Modules for Design Support

Katy Börner (Ed.)
University of Freiburg
Center for Cognitive Science

FABEL-Report No. 35
Synopsis

This report summarizes major work carried out in the FABEL project in 1994/1995. Starting with a characterization of the domain particularities and desired support, eight approaches to assessment and design are presented. The theoretical background, the basic idea accomplished with illustrative examples, and the implementation details are provided. A comparison of the approaches concludes the report. Two approaches are already integrated in the second FABEL-prototype. The remaining approaches will be integrated into the third FABEL-prototype to be ready for evaluation in autumn 1995.

This research was supported by the Federal Ministry of Education, Science, Research and Technology (BMBF) within the joint project FABEL under contract no. 01IW104. Project partners in FABEL are GMD – German National Research Center for Information Technology, Sankt Augustin, BSR Consulting GmbH, München, Technical University of Dresden, HTWK Leipzig, University of Freiburg, and University of Karlsruhe.

BSR Consulting GmbH
Wirtstraße 38
81539 München
Ansprechpartner: Dr. Brigitte Bartsch-Spörl
Fax: 089/695158
Tel.: 089/695545
E-mail: brigitte@bsr-consulting.de

GMD – Forschungszentrum
Informationstechnik GmbH
Institut für Angewandte Informationstechnik
Forschungsbereich Künstliche Intelligenz
53754 Sankt Augustin
Ansprechpartner: Dr. Angi Voß
Fax: 02241/14-2384
Tel.: 02241/14-2726
E-mail: angi.voss@gmd.de

Technische Universität Dresden
Fakultät Informatik
Institut Künstliche Intelligenz
01062 Dresden
Ansprechpartner: Doz. Dr. Uwe Petersohn
Fax: 0351/4575-335
Tel.: 0351/4575-431
E-mail: peterson@freia.inf.tu-dresden.de

HTWK Leipzig (FH)
FB Informatik, Mathematik u. Naturwissenschaften
Universitäts-Studiengang Informatik
Postfach 66
04251 Leipzig
Ansprechpartner: Prof. Dr. Klaus P. Jantke
Fax: 0341/328028
Tel.: 0341/3928-426
E-mail: jantke@informatik.th-leipzig.de

Universität Freiburg
Institut für Informatik und Gesellschaft
Abt. Kognitionswissenschaft
Friedrichstr. 50
79098 Freiburg
Ansprechpartner: Prof. Dr. Gerhard Strube
Fax: 0761/203-4938
Tel.: 0761/203-4934, -4933 (Skr.)
E-mail: strube@cognition.iig.uni-freiburg.de

Universität Karlsruhe
Institut für Industrielle Bauproduktion
Englerstr. 7
76128 Karlsruhe
Ansprechpartner: Dr. Ludger Hovestadt
Fax: 0721/661115
Tel.: 0721/608-2167
E-mail: ludger@ifib1.ifib.uni-karlsruhe.de
Contents

I Introduction 1

1 Design Support in Industrial Building Design 3
Katy Börner
1.1 Industrial Building Design ............................................. 3
1.1.1 ARMILLA, MIDI, A4, DANCER .................................. 3
1.1.2 Task-structure ......................................................... 4
1.1.3 Knowledge .............................................................. 7
1.1.4 Methods ................................................................. 8
1.2 Degrees of Support ...................................................... 9
1.2.1 Assessment Modules .................................................. 9
1.2.2 Adaptation Modules .................................................. 10
1.2.3 Construction Modules ............................................... 11

II Assessment Modules 13

2 Assessing Realizations of Design Tasks (CheckL & CheckUp) 15
Dietmar Janetko and Oliver Jäschke
2.1 The Idea ................................................................. 15
2.2 Knowledge and its Representation .................................... 16
2.2.1 TheKnowledgeRepresentationSchemaA4 .......................... 16
2.2.2 Knowledge Used for Assessing Design Task ...................... 17
2.3 Approach ................................................................. 19
2.4 Example and Implementation ......................................... 21

3 Assessment Supported by a Domain Ontology (DOM) 23
Brigitte Bartsch-Spörl, Shirin Bakhtari and Wolfgang Oertel
3.1 The Idea ................................................................. 23
3.1.1 Underlying Assumptions ..................................... 24
3.2 Knowledge and its Representation ............................ 25
3.3 Approach ............................................................ 27
    3.3.1 Current Status of the Implementation .................. 29
3.4 Two Examples ....................................................... 29

III Adaptation Modules ............................................ 31

4 General Geometric and Topologic Retrieval and Adaptation (TOPO) 33
Carl-Helmut Coulon
  4.1 The Idea .......................................................... 33
  4.2 Example ........................................................... 33
  4.3 Knowledge and its Representation ............................ 35
      4.3.1 Domain-specific extension ............................... 37
  4.4 Approach .......................................................... 37
      4.4.1 Retrieval ................................................... 37
          4.4.1.1 A common maximum clique algorithm ............... 38
          4.4.1.2 The improvement ..................................... 40
      4.4.2 Transfer .................................................... 40
      4.4.3 Correction .................................................. 42
  4.5 Implementation .................................................. 43
      4.5.1 Handling exponential complexity for interactive use . 44
      4.5.2 State of the work on the algorithm and further improvements . 44

5 Adaptation by Active Autonomous Objects (AAA O) 46
Parivash Adami
  5.1 The Idea .......................................................... 46
  5.2 Knowledge and its Representation ............................ 47
  5.3 Approach .......................................................... 47
  5.4 Example ........................................................... 48
  5.5 Implementation .................................................. 48

6 Case Adaptation using Agents (AgentEx) 51
Raghu R Bhat
  6.1 The Idea .......................................................... 51
  6.2 Knowledge and its Representation ............................ 52
6.2.1 Model of Task Space ........................................ 52
6.2.2 Domain Rules ............................................... 53
6.2.3 Spatial relations ............................................. 53
6.3 Approach .......................................................... 54
6.4 Example ............................................................ 54
6.5 Implementation ................................................... 56

IV Construction Modules ................................. 57

7 Analogical Layout Design (SYN) ................. 59
Katy Börner and Roland Faßauer
7.1 The Idea .......................................................... 59
7.1.1 Domain Characteristics ..................................... 59
7.1.2 Memory Organization and Analogical Reasoning .... 60
7.2 Knowledge and its Representation .................... 61
7.3 Approach .......................................................... 62
7.4 Implementation and Example Run .................... 64

8 Exact and Correct Placement of Pipes (ANOPLA) . 69
Wolfgang Gräther
8.1 The Idea .......................................................... 69
8.2 Knowledge and its Representation ....................... 70
8.2.1 Design Objects ............................................. 70
8.2.2 Strategic Plans and Cases ................................. 71
8.2.3 Templates .................................................... 72
8.2.4 Rules and Heuristics ........................................ 73
8.3 Approach .......................................................... 74
8.4 Example ............................................................ 75
8.5 Implementation ................................................... 75

9 Operators Used in Routine Design (Roude) ........ 78
Oliver Jäschke and Dietmar Janetzko
9.1 The Idea .......................................................... 78
9.2 Example ............................................................ 79
9.3 Knowledge and its Representation ....................... 79
9.4 Approach .......................................................... 80
9.5 Implementation ................................................... 80
V Comparison

10 Comparison of the Modules

Katy Börner and Gerhard Strube

10.1 Domain of Support .................................................. 83
  10.1.1 Functionality .................................................... 84
  10.1.2 Scope of Application ........................................... 85
10.2 Domain Knowledge .................................................. 85
  10.2.1 Content .......................................................... 86
  10.2.2 Representation Formalism ...................................... 87
  10.2.3 Dynamics ........................................................ 87
10.3 Concluding Remarks ................................................ 89
Part I

Introduction
Chapter 1

Design Support in Industrial Building Design

Katy Börner

1.1 Industrial Building Design

Industrial building design belongs to the class of difficult real-world problems. Rules or functional explanations of designs and the appropriateness of layouts are rare. Reasoning is often based on prior experience referring to CAD layouts. Storage, retrieval and adaptation of prior layouts require geometrical and topological aspects to be taken into account.

This present chapter is organized as follows: Firstly, we provide an overview of the particular school of industrial building design we are using to motivate and evaluate our work. Borrowed from [Janetzko and Börner, 1993; Janetzko et al., 1994a], we introduce the task-structure that has been developed to organize this complex domain and to offer task-oriented support. In particular, we discuss how the task-structure may be exploited for knowledge acquisition and system design. Finally, this section provides an overview of the basic knowledge and its representation as used for design support.

1.1.1 ARMINILLA, MIDI, A4, DANCER

Our work builds upon results from the Institute of Industrial Building Design, University of Karlsruhe. Several tools have been developed to handle this complex and not well understood geometrical domain. The tools are listet in a chronological order:

Construction kits, among which we chose the MIDI system [Haller, 1974] for medium sized industrial buildings like schools and laboratories (see Fig. 1.1). The kits come with a catalogue of beams and columns suitable for installation according to the graphic rule system ARMINILLA.
ARMILLA [Haller, 1985; Hovestadt, 1993c] is used for the systematic and conflict free layout of various subsystems in a building, such systems being water, air, etc. The system basically consists of components, standardized grids, a hierarchy of pipe systems, predefined reservations in the grids for outlets and inlets, templates to place components within this area and a set of rules which define the placement of these grids in a conflict free manner.

