data. Integrity pointers (66), the pointers maintained by the description system to keep track of the functional relations between slot values.

Knowledge base (47), a collection of forms which defines a description domain.

Method (42), a Lisp expression used to compute a slot value.

Parameter (27), one of the attributes of a form that suffice to define an instance of the form.

Property (27), an attribute of a form that is computed from the parameter values.

Query (55), a request to system for a slot value. Ask-user query (57), a request from the system to the user for a parameter value.

Shell (47), the basic description system program, separate from the knowledge base, free of domain knowledge. Shell function (86), a Lisp function defined in the shell and used as a system command.

Slot (35), a sub-association of a form or an instance which represents a single attribute; each slot has a number of facets.
Slot value (35), the value of the slot, stored in the Filler facet of the instance.

Specialization (28), the process of creating new class of objects by applying constraints to an existing class; the class thus created.

Substantial value (75), a value that cannot be further resolved, i.e., a numerical or logical value, or a fully instantiated frame.
Bibliography

[Abbott87]

[Abelson85]

[ACM85]

[Aikins83]

[Atwood85]

[Baer79]

[Barbuceanu84]

[Barbuceanu85]

[Barr81]


[Buchmann79]

[Buchmann82]

[Buchmann84]

[Button84]

[BYTE81]

[Cercone87]

[Charlesworth76]

[CISC80]

[Cohen83]

[Collins80]
Collins, E. B. "Experience in the use of the BDS system (OXSYS) on a major project in

[Computer83]

[Cornick85]

[Coyne88]

[Daniel79]

[Darden87]

[Date86]

[Davies80]

[Davis85]

[Davison78]
[Davison80]


[Dillon87]


[Dym88]


[Eastman76a]


[Eastman76b]


[Eastman78]


[Eastman79]


[Eastman80]

[Eastman81a]


[Eastman81b]

———. "Recent developments in representation in the science of design". *ACM IEEE 18th Design Automation Conference Proceedings*.

[Encarnacao83]


[Fenves82]


[Fenves85]


[Fikes85]


[Filman88]


[Forsyth84a]


[Forsyth84b]


[Fox86]

Fox, M. S., J. M. Wright and D. Adam. "Experience with SRL: an analysis of a

[Frost86]

[Genesereth85]

[Gero85a]

[Gero85b]

[Gero87]

[Glass84]

[Glass86]

[Glass82]

[Goldberg84]


[Hayes79]

[Hayes-Roth83]

[Hayes-Roth84]

[Hayes-Roth85]

[Hendler88]

[Hunt87]

[Intergraph86]

[Intergraph87]

[Jackson86]

[Kalay85]
[Kerschberg86]

[Kerschberg89]

[Lafue78]

[Lafue82]

[Lafue83]

[Latombe78]

[Leesley78]

[Leesley82]

[Lewin79]
[Maher84]

[Marcus87]

[Minsky75]

[Mitschang89]

[Morgenstern89]

[Oxford83]

[Pham88]

[Plato63]

[Port84]
[Preiss80]

[Preiss83]

[Pugh76]

[Rasdorf85]

[Rauch-Hindin86]

[Reddy86]

[Rehak84]

[Requicha80]

[Richens76]
Richens, P. "New developments for the OXSYS system". CAD76: 2nd Int. Conf. on

[Richens78]

[Romanycia85]

[Rouse87]

[Rubinstein84]

[Rush86]

[Sandberg81]

[Schaefer82]

[Schank75]

[Scheme87a]
[Scheme87b]

[Schueller83]

[Smith77]

[Spoonamore82]

[Sriram85]

[Sriram86]

[Stefik79]

[Stefik83a]

[Stefik83b]

[Stefik86a]
[Stefik86b]


[Tesler81]


[Thompson78]


[Thompson80]


[Tichy87]


[Tomiyama84]


[Trickett82]


[Walker88]


[Waterman86]

[Winston77]

[Winston81]

[Winston84]

[Winston87]

[Yasky80]

[Zadeh88]