Figure 1.1: Swiss railway company's education center in Murten, designed by Prof. Haller

A4 represents any object occurring in architectural design by its geometrical attributes and its type attributes [Hovestadt, 1992; Hovestadt, 1993a; Hovestadt, 1993b]. The former represent the placement and extension of objects. The latter refers to the subsystems of the building (e.g., c=construction, k=climate, r=room, z=fresh-air); the general function of the area addressed (e.g., a=access, e=development, v=connection); the kind of resolution employed (e.g., b=area, h=bounding-box, t=parts); and the part of the building that is envisaged (e.g., 4=hallway, 6=room, 8=areas within a room). For more details see also [Hovestadt and Schmidt-Belz, 1994] and the first section of chapter 2.

DANCER is a special design editor [Hovestadt, 1993b], which allows the inspection and graphic manipulation of A4 objects graphically. Figure 1.2 shows a projection of the Murten project on the climate system represented on the x and y axes. On the code level the design is represented as a set of attribute-value lists, one for each design object. Different states during the design of a building correspond to different configurations of objects.

1.1.2 Task-structure

Methodologies to decompose complex domains into manageable parts are an active research area. Focusing on the application domain of building design, a number of different tasks have to be done by the system or the user and various methods have to be applied. Additionally, several kinds of knowledge are needed to realize this elaborate support by

---

The ellipses in figures are graphical representations of a cubic bounding-box, and represent the spatial scope of the objects.
a system. In Fabel, results of knowledge engineering promote a task-oriented user support. Based on this decision, we developed a methodology for task-oriented knowledge acquisition and system design [Janetzko et al., 1994a].

Tasks play an important role in research on expert-systems, knowledge acquisition, man-machine interaction, distributed artificial intelligence, work analysis, etc. In general, tasks are considered as something to be achieved, i.e. goals that require problem solving activities [Chandrasekaran et al., 1992]. Tasks differ with respect to granularity and abstraction. Subsequently, we apply the task approach that is dedicated toward modeling problem-solving activities to task-knowledge imbalances in CBR as applied to building design.²

Our usage of the notion of task is concerned with the mapping of aspects of human expertise to knowledge-based systems. Since our concept of tasks has accrued from a bottom-up approach, it is closely tailored to the requirements of our domain. Specifications of input and output, the goal to be accomplished, and methods used to realize the task

²An excellent discussion of the application of a task approach CBR is provided by A. K. Goel (1989)
are also included. Task-structures made up of task-subtask links are also provided. These are enriched by methods, i.e., cases or rules, and by a specification of interactions between tasks. Enlarging a set of tasks to a full-blown task-structure is accomplished by incremental knowledge acquisition.

Figure 1.3: Task-structure: Tasks and their interactions

To establish a task-structure (see Fig.1.3), the overall domain task (e.g., design of supply nets in a building) is at first divided into manageable subtasks. This can be done along the physical components involved in the overall domain task. The resulting task-structure reflects the structure of the physical components that occur in design.

Secondly, dependencies, viz. interactions between tasks, are specified. There are dependencies between two or more tasks if one task cannot be achieved without considering the realization of one or more other tasks.

Thirdly, the input and output of each of the resulting subtasks are specified. In so doing, the flat list of subtasks is replaced by a structure made up of tasks and enabling relations. The task-structure is constructed in collaboration between domain experts and knowledge engineers using knowledge elicitation techniques that range from observation and interviews to highly structured methods (cf. Strube et al., 1994 in press).

Once the task-structure is specified it provides a useful platform for incremental knowledge acquisition, problem solving, and system design. In knowledge acquisition, the elicitation of cases and rules may be advanced in a focused manner. Eliciting cases is supported by referring to instances of tasks as cases; eliciting rules by referring to knowledge that realizes task transitions as rules. A structured assembly of all types of knowledge together
represents a model of the domain. In problem solving, the task-structure provides hints as to the tasks subsequently to be tackled. Thus, the task-structure serves as coordinating knowledge, e.g., for planning. In system design, the task-structure may be used to select tasks that can be supported by the system.

1.1.3 Knowledge

If the task to be modeled and the methods that are applied to the task are selected, the appropriate type of knowledge can hardly be freely chosen. This focusing effect may be exploited in knowledge acquisition. Having selected design as the general task and case based reasoning as the general problem solving method a pressure is exercised to use cases, concept hierarchies to support indexing, and similarity assessment and rules to support modification. On the other hand, it is quite possible to start with selecting the task and the knowledge to be used, thereby constraining the selection of a suitable method.

Cases: During design, architects frequently browse through old drawings. Case based reasoning seems to be an appropriate reasoning method to support design tasks (see [Goel, 1989; Navinchandra, 1991; Hinrichs, 1992; Hua et al., 1993; Domeshek and Kolodner, 1992]). Applying CBR, the domain of discourse is represented by a finite set of already solved cases (stored in the case base CB) and a similarity relation over them. Case based reasoning proceeds as follows. Given a new problem, cases with (modulo the similarity relation) similar problems are selected from a case base. The solution of the most similar case is transferred to the new problem and adapted if necessary. Storing the new problem including its solution and updating the similarity relation can be seen as a kind of learning [Aamodt, 1990]. Detailed introductions into CBR may be found in [Riesbeck and Schank, 1989; Kolodner, 1993].

Up until now, CBR has been applied mainly to analytic tasks. Here, properties or symptoms and corresponding concepts or diagnoses are readily at hand. The former are represented by the problem, the latter constitute the solution to a case. There is an ongoing debate about how to cut cases in synthesis tasks. The issues at stake in this debate are the method to cut cases and as a consequence the grainsize of cases. In building design, cases may be cut ad lib from an overall architectural plan. That is, no particular method is applied. Applying a task-oriented approach to knowledge acquisition, cases are specified as instances of tasks. They provide episodic or specific knowledge about state transitions. Concerning the grainsize, in an application domain like architecture complete buildings have been taken as cases [Goel, 1989; Domeshek and Kolodner, 1992; Hinrichs, 1992]. Again, the task-structure may be employed to achieve units which are better suited to problem solving.

Note that the main reason for employing a well-founded method for cutting cases is not the size of cases as such but the need to come up with cases that are usable in case based reasoning. Cases should reflect and support the way in which architects design buildings. That is, they should allow the tackling of specific problems that are known to recur in the application domain. Embarking on specific problems becomes extremely difficult if the grainsize of the problem to be solved, e.g., the design of the supply air system, and the grainsize of the case, e.g., a complete building, do not match. A mismatch is likely to occur
if no particular method for cutting cases is employed, or oversized units in the domain are viewed as cases.

**Rules:** Apart from *cases*, *rules* are used to accomplish tasks. Rules are derived by generalizing transitions between tasks. Thus, rules provide generic knowledge about task transitions. At the most general level, once a set of conditions on the left hand side of a rule is met, the right-hand-side of the rule will produce a certain effect. This effect can be employed to drive reasoning that spans different types of problem solving like planning, design, or verification. For example, when designing the network for air supply in a certain part of the building, rules are employed to make sure that the task is achieved, i.e., the network for supply air in a certain part of a building is actually realized.

Rules can be used and formulated for quite a range of design tasks, which are more or less 'routine'. Accordingly, when observing and interviewing our domain experts we found that routine construction tasks are those where they indeed apply rules and prefer rule application to the use of cases.

**Models:** The most elaborate kind of knowledge to be acquired is knowledge encapsulated in domain models. Models applicable in supporting such complex domains like industrial building design are rare. Models like ARMILLA and MIDI restrict complexity but are usually partial. They describe physical components, their structural features, and the relations between them that are required to design a building. The task-structure may be seen as the most elaborate model. It represents knowledge about sequences of design steps. It also comprises methods suitable for embarking upon concrete design tasks, viz. cases and rules. Additionally, it contains meta-knowledge about how to combine all knowledge in a consistent and reasonable way. This kind of knowledge may be represented by descriptive interrelations and dependencies between the aforementioned kinds of knowledge. Its purpose is to guide the process of problem solving and the selection of appropriate problem solving methods.

### 1.1.4 Methods

Methods are procedures that implement abstract problem solving models. Methods serve three functions. Firstly, they provide the active component that carries out problem solving. Secondly, methods add constraints and requirements to the knowledge that can be used for solving a task. If, for example, CBR is selected as a general problem solving method, case-knowledge is needed. Thirdly, the method selected imposes a structure on the task resulting in a task-method-subtask decomposition also referred to as task-structure. Note that the decomposition of a task into subtasks is a function of the method applied. For example, using CBR as a problem solving method decomposes the overall task of design into the subtasks storage, retrieval, and adaptation of design. In general, a number of methods may be used to tackle a task [McDermott, 1988]. Which method is chosen and applied to the task hinges on criteria like availability of knowledge, computational costs, and reliability of the solution. A method which is applied at one level of a task-structure may also be applied at another level of the same task-structure.
1.2 Degrees of Support

According to the task tackled and the methods and knowledge the system is equipped with, the system provides different degrees of support. Each degree of support ought to be interactive. That means, the user should always be in a position to intervene in the problem solving process during critical situations, or if he/she is dissatisfied with the performance of the system. We distinguish three main degrees of support involved in building design, *browsing*, *assessment*, and *design*. Corresponding to the reasoning methods applied, *design* may be further divided into *adaptation* and *construction*. Whereas an overview about modules supporting *browsing* has been provided in [Voß, 1994; Voß et al., 1994], this report presents and discusses modules supporting *assessment*, *adaptation*, and *construction*.

The remaining part of this chapter characterizes the three support modes, provides an overview of the corresponding modules, and the kind of tasks which these modules may assist in.

1.2.1 Assessment Modules

The assessment of CAD layouts, i.e., to check geometric object arrangements as to whether or not they meet a set of conditions, is a necessary task to be accomplished before the results of problem solving can be accepted.

![Figure 1.4: Assessment modules](image-url)
**CheckUp and CheckL:** CheckL uses a task-structure to cut technical drawings into cases; it learns which topological predicates hold in these cases. CheckL uses a signature made up of topological predicates. The signature is treated as a knowledge base of CheckL, which may be substituted by a different signature. The knowledge learned is used by CheckUp to assess realizations of design tasks. The predicates provided are especially suited to representing and later on checking topological relations among objects of type *ab4, *vb4, *vb6 and *ab6.

**DOM** uses a sophisticated and elaborate domain ontology modeling to assess the quality of design fragments ranging from single cases to whole buildings. Envisioned tasks are collision and coherence tests of object types *vh4, *vh6, *ab6, *vh5, and *vh7 (see Fig. 1.4).

### 1.2.2 Adaptation Modules

As for adaptation, concrete CAD layouts (cases) are retrieved, matched, and adapted to fit current situations.

![Diagram of Adaptation Modules](image)

**Figure 1.5:** Adaptation modules

**TOPO** does general geometric and topological adaptation. Therefore, a topological representation and statistics about the frequency of topological relations in prior layouts are
extracted. Given a new problem situation, TOPO analyzes the topological and geometric relations between objects. Statistics about prior layouts with similar geometry and topology are used to correct, extend, or detail the problem situation.

**AAA O** adapts columns by applying active autonomous objects. Contrary to the other modules discussed, AAA O arranges a set of pre-given objects (e.g., objects of type `cvb4`, see Fig. 1.5) until they meet a set of pre-given requirements. It does not create new objects.

**AgentEx** provides case adaptation by using agents. Agents are software objects encoded with domain knowledge and capable of spatial actions, and a simple negotiation mechanism. Given a layout of design objects, the objects are instantiated into knowledge-based agents. These agents then align themselves in the new context according to their domain knowledge and the transferred spatial knowledge. AgentEx is specialized in the conflict-free layout of supply air pipes (see Fig. 1.5).

### 1.2.3 Construction Modules

Construction corresponds to the application of a more general knowledge, like case-schemata or rules, in order to arrive at correct design solutions.
SYN* performs structural memory organization as a basis for computationally efficient analogical layout design. Given a set of prior layouts, SYN* uses an algebraic representation to re-represent the layout in terms of their topology. Next, common structures (called prototypes) of layout sets are extracted and used for memory organization. A new problem situation, e.g., objects of type \(vb4\), \(eb4\), \(ab6\) (see Fig. 1.6), is reformulated in terms of these prototypes and compared to them. Given structural similarity, the prototypical solution is applied to solve the new problem through the design of objects \(vb6\).

ANOPLA supports the configuration of pipes in an exact and correct way. It uses generic templates, rules, and heuristics for installations provided by the ARMILLA system to solve this design problem. As depicted in Fig. 1.6, it may be applied to construct objects of type \(vh6\).

ROUDE Roude is a module tailored to carry out routine design tasks like \(*vb4\) out of \(snb4\) and \(*eb4\) or \(*ab6\) provided objects of type \(*ab4\) are given (see Fig. 1.6).
Part II

Assessment Modules
Chapter 2

Assessing Realizations of Design Tasks (CheckL & CheckUp)

Dietmar Janetzko and Oliver Jäschke

2.1 The Idea

Assessing artifacts in design is a necessary task to be accomplished before the results of problem solving can be accepted. To test accomplishments of design tasks, artifacts in design problem solving are scrutinized whether or not they meet a set of conditions [Genesereth, 1984], [Silverman and Wenig, 1993]. In real-world domains, like building design, there is seldom an explicit rationale to prove the correctness and completeness of solutions automatically. Hence, the assessment of design tasks is performed predominantly by human experts. A challenge to knowledge acquisition and machine learning is to acquire and apply knowledge required for testing such that a knowledge-based system may be developed that tests accomplishments of design tasks. The goal of this chapter is to describe an approach for combining manual knowledge acquisition and machine learning in an attempt to deliver knowledge to be used for assessing design tasks. Our approach to build a system that learns and applies knowledge suitable for testing accomplishments of design tasks will be presented in five steps: First, a portrait of the application domain along with the representation scheme employed is given. We present specific and general knowledge human experts apply when testing design tasks. Second, we will delineate the approach from a more general perspective and delineate how the knowledge is brought to bear in assessment. Third, we provide examples of how design tasks are tested by CheckUp, which is a system developed to that purpose [Jäschke and Janetzko, 1994b]. In addition, we describe Check-L, which is an enlargement of CheckUp that uses CAD-based accomplishments of design activities to learn the knowledge required by CheckUp.
2.2 Knowledge and its Representation

2.2.1 The Knowledge Representation Schema A4

In what follows, we make use of A4 [Hovestadt, 1993a], i.e., a representation scheme developed to support computer-based building design. Seen from the viewpoint of the application domain, A4 allows to represent objects used in building design on a graphical level and on a code level. Different states in the workflow of designing a building may be redescribed in terms of different collections and configurations of objects. On the graphical level, A4-objects are represented by geometrical objects, in particular by ellipses. Ellipses are a substitute for circumscribing rectangles. Using ellipses instead of rectangles is an unaccustomed but very useful trick: Ellipses overlap only in a few points. Thus, the readability of drawings becomes essentially improved. On the code level, A4-objects are represented by values for a fixed set of attributes as described in Fig. 2.1.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>:object-id</td>
<td>number used for identification of the object</td>
<td>natural number</td>
</tr>
<tr>
<td>:x</td>
<td>x-coordinate for the left lower edge of the object</td>
<td>decimal number</td>
</tr>
<tr>
<td>:dx</td>
<td>x-extension of the object</td>
<td>decimal number</td>
</tr>
<tr>
<td>:y</td>
<td>y-coordinate for the left lower edge of the object</td>
<td>decimal number</td>
</tr>
<tr>
<td>:dy</td>
<td>y-extension of the object</td>
<td>decimal number</td>
</tr>
<tr>
<td>:z</td>
<td>z-coordinate for the left lower edge of the object</td>
<td>decimal number</td>
</tr>
<tr>
<td>:dz</td>
<td>z-extension of the object</td>
<td>decimal number</td>
</tr>
<tr>
<td>:time</td>
<td>date of creation</td>
<td>decimal number</td>
</tr>
<tr>
<td>:dtime</td>
<td>t-extension</td>
<td>decimal number</td>
</tr>
<tr>
<td>:ttag</td>
<td>date of creation</td>
<td>integers</td>
</tr>
<tr>
<td>:dttag</td>
<td>date of deletion</td>
<td>integers</td>
</tr>
<tr>
<td>:aspect</td>
<td>subsystem of the building</td>
<td>e.g., building, rooms, paths, shaft, climate, used air, supply air</td>
</tr>
<tr>
<td>:morphology</td>
<td>general function of the object</td>
<td>linkage, development, connection, usage</td>
</tr>
<tr>
<td>:precision</td>
<td>kind of resolution that is employed</td>
<td>e.g., area, bounding box</td>
</tr>
<tr>
<td>:scale</td>
<td>part of the building that is envisaged</td>
<td>e.g., 2 (= building), 4 (= floor), 6 (= room), 8 (= areas within a room)</td>
</tr>
</tbody>
</table>

Figure 2.1: A4-Attributes and their possible Values

Seen from a more formal vantage point, A4 is a representational scheme based upon pairs of attributes and values. There is a set of attributes made up of numerical and string-like attributes. The string-like attributes are called type attributes. The values of the numerical attributes describe placement and extension of the A4-object. The values of the type attributes refer to structured annotations of A4-objects. In this chapter we will treat four type
attributes: aspect, morphology, precision, scale. Abbreviations (derived from the German language) of the four type attributes are used throughout this paper. That means, the type will be expressed by three letters and a number. The first letter relates to the type attribute aspect, i.e., the subsystem of the building; the values of this type attribute are a = used air, g = building, k = climate, r = rooms, w = paths, s = shaft, z = supply air. The second letter denotes the type attribute morphology, i.e., the general function of the object; the values of this type attribute are a = linkage, e = development, v = connection, n = usage. The third letter specifies the type attribute precision, i.e., the kind of resolution that is employed; the values of this type attribute are b = area, h = bounding-box. The number relates to the type attribute scale; the part of the building that is envisaged; the values of this type attribute are 2 = building, 4 = floor, 6 = room, 8 = areas within a room. For example, zab6 is the type of the object presented in Fig. 2.2.

\[
\text{:object-id 602 :x 120.001755 :dx 479.99982 :y 120.00674 :dy 479.99982 :z 300 :dz 100 :time 0 :dt ime 0.01 :tt ag 751121990 :dtt ag 999999999 :aspect ZULUF T :morphology ANSCHLUSS :precision BEREICH :scale 6}
\]

Figure 2.2: Example of an object of A4 of type zab6

2.2.2 Knowledge Used for Assessing Design Task

In the previous section we concentrated on the type of representation schema we employ. CheckUp and Check-L can be brought to bear whenever a representation schema based on attributes and values is used. We will now turn to an introduction of the knowledge that is actually extracted from the schema described above and employed for assessment. Our approach to testing rests upon a collection of concepts used to scrutinize the outcome of design activities: The reduct (of an object of A4) is a 5-tuple like (type, x, dx, y, dy) with x, dx, y and dy being numerical entries and type being a symbol as described above. We will use reducts for assessment. Types are also used to represent the tasks that have to be accomplished. This means that the objects of a certain type can be viewed as the accomplishment of the task of this type. If we collect all types we have the full list of tasks at hand that have to be addressed. We will denote the set of tasks by T, elements of T will be called t₁, t₂, ..., tᵢ, ... Whenever we describe the string-like entry of an A4 vector on a syntactical level we use the notion of type. Whenever we refer to problem solving activities like design, planning, or testing we use the concept of task [Janetzko and Börner, 1993].

The 4-tuple of the numerical entries x, dx, y, and dy of a reduct, is an area. By using the numerical components of areas we are able to define predicates that represent relationships of placement between them, e.g., cover, overlap (cf. Fig. 2.3). Based on these area predicates, we are in a position to refer to more complex concepts. We may then define relations between sets of areas, and characterize certain kinds of sets of areas. Sets of areas will be
### Figure 2.3: Predicates Used by CheckUp and Check-L

<table>
<thead>
<tr>
<th>Area / Area</th>
<th>Figure / Area</th>
<th>Predicate</th>
<th>Description</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>O ab</td>
<td>C Xb</td>
<td>area a and area b overlap</td>
<td>the areas in X cover the area b</td>
<td></td>
</tr>
<tr>
<td>P1 X</td>
<td></td>
<td>( \forall a, b \in X: \neg O \text{ab} )</td>
<td>the areas in X are disjoint</td>
<td></td>
</tr>
<tr>
<td>P2 XY</td>
<td></td>
<td>( \forall b \in Y: C \text{Xb} )</td>
<td>the areas in X cover as a figure all areas in Y</td>
<td></td>
</tr>
<tr>
<td>P3 XY</td>
<td></td>
<td>( \forall a \in X \exists b \in Y: O \text{ab} )</td>
<td>all areas in X overlap with one area in Y at least</td>
<td></td>
</tr>
<tr>
<td>P4 XYZ</td>
<td></td>
<td>( \forall a \in X \left( \exists k \in Y: P3 \text{AK} \rightarrow \exists c \in Z: P2 \text{cA} \right) )</td>
<td>the areas in X that overlap with a complex in Y are covered by an area in Z</td>
<td></td>
</tr>
<tr>
<td>P5 XY</td>
<td></td>
<td>( { \forall k \in Y \left( \exists a, e \in X: C \text{Kakb} \rightarrow a-b \right) } )</td>
<td>in every complex in Y there is at most one area of X, and each area of X is covered by a complex in Y</td>
<td></td>
</tr>
<tr>
<td>P6 XY</td>
<td></td>
<td>( { \forall b \in Y \left( \exists a, e \in X: C \text{bka} \rightarrow a-e \right) } )</td>
<td>in every area in Y there is only one area of X, and each area of X is covered by an area in Y</td>
<td></td>
</tr>
</tbody>
</table>

called *figures*, and for example, a set of areas that are connected will be called *complexes*. But especially figures consisting of objects of a certain type are nothing else than the accomplishment of a certain task, and we have *predicates on the set of tasks* \( T \). We will denote the predicates by \( P_1, \ldots, P_m \) and their structure on the set of tasks by \( P^T_i \) for \( i = 1, \ldots, m \), respectively. Thus, if \( P_i \) is an \( n_i \)-ary predicate symbol \( P^T_i \subset T^{n_i} \) holds, and \( P^T_i \) represents the \( n_i \)-tupels over \( T \) for which the relation represented by \( P_i \) holds.

*Projects* are collections of reducts and refer to a particular state in the work-flow of designing a building. *Situations* are subsets of projects, i.e., a situation may contain only a part of the objects designed in a project. Situations will be denoted by \( \sigma, \sigma, \ldots \). Thus, the set of objects of A4 realizing a task \( t \in T \), i.e., objects of type \( t \), in a situation \( \sigma \) can be denoted by \( t^\sigma \). Up to now, we have introduced a set of general concepts, which allow for expressing conditions to be met in testing. These concepts provide a kind of conceptual grid that is used in manual knowledge acquisition and in machine learning. The conditions we use are criteria for successful achievements of subtasks of an overall task in design, e.g., designing the supply air system of a building. Conditions pertain both to the position of single areas or figures or to relationships between areas and figures. If the accomplishment of a task meets a particular condition it is considered to be a positive example of that
condition. Note, that this approach is not atomistic in nature since conditions to be met may very well refer to quite a number of other subtasks. Techniques of manual knowledge acquisition have been employed to elicit the conditions to be met when attacking design tasks [Strube et al., 1994 in press]. However, the scope of this knowledge acquisition has been small, i.e., only conditions pertaining to a few design tasks could be specified in this way. Below we present a number of conditions, which have been specified by using manual knowledge acquisition techniques. More specifically, we carried out task-analyses with human experts to make the knowledge, i.e., the requirements to be obeyed, explicit and available Strube4:94. In so doing, we examined (i) how experts in the field divide the overall task of building design into packages of works; we (ii) investigated and tested the criteria experts use to check those tasks on an empirical basis. This knowledge is used by CheckUp to test accomplishments of design tasks.

- All objects referring to areas of supply air are within the area of knb2 objects. This means that the collection of knb2 objects covers all areas of supply air.
- Both the A4-objects of type zab6 and the A4-objects of type zeb4 are disjoint meaning that two A4-objects are not overlapping.
- All objects of type zvb4 are placed within the area of a snb4 object, and all zvb4 objects are disjoint.
- All zab6 objects are within a zab4 object. If they are complete they establish a disjunctive coverage.
- The same is true for the objects of type zeb4 with respect to the objects of type knb2.

In addition, we make use of the fact that the zab6 objects may form complexes that have to obey a set of conditions specified such as:

- For each zvb4 object there is exactly one complex of zvb6 objects.
- A necessity for the completeness and correctness of zvb6 is that each zab6 object overlaps with exactly one complex of zvb6 objects.
- The zab6 objects that overlap with a complex of zvb6 objects are covered by exactly one zeb4 object.

Figure 2.4: Specific Knowledge Used by CheckUp

If for example we denote the unary predicate "the areas of type _ are disjoint" by $P_D$ we have $zab6 \in P_D^T$, $zvb4 \in P_D^T$ and $zeb4 \in P_D^T$. If we denote the binary predicate "the collection of areas of type _ covers the areas of type _" by $P_C$ we have $(knb2, z*** ) \in P_C^T$ for all $z*** \in T$ (with * being an arbitrary entry) and $(snb4, zvb4) \in P_C^T$, $(zeb4, zab6) \in P_C^T$, and so forth.

2.3 Approach

Our approach for assessing design tasks has two steps. First, we address assessment in a very specific sense. That means, we have developed a rationale to scrutinize whether or not an artifact accords with a number of requirements that have to hold in order to consider it as correct. We concentrate on requirements that refer to the state of affairs when 2
dimensions are concerned. However, the approach can also be applied to requirements that refer to 3 dimensions, e.g., checking potential collisions of physical objects. Second, we have conceptualized a machine learning component that extracts the requirements needed for assessment. Both steps have been worked out, and there are now two systems with CheckUp carrying out the checking of requirements, viz., assessment and CheckL performing the learning of requirements.

**CheckUp.** To assess design tasks CheckUp uses a two-layered framework sketched above. CheckUp has been developed in order to support testing in a bigger knowledge-based system as part of the research project FABEL. CheckUp provides some kind of set of filters, and each accomplishment of a task has to pass certain of them. On the formal level, each filter is a fact, i.e., an instance of a predicate like $P_i t_j \ldots t_{j_n}$ for the $n_i$-ary predicate $P_i$ and the tasks $t_j, \ldots, t_{j_n} \in T$.

**Check-L.** The knowledge employed by CheckUp provides specific guidelines for the knowledge to be learned by CheckL. First, the general knowledge representation scheme of CheckUp and Check-L is based upon predicates regarding placement. Second, the task-specific knowledge learned by Check-L and applied by CheckUp is the structure of predicates on the set of tasks representing the requirements on the placement of their realizations. The input to Check-L are examples of correct realizations of tasks. Thus, the input may be described as consisting of a number of positive examples, i.e., combinations of correct realizations of tasks, where the predicates hold. The learning task of Check-L is to find out whether or not a set of predicates hold within a subset of two or more realizations of tasks. Note that the number of tasks considered is depending on the arity of the predicates. Once learning is based on an unconstrained testing of realizations of all tasks there are also irrelevant examples that accrue from permutation of all tasks. Check-L may take two alternative tracks to learn the structure of predicates on the set of tasks:

![Diagram](image.png)

**Figure 2.5: Structure of Check-L**

*I a: Unconstrained Testing of Realizations of Tasks.* The simplest approach to learn the structure of predicates on the set of tasks is to test all permutations of realizations of tasks that occur in a situation whether or not any of the predicates hold. In that way, a statistics
for the possible facts is generated, which states (i) the number of situations where an instance of predicate holds and (ii) the number of situations with a particular combination of tasks. For the next step we denote the set of situations, in which certain tasks $t_i, \ldots, t_k$ are simultaneously realized by $S(t_i, \ldots, t_k)$. Let $P$ be binary and $t_i, t_j \in T$. We can define the relative number of situations, in which $P_{t_i \cap t_j}$ holds, by the following fraction:

$$
\mu(P_{t_i \cap t_j}) := \frac{|\{\sigma \in S(t_i, t_j) | P_{t_i \cap t_j} \}|}{|S(t_i, t_j)|}
$$

I b: Constrained Testing of Realizations of Tasks. In an unconstrained testing all permutations of realizations of tasks are tested. Thus, with an increasing number of tasks the number of facts to be tested grows polynomially. There is, however, an alternative to testing all permutations of tasks in order to learn the structure of predicates on the set of tasks. Testing whether or not any of the predicates hold may be limited to realizations of those tasks that depend on each other. This may be achieved by using a task-structure of the domain that points out the interdependencies between tasks. Thus, we may cut down the number of facts to be tested to the number of tasks a particular task depends on.

II. Building the Structure of Predicates. $\mu(P_{t_i \cap t_j}) = 1$ indicates that $P_{t_i \cap t_j}$ holds in all situations in which $t_i$ and $t_j$ are comparable. Thus, if we define

$$(t_i, t_j) \in P^T : \iff \mu(P_{t_i \cap t_j}) = 1 \ (\ast)$$

$P^T$ will contain all pairs of tasks that need to fulfill $P$ according to all known situations, i.e., cases. This setting is some kind of hard, as it requires that the set of situations CheckUp uses as input contains no situations in which realizations of tasks are wrongly placed. To weaken this approach and allow for erroneous material, one could substitute $(\ast)$ by $(\ast\ast)$ with $\varepsilon$ representing the deviation allowed:

$$(t_i, t_j) \in P^T : \iff \mu(P_{t_i \cap t_j}) > (1 - \varepsilon) \ (\ast\ast)$$

III. Reducing Processing of Facts. The intermediate result of the steps described so far is a number of instances of predicates. On this basis a fact statistics is computed that states how often a particular predicate holds. (cf. Fig. 2.5). The number of facts to be further processed can be reduced, too. If a fact is a simple deduction from another one it needs not to be tested. For example, if a set $t_i$ of areas covers another set $t_j$ of areas, it is clear that each area in $t_j$ overlaps with one area in $t_i$ at least. In other words, we reduce the number of facts considered by using topological theory.

2.4 Example and Implementation

CheckUp and Check-L are implemented in Allegro Common Lisp. The implementation follows the principle of modularity: With respect to the implementation CheckUp and Check-L are independent. Moreover, the knowledge base CheckUp uses for testing and Check-L employs for learning (cf. 2.3 is also modular and can be exchanged by a different one.
In the sequel, we will present exemplary applications of CheckUp. To convey a precise picture of CheckUp the examples are presented on the level of the simple interface currently used (cf. Fig. 2.6). Display of testing judgements with respect to tasks tackled by the user is shown in Fig. 2.7:

\[
\text{CHECK-TASK } \text{zeb4 project } \text{O-2-1} \Rightarrow ((\text{ZEB4 INCOMPLETE-DESIGN}) \text{CHECK-TASK } \text{zeb4 project } \text{O-2-2} \Rightarrow ((\text{ZEB4 OK}) \text{CHECK-TASK } \text{zvb4 project } \text{O-4-1} \Rightarrow ((\text{ZVB4 INCOMPLETE-DESIGN}) \text{CHECK-TASK } \text{zvb4 project } \text{O-4-2} \Rightarrow ((\text{ZVB4 ERRONEOUS-PLACEMENT}) \text{CHECK-TASK } \text{zvb4 project } \text{O-4-3} \Rightarrow ((\text{ZVB4 OK}) \text{CHECK-TASK } \text{zvb6 project } \text{O-4-4} \Rightarrow ((\text{ZVB6 OK})
\]

Figure 2.6: Exemplary output of CheckUp when testing completeness and placement of design tasks

Figure 2.7: Sequence of Projects tested by CheckUp

This figure shows only the incremental expansions of each project when compared with the project before. So for example project/O-3 consists of the former knb2 as in project/O-1-1, the zab6 as in project/O-1-3, the zeb4 as in project/O-2-2 and the new new snb4, as shown in the graphic for project/O-3.
Chapter 3

Assessment Supported by a Domain Ontology (DOM)

Brigitte Bartsch-Spör, Shirin Bakhtari and Wolfgang Oertel

3.1 The Idea

The name of the system described in this chapter – DOM – stands for Domain Ontology Modelling. The essential motivation behind the development and naming of the DOM system was the insight that for building context and situation sensitive design support functions we would need a rather sophisticated and elaborated domain ontology. The amount of knowledge necessary for such an ontology can be roughly estimated by an examination of the knowledge needs. The necessity to build such a system on a sound basis is supported by the fact that we cannot confront our users with too much risky or unreliable suggestions without running into acceptance problems.

Furthermore we found out that the adaptation functions have much in common with the assessment functions and that construction functions also rely to a large extent on the same knowledge [Bakhtari and Oertel, 1995]. Thus we can say that the main idea behind this sharable and shared domain ontology is its use for multiple purposes like

- assessing the quality of design fragments ranging from single cases to whole buildings,
- proposing possible adaptations of old cases in new contexts and
- suggesting constructive steps to further refinement, and completion based on a set of design operators whose applicability is situation specific.

We emphasise the attribute sharable because we offer the knowledge contained in the ontology to other FABEL modules which can make use of it. The attribute shared is also emphasized because all existing and planned functions of the DOM system are built upon and thus share the same underlying knowledge base. Moreover, the basic concepts
and relations contained in the ontology can serve as a basis for communication within groups of designers coming from different disciplines and working together on the design of one complex building. The main advantage of such a shared ontology is that it is a constructive approach for minimising the problems that usually come along with large amounts of complicated knowledge structures in different representations not compatible with each other.

Another important point is that with the DOM system we want to make a contribution to one of the main methodological goals of the FABEL project which is the seamless integration of case-based and model-based problem solving approaches [Bartsch-Spörl and Bakhtari, 1995]. And last not least, we have developed DOM’s software architecture in a way that enables the system to work on top of different drawing or CAD tools.

### 3.1.1 Underlying Assumptions

The DOM approach is built upon the following application-oriented assumptions [Bakhtari and Bartsch-Spörl, 1994]:

- There is a need for active assistance functions like assessment, adaptation and completion-oriented construction aids.
- All these assistance functions can and should be carried out both on small and on larger portions of the artefact to be designed.
- These functions can and should be activated both explicitly by the user at any point in time (s) he wishes to do this and automatically by the system whenever a piece of design is declared to have reached its final stage.
- The computational effort necessary for these assistance functions can be carried out within a time the users are willing to wait for the results.

Moreover, there are two other important assumptions which result from the way DOM interacts with its environment:

- There is an external layout platform used by the architects and engineers for developing their designs.
- There is an import/export interface for exchanging design fragments between the external layout platform and the DOM system. This interface is able to translate the representation format used by the drawing tool(s) to the DOM modelling language and vice versa. The translation will not be free from losses but it has to preserve the objects, features and structures which are essential for carrying out the assistance functions described above.

From these assumptions it becomes evident that the DOM approach is a rather generic one and has no chance of being fully accomplished in a short time. Thus we are forced to concentrate particularly our implementation work on a meaningful section of the design domain for each new DOM prototype system. We decided to focus on the design of heating, ventilation and air conditioning (HVAC) systems, on ARMILLA [Haller, 1985] as a
methodology for the spatial organisation of HVAC systems and on DANCER [Hovestadt, 1993a] as a first example for an external layout platform.

The figure 3.1 showing the DOM system\textsuperscript{1} environment is given in order to clarify the relations between DOM and other tools within the FABEL framework. This figure illustrates that DOM is one of several tools which interact on the one hand with the A4 data world [Hovestadt, 1993c] and on the other hand with the architects and engineers through a user interface which is common to all FABEL modules. Thus the DOM system can be viewed as a FABEL module which can be integrated into the FABEL framework through using the common interfaces to the A4 data world and to the end user.

### 3.2 Knowledge and its Representation

The knowledge used by the DOM system is the knowledge belonging to its domain ontology. The ontology contains – at least so far – the following kinds of knowledge:

- knowledge about how to classify design objects,
- knowledge about allowed design objects and their allowable spatial relations,
- knowledge about necessary/permissible aggregations of allowed design objects,
- configuration maxims for the spatial ordering of outlets,
- organisational maxims for the different service space divisions,
- coordination maxims for the different subsystems,

\textsuperscript{1}In German, DOM means cathedral. This may be a helpful information for the understanding of the figures contained in this chapter.
• knowledge about allowed and forbidden collisions between the different design objects, and

• knowledge about the necessary coherence of supply air systems.

For example, the knowledge how to classify design objects relies on taxonomic knowledge and uses decision rules which allow to distinguish between the different kinds of design objects. The knowledge about allowed design objects and their allowable spatial relations concentrates on specific subsystems like return air and contains regulations about where to place the objects relative to the underlying grid and relative to one another. The knowledge about aggregations allows to compose higher order objects like e.g. pipelines from an outlet to a trunk inlet.

The maxims for the spatial ordering of outlets define which formations of outlets are desirable and which sorts of variations are still allowed. The organisational maxims for the service space divisions indicate which layers of the service space are to be used for which sorts of ducts lying in which directions. The coordination maxims give regulations and priorities for the spatial organisation of different subsystems for e.g. fresh and return air, different kinds of water supply etc.

The last two points are closely related to two implemented assessment functions. The knowledge about allowed and forbidden collisions is used for the detection of forbidden collisions and the knowledge about the necessary coherence is used to assess the coherence of e.g. supply air systems. The list given above can and will be extended with each new group of support functions we will add to the DOM system.

If we focus on the representational aspects, the DOM ontology can be viewed as a knowledge-based system using the following entities:

• concepts,

• cases,

• schemes, and

• associations.

The DOM concepts represent the essential ARMILLA terminology together with additional knowledge useful for assessment, adaptation and construction purposes. A simple example for a DOM concept is an outlet for supply air. Cases in DOM are sets of concrete objects which belong together and which carry additional attributes for the sake of guiding their reusability. A simple example for such a case is a group of four supply air outlets coming from a building which was planned with a grid size of 12 units and which is arranged in the typical ARMILLA pattern.

Schemes in DOM describe higher order ARMILLA patterns which can be seen as connecting elements between concepts and cases. They also can be combined with additional knowledge beneficial for their reuse. An example for a scheme can be abstracted from the case mentioned before if we keep the four outlets and the essential topology of their arrangement and replace their fixed distance from the center of the arrangement by an adaptable one which depends on the underlying grid size.
The DOM associations represent different sorts of taxonomic relations. As we have neither complete nor strict hierarchies we avoided to call it a taxonomy. An example for such an association is that all supply air subsystem concepts belong to the class of supply concepts and inherit certain attributes from there. A much more detailed description of the DOM ontology is contained in the report [Bakhtari et al., 1995].

3.3 Approach

The DOM system concept shown in figure 3.2 is useful to illustrate the DOM approach from both a global and a conceptual point of view.

It shows the following parts of the system:

- The interface gate contains translation procedures which e.g. can take a design fragment given in an external drawing or CAD format and transform it into a DOM representation which is then put on the design board shown on the ground floor of the cathedral. Other translation procedures being able to work in the opposite direction take a DOM representation from the design board and transform it to an external drawing or CAD format.

- The design board can be regarded as the DOM system's project specific working memory. For every new building, an initialisation process has to take place for the determination of the overall building geometry, the internal division and geometry of
the service cores, the planning grid, etc. As soon as this initialisation is done design fragments from the external drawing board can enter the DOM system in order to be assessed, adapted or specified in more detail and sent back to the external drawing board.

- The domain ontology on the middle floor of the cathedral contains the knowledge described in the last section, orderly classified into concepts, cases, schemes, and associations. This knowledge is used by all the assistance functions in order to e.g. assess, adapt or work out design fragments lying on the design board.

- The upper floor houses the DOM assistance functions which are currently offered to the user under the headlines retrieval, analysis, classification, adaptation, generation and organization. These functions have full access to the domain ontology and to the design board on the floor below.

A more software technical view at the DOM system is contained in the figure 3.3 of the DOM system architecture.

We explain the DOM system architecture figure 3.3 from the bottom to the top and start with the transfer base which is equivalent to the interface gate where the import/export of design fragments is carried out. The shorttime data base is what we formerly called the design board and the knowledge base comprises the domain ontology. The behaviour base contains the growing set of assessment, adaptation and construction functions. The user interface and management system shown on the top enable the DOM knowledge engineer to work with the DOM system in a mode that grants full access to all available functions. This KE user interface is significantly different from DOM’s end user interface.
3.3.1 Current Status of the Implementation

The first prototype of the DOM system is implemented in Allegro Common Lisp on the basis of the FABEL development system FAENSY [Oertel, 1994]. The system – in its current state of implementation – runs on SUN workstations under UNIX. It contains the following assessment functions:

- Validity and plausibility of the involved concepts, e.g. the examination of the geometric-topological features of supply air outlets and their position.

- Coherence assessment, e.g. there must be a closed loop that connects all outlets with the trunk in order to get supported by fresh air.

- Spatial ordering and organisation maxims for topological relations in different service space divisions within the installation service space, e.g. all connecting ducts (pipelines) ought to be placed in the structured area of installation services in the ceiling.

- Coordination maxims for subsystems within the installation service space, e.g. no collision between the different subsystems – supply air, return air, etc. – in the service area is allowed.

The adaptation and construction functions are still under development.

3.4 Two Examples

Finally we will illustrate the DOM assessment functionality by a positive and a negative example.

Imagine that an engineer has worked out the layout for the return air system which is shown in figure 3.4 and placed it on a design board which has already been initialised with a grid and where all the service spaces for the air supply systems etc. have been determined. The
DOM assessment functions now get active and first classify all design elements contained in the layout. Then the spatial relations and possible aggregations between the classified elements are assessed and found to be allowable. Next the configuration maxims for the spatial ordering of outlets are examined and found to be obeyed to. In the following step, it is examined whether the geometry of the given layout fits into the service space planned for return air during the initialisation of the design board.

In case there are other subsystems already planned then a collision check is made and finally the coherence of the given return air layout is assessed. Our example passes all these checks successfully and in the end the engineer is told that this layout may remain as it is – at least for the moment.

The second example, shown in figure 3.5, looks at first sight rather similar to the first example. A closer look reveals that there is an additional outlet above the vertical branch in the middle and another additional outlet connected to the vertical branch on the right side by an additional twig duct. Moreover the capacity of the horizontal branch is much smaller than it was in the first example. The result of the DOM assessment functions for this example is as follows:

- The stand-alone outlet violates the coherence constraints. This is a fault that has to be corrected.
- The other additional outlet is not conform to the patterns which guide the configuration maxims for the spatial ordering of outlets. This is regarded as a suboptimal solution which leads to a warning.
- The radius of the horizontal branch is not sufficient. This is also a fault that has to be corrected.

So the assessment result consists of a red blinking stand-alone outlet, a red blinking horizontal duct and a slight change of colour for the other additional connected outlet. Now it is the user's term to correct the faults and to decide whether to improve the suboptimal solution indicated.
Part III

Adaptation Modules
Chapter 4

General Geometric and Topologic Retrieval and Adaptation (TOPO)

Carl-Helmut Coulon

4.1 The Idea

The main idea of TOPO is to transfer the knowledge about useful topologies from former layouts to an actual query in order to correct, extend or detail the query. The transfer uses no domain specific knowledge besides the information stored in the layouts. Therefore it’s possible to support layout in any part of a domain, if there exist former layouts. Fig. 4.1 shows one of the former layouts used to support the layout of air-condition, water and lights.

The first part of this chapter will describe a working scenario (section 4.2) followed by the description of the used knowledge representation and the approach itself. Both parts use an example from the domain of office-layout in order to explain the main concepts. Section 4.5 gives an example of the results of current implementations using the layout of Fig. 4.1 to support the layout of fresh-air lines.

4.2 Example

The designer marks a situation using his CAD-tool by selecting involved objects (Fig. 4.2 (1)). Some objects might be unplaced. On demand TOPO interprets the situation as a query and analyzes the topological and geometrical relations between the objects considering their attributes. The collection of former designs had been analyzed the same way and is now compared with the problem. The retrieval result (Fig. 4.2 (2)) is a part of a former
design including the maximum number of objects which match the objects of the problem by their relations.

Figure 4.1: One of a set of former layouts: Subsystems of two floors.

Figure 4.2: A complete run of T38.

T38 displays the surrounding of the result in order to detail or complete the query. The user selects part of it and the system transfers the objects and relations to the query ("smart paste"). Adaptation (3) cannot transfer all relations, because the table and two chairs cannot be placed between the door and the right wall like in the former design. Additionally the collision of a door with a table and a chair points on a conflict at the lower right corner, because it’s unusual for office designs.

Conflicts which are confirmed by the user are automatically interpreted as new queries. T
A searches for the smallest part of a former design including the same objects, but with correct relations (Fig. 4.2 (4)) in order to resolve the conflict. The last step transfers the topology of this part to the query.

### 4.3 Knowledge and its Representation

Layouts are the only necessary knowledge source of Top. Top extracts two informations from them, a topological representation and a statistics about the frequency of topology relations.

The topological representation consists of binary relations of various types. The type of a relation is determined by the type of the involved objects and their 3-dimensional relation (Fig. 4.3). The six shown and their opposites. The symbolical representation is like the common representation used in the field of pattern recognition [Lee and Hsu, 1992]. In order to use them to reconstruct the position of objects some of them are extended by the parameter $\Delta d$.

A 3-dimensional example is shown in Fig. 4.4. Relations of connected objects are visualized by circles. Additionally all relations are visualized by a list. The example shows the relation between a window and a table. The window includes the table in x-dimension, touches it from above in y-dimension and overlaps it from above in z-dimension.

The definition of relations shows the only restriction to layouts required by Top: The layouts must consist of typed objects. An example of a relation type is "window $\parallel_b$ table". It's the type of a relation like shown in Fig. 4.4. (In the actual test domain Top has to deal with several hundreds of object types.)

The statistics describes the frequency of occurrences of relation types for each type of object. For example, 95 % of air-condition supply zones have a relation to air-condition connection zones, which overlap them in x- and y-dimension and touch them in z-dimension. A similar approach has been introduced by [Janetzko et al., 1994b].

![Diagram](attachment:figure43.png)

**Figure 4.3:** Two objects can be related in 12 different ways for each dimension.

Topology consists of more than binary relations. In order to retrieve similar situations the context of relations must be considered. Fig. 4.5 gives an example. Situation 1 and
situation 2 both include all the relations of the query. Nevertheless case 2 is more similar to the query, because the relations occur in the same context as in the query. By using the context it is also possible to determine which B<\(A\land d\) of case 2 correspond to the B<\(A\land d\) of the query.

Figure 4.5: Considering the context, situation 2 is more similar to the query.

\(\text{T}\&\text{B}\) represents the context in a graph like [Bartsch-Spörl and Tammer, 1994]. Building a graph out of objects and relations, one must decide which ones should be the nodes and and which ones the edges. Depending on this decision different parts of two similar situations become identical (Fig. 4.6).

The first line of figure 4.6 shows two situations which differ in one relation only. It’s the relation between the wall \(W\) and the table \(T\). The second line shows the node-graph of the above situations. It uses the objects of both situation as the nodes of a graph and connect them by their relations. The largest common subgraph cannot include the wall \(W\) and the table \(T\) because they occur in both situations in different relations in spite of the fact that both situations include the same objects. By building an edge-graph from both situations a better result is reached. It uses the relations as nodes and connect them by shared objects. The largest common subgraph includes all but the one discriminating
relation. To uses relations as nodes and connect them by shared objects.

Figure 4.6: Comparing the node and the edge-graph of two situations leads to different results.

4.3.1 Domain-specific extension

The transfer shown so far uses no domain-specific knowledge besides the types of objects given by the layout and the straightforward definition of structure. Adding knowledge about transformations of structure which do not tackle the usability of a case leads to a higher quality of the result.

In the domain of building layout, rotation (in steps of 90°) or reflection in the horizontal dimensions of parts of a topological structure are such transformations.

Using this knowledge To is able to match the fragments shown in figure 4.7 (a) completely, although a subpart is reflected. To searches for the largest common subgraph by applying the mentioned transformation to the structure of the query as a whole. It finds the matchings shown in (b). The differences of both results match disjunctive sets of nodes and therefore the two matchings can be combined to (c).

4.4 Approach

As described in the example To supports several steps during case-based reasoning. These steps of retrieval, transfer and correction are illustrated in this section.

4.4.1 Retrieval

Given a layout of any size, To matches the topology of the layout and the query. This is done by computing a edge-graph representation of the layout and the query and searching for the largest common subgraph. In order to solve a similar problem, the problem
of finding a maximum clique of a graph, various algorithms were developed [Babel and Tinhofer, 1990]. A clique is a complete connected subpart of a graph (every node knows every other). Instead of searching for a common subgraph of two graphs one searches for a maximum clique in one graph representing all possible matchings between the two graphs, called "combination graph". A maximum clique is the largest of all cliques of a graph.

Using the transformation described in [Barrow and Burstall, 1976] the nodes of the combination graph represent all matchings of nodes of equal type of the source graphs. Figure 4.8 shows an example. As mentioned before we decided to match the topological relations instead of the objects themselves for reasons described in [Coulon, 1995]. Therefore the nodes of the combination graph shown in figure 4.8 represent all matchings between the relations of the source graph of equal type. The source graphs (f) and (g) contain objects of type a and b connected by directed relations. The type of a relation is defined by the types of the source and the target object. Two nodes are connected in the combination graph if and only if the matchings represented by the nodes do not contradict each other. The matchings \( R_2(ab) \Leftrightarrow R_3(ab) \) and \( R_5(bb) \Leftrightarrow R_{10}(bb) \) are connected because both relations occur in both source graphs in the same context. Both are connected by a shared object of type b. \( R_2(ab) \Leftrightarrow R_3(ab) \) and \( R_1(ba) \Leftrightarrow R_{06}(ba) \) are not connected because the matched relations share an object of type a in graph f but do not share any object in graph g.

The maximum clique in this combination graph and the corresponding maximum subgraphs are marked in black.

### 4.4.1.1 A common maximum clique algorithm

The algorithm of [Bron and Kerbosch, 1973] (for further use called "max-clique\_BK") finds all cliques in a graph by enumerating and extending all complete subgraphs. It extends complete subgraphs of size k to complete subgraphs of size k+1 by adding iteratively nodes which are connected to all nodes of the complete subgraph.
Figure 4.8: Transformation of the problem of the maximum common subgraph to the problem of finding a maximum clique in a graph: According to the edges of both graphs a combination graph is computed. The maximum clique and the corresponding matching is marked in black.
4.4.1.2 The improvement

The idea of the improvement is to search for matchings of connected subgraphs only and combine those matchings in a second step. This strategy requires to change the transformation and the search algorithm.

The transformation must generate two types of edges. One type is called "expressive edge". A relation of this type is generated if and only if the two matchings connected by the expressive edge match relations which are connected in both source graphs in the same way. Edges between matchings of relations which are not connected in both source graphs get just the normal type "edge". Let us consider the combination graph of the comparison between the query and the case of figure 4.9. The matchings (A1 ⇔ B2) and (B1 ⇔ C2) are connected by an expressive edge because the matched relations A1 and B1 share a chair in the same way as B2 and C2. The matchings (A1 ⇔ B2) and (D1 ⇔ D2) are connected by a "normal" edge because the matched relations do not share any object in both situations.

The matchings (A1 ⇔ B2) and (B1 ⇔ D2) are not connected by any edge because the matched relations A1 and B1 share a chair but B2 and D2 do not.

Comparing the query and the case (Fig. 4.9) max-clique BK takes e.g. two arbitrary chairs of the query and searches for a corresponding group in the case. If these chairs are not related to each other, all not related pairs of chairs of the case might correspond to them. In larger buildings there are thousands of such pairs. Therefore the comparison becomes very costly. Doing it by hand, one searches as far as possible for corresponding relations of connected parts only. Extending the algorithm by this behaviour we get max-clique BK+.

The new search algorithm (for further use called "max-clique BK+") must distinguish between both types of edges. It extends complete subgraphs of size k to complete subgraphs of size k+1 by adding iteratively nodes which are connected to all nodes of the complete subgraphs and at least one edge must be an expressive edge. Therefore all complete subgraphs found by this algorithm contain a path between any two nodes which consists of expressive edges only. For further use we call these complete subgraphs "complete expressive subgraphs". They represent matchings between connected subgraphs only. The largest common subgraph is the largest combination of connected common subgraphs consisting of disjunctive sets of nodes.

4.4.2 Transfer

The retrieved situation consists of objects which match part of the objects of the query. There are two informations which might be transferred from the situation to the query. On one hand the topology of the situation might be used to correct the query. On the other hand the surrounding of the situation might be transferred in order to detail or extend the query (Fig. 4.10).

The topology of the retrieved situation is used in order to correct the topology of the query (the chair is placed next to the table) and to place the unplanned objects of the query. The surrounding of the retrieved situation includes shelves which do not occur in the query. Therefore suggests to transfer these objects to the query in order to complete the layout.

Some of the objects, which might be transferred, have absolute positions in relation to the objects of the query. For example the shelf in the upper left corner. These objects
Figure 4.9: The new algorithm $\text{max-clique}_{BK+}$ compares connected groups only.

Figure 4.10: Topology and additional objects of the retrieved situation are transferred to the query.
are transferred first. The rest of the objects is transferred later. For example the shelves below the upper left one have only a relative position in relation to the wall, they should be placed in the middle of the wall. These shelves are transferred one by one, each one determining the position of the next one. The last step places all objects, which still have relative positions.

The result of the transfer is a new layout, which need not to be correct (Fig. 4.10; lower right corner). Therefore the next step is the correction of the layout.

### 4.4.3 Correction

The spatial relations of the result are compared with the relations occurring in the layout. Unusual relations are presented to the user as possible conflicts. Conflicts which are confirmed by the user are automatically interpreted as new queries. LiteDiag searches for the smallest part of a former layout including the same objects, but with correct relations in order to resolve the conflict.

Figure 4.11 illustrates how LiteDiag corrects the result of the transfer. The result (Fig. 4.10) consists of unusual relations in the lower right corner. The door overlaps two chairs and a table. If the user confirms these relations to be incorrect, LiteDiag initiates a new retrieval. This time it is not useful to search for a situation with the same topological relations, because these relations are incorrect. The task is to find a correct placement of those objects, which belong to the incorrect relations.

Additionally the placement must not use more space than used by the wrong placement in the query. Therefore LiteDiag searches for a correct placement of a door, two chairs and a table within radius I. Because there is no such placement LiteDiag increases the radius and searches for a larger placement consisting of all objects inside radius II of the query. A situation is found and it’s topology and additional objects are transferred to the query.

![Figure 4.11: LiteDiag corrects the result of the transfer.](image)
4.5 Implementation

This first example is completely artificial, but makes it easy to understand the perspective of TL8. After implementation of the retrieval step it’s now possible to work on realistic examples: Fig. 4.12 shows a sample run of TL8 in order to support the layout of twig lines for fresh air supply using the layout of Fig. 4.1.

Figure 4.12: A sample run of TL8.

Figure 4.12 shows a query (a) consisting of a trunk line connected to a hook of branch lines and an incorrectly placed hook of twig lines to be compared with the layout of part (b), (b) shows the nearly complete technical layout of two floors. Their topological representations consist of 8 (a) and 395000 (b) instances of relations. As a result of the comparison there are two fragments ((c) and (d)) of the layout (b) which match most of the topology of the query. The only difference is the placement of the hook of twig lines. The disconnected twig lines of (d) are connected to a branch line in the surrounding of (d) (compare part (f)). Both retrieved parts are presented to the user.

If the user chooses part (c) to be transferred, part (e) shows a transfer of the relations of (c) to (a) which corrects the incorrect placement of the twig lines in (a). Part (f) shows the result of a transfer of additional supply-air objects from the surrounding of (c). A transfer of all objects of the surrounding of (c) shown in (g) might be feasible and is supported by TL8, but must be coordinated with the surroundings of the query.
4.5.1 Handling exponential complexity for interactive use

Because ToKo wants to support interactive work and uses an algorithm which is practically very efficient, but has still an exponential complexity, it provides in addition several features in order for controlling the runtime:

**Anytime behaviour:** The algorithm can be interrupted by the user at any time and the quality of the result is always better than or equal to the best earlier result [Russell and Zilberstein, 1991].

**Transparency:** In order to help the user to decide if and at which moment he might interrupt the process, ToKo visualizes three indicators during runtime:

- **The progress indicator** shows how much of the search space is already searched.
- **The quantity indicator** shows how many cliques have been found.
- **The quality indicator** compares the size of the largest clique found so far to the size of the largest clique possible. The size of the largest possible clique depends on the minimum of the number of relations of the case and the query for each type of relation occurring in the query:

\[
\sum_{t \in \text{Types}-\text{of-}\text{-query}\text{-min}} \left( \#\text{relations}_{\text{type}-t}(\text{query}), \#\text{relations}_{\text{type}-t}(\text{case}) \right)
\]

**Reflective knowledge:** The algorithm estimates its own runtime till completion before and during running [Coulon et al., 1993]. If the runtime is higher than a threshold it asks the user for a strategy in order to reduce the problem.

**Reduction strategy:** If the estimated runtime is too high, ToKo suggests to sort the relations of the query by their importance given by the user. By this preferences the amount of complete subgraphs is reduced further. ToKo generates only those complete subgraphs of size k which include matchings for the first k relations of the ordered set of relations of the query. This feature enables the user to easily restrict the search space depending on his own preferences.

4.5.2 State of the work on the algorithm and further improvements

The described algorithm max-clique$_{BK+}$ is implemented and tested in the domain of technical layout and molecular chemistry providing all features mentioned in section 4.5.1 besides the reflective knowledge. This knowledge will be acquired and added as described in [Coulon et al., 1993].

For some of the comparisons the measured runtime was exponentially higher than expected. In these cases the query and the case had chains consisting of identical links. For example a common row of outlets, rooms or larger links like \{supply-air-branch-line, supply-air-branch-line, used-air-branch-line\}$^n$. Comparing two such chains, all possible subparts are matched. To avoid this problem, chains could be substituted by abstract elements (chains) with the type of the contained relations and their size. Instead of comparing all possible subparts a future version of our algorithm will match those abstract
elements. The used knowledge will be similar to part of the knowledge used in [Börner, 1994a]. The difference is that in our approach the content of such abstract chains is not restricted.
Chapter 5

Adaptation by Active Autonomous Objects (AAAO)

Parivash Adami

5.1 The Idea

In Architecture, many constraints and rules deal with a restricted neighborhood of building elements only. There is a lack of global building models and global design strategies and systems developers have to be content with that. Developers have thought of different answers to this situation, CBR being a prominent one. Another answer could be models where knowledge is only locally formulated and the global development of a design is the result of parallel execution of simple models. In our approach we use the metaphor of active autonomous objects, hence the module’s name AAAO. Our model is drawn up in detail for an exemplary task, that is the placement of columns, but can be extended to other tasks as well.

In our domain architects use the steel frame construction set MIDI for the construction of great, complex buildings like schools, laboratories or modern office buildings. During knowledge acquisition we found a set of constraints and criteria for the placement of columns and beams, which consider especially statical requirements and then architectural demands with respect to the layout of rooms and the kind of their use. These are refined to very limited surrounding of a column or a zone of use. This led in a very natural way to a model of active autonomous objects, columns being the objects, that behave according to simple heuristics trying to perform a set of applicable constraints.

The model has already been fully described in FABEL by [Morgenstern, 1993]. We are currently implementing it and experimenting with different sets of constraints and heuristics. So far we are rather optimistic that this type of model is very appropriate for a great number of design problems in architecture and that it is more efficient both to develop and to perform than global search algorithms.
5.2 Knowledge and its Representation

AAA0 uses knowledge which is formalized in constraints. These constraints depend on the statical requirements of MIDI and some constraints defined by the use of rooms. Some of the constraints are hard, which means that they have to be performed by correct solution. Weak constraints should be performed but might be violated. The constraints are represented procedurally and are applicable to a column and its surrounding.

5.3 Approach

![Flowchart](image)

Figure 5.1: The problem solving process in AAAO

The input of AAAO are the layout of the building and rooms (use zones in A4) of the problem case, and the placement of columns of the case. The case may be retrieved before by any of the retrieval modules, than input case is missing, we a standard placement of columns instead which fills the layout with regularly placed columns.

The process of problem solving (figure 5.1) consists of several cycles. During each cycle each column reads its local surrounding, consisting of rooms and nearby columns from a building plan. It applies the constraints in order to evaluate its position and directly neighboured positions. If there is a better position than the actual one it moves to it. If hard constraints are violated and it was not possible to perform the constraint by autonomous moving, the column communicate with its neighbourd columns in order to perform the
constraint by a coordinated action. Possible actions are coordinated moving, destroying a column or creating one.

When no improvement was accomplished during a number of cycles the so far best solution with the most performed constraints is presented. If there are still hard constraints violated AAAO was not able to solve the adaptation task. In any case the actual evaluation of the columns describes the quality of the result.

5.4 Example

In our example (figure 5.2) an architect is designing an office building with the steel frame construction set MIDI. The figure shows one floor with offices and seminar rooms. The dark areas within the bounding box are "outside" the building.

Figure a) shows the systems input, that is the outline of the building and the intended zones of use, together with an initial placement of columns, which does not satisfy neither statical nor architectural needs.

Figure b) shows the situation after columns outside the building have been removed and the remaining have evaluated their position. Bold circles indicate problem positions, where hard constraints are violated.

Figure c) shows the evaluation of all neighbour positions of the problem positions. A column will move to that neighbour position with lowest score.

Figure d) shows the new situation after all problem columns have moved one step. The new scores are indicated and we see that there are still violated constraints. The next step would be performed as exemplified in figure c).

5.5 Implementation

The AAAO model is implemented in Common Lisp [Steele, 1984] Steele:84) and its object oriented extension CLOS (Common Lisp Object System, [Keene, 1989]). Keene:89).

The AAAO model is implemented in full detail for the adaptation of column positions on a floor to fit a certain use. The object oriented implementation, however, was designed to be extendable to other tasks in building design as well.

All essential parts of the model are implemented as object classes. That are

- The building objects to be generated, or adapted or considered (columns in our example).
- Then a manager for each task (columns-manager in our example) who knows about initialisation, applicable constraints or interaction of the aa-objects.
- The constraints. For each task there is a set of constraints that serves as behavioral knowledge of the aa-objects. These constraints are also implemented as an object class. This way the constraints can be used in different tasks and in various combinations.
Figure 5.2: Example of our object models behavior.
Initial state, first cycle of evaluation and reaction, second evaluation
• a map of the area within the building, that shows the position of the aa-objects and building objects (a map of the floor with the possible positions of columns, their actual position and the position of the zones of use to be considered).

The parallel action of the autonomous objects is simulated only. Essential means of simulating this parallel activity and interaction are time cycles where each aa-object executes its actions, and the action depending on the results of the last cycle, only, never on results meanwhile achieved in the current cycle.
Chapter 6

Case Adaptation using Agents (AgentEx)

Raghu R Bhat

6.1 The Idea

This module assumes an interactive human-agent problem solving scenario where the human observer activates various agent systems, continuously accepting, modifying or rejecting the suggestions put forward by the agents. The emphasis is on a fast generation of nearly correct solutions, presented visually for the users evaluation, rather than their exhaustive enumeration and evaluation.

This module provides a semi-automated adaptation of a current design problem by adapting the input from a retrieved case. A case is a layout of design objects of various types which solve a similar problem. Information from the case is transferred into the current context and the objects are instantiated into agents. These agents then align themselves in the new context according to their domain knowledge and transferred case information.

A knowledge base provides possible functional relations between design components, while a case provides the actual components and the spatial relations between these components. Since functional relations can have many spatial realisations, a case provides an actual example of desired spatial relation between components.

A particular domain, that of conflict free layout of supply air pipes is chosen as the problem domain using the knowledge provided by the ARMILLA system [Haller, 1985]. The knowledge base consists of the elements of this system and functional and spatial relations between the components. Agents are software objects encoded with domain knowledge, capable of reactive behaviour [Bhat, 1995], and possess properties of observation, default actions and a simple negotiation mechanism. The representation scheme is the A4 [Hovestadt, 1993b] scheme, which assigns every object a location in a multidimensional design space.
6.2 Knowledge and its Representation

The operative knowledge in the system can be divided into three categories - the model of the task space and rules for solving each task, background domain knowledge and the spatial relations extracted from the retrieved case, and each is described briefly below.

6.2.1 Model of Task Space

Objects in the design space are assigned a four character label, representing a type as well as a location in a multi-dimensional space. Thus an object label such as zeb2 represents semantically the task of designing the supply air subsystem, its general spatial location in an area, with the subsystem encompassing the whole building.

These task objects, their place in a planning hierarchy and their interrelation are specified as a directed graph by the planning guide PM5 [Hovestadt, 1995]. For each of the task objects for the domain supply air, there exists a list of subgoals, heuristics for their ordering and rules for achieving each subgoal.

```
<table>
<thead>
<tr>
<th>PA</th>
<th>PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2-zul</td>
<td>zul-eb4</td>
</tr>
<tr>
<td>v1-zul</td>
<td>zul-eb6</td>
</tr>
<tr>
<td>lp-zul</td>
<td>zul-vb4  -&gt; zul-vb6</td>
</tr>
<tr>
<td>hp-zul</td>
<td>zul-vh4  -&gt; zul-vh5</td>
</tr>
<tr>
<td>ep-zul</td>
<td>zul-vt4  -&gt; zul-vt5</td>
</tr>
</tbody>
</table>
```

Figure 6.1: Planning tasks in the domain supply air

This is shown in Fig. 6.1 which shows the domain-wise break up of tasks and their interrelations according to the planning methodology for the domain supply air. As an example for the zeb4 task the predicates `create(zeb6)` and `align(zeb6)` would be defined with knowledge based information to achieve these subgoals.
A graphical view of the relation is shown in Fig 6.2. The containment relation is transitive so that the relations, A contains B and B contains C, imply A contains C. The mathematical computation of the containment relations is straightforward, and is done by direct comparison of their coordinates of the bounding boxes.
• Adjacency: This relation defines all neighbours for each object, which are directly adjacent in each of the cardinal directions. A graphical view of the relation is also shown in Fig 6.2. As before, computing adjacencies is straightforward, and is done by direct comparison of coordinates of the bounding boxes.

These relations can be extended to three dimensional relations by adding two more cardinal directions up and down.

6.3 Approach

The current task is one of the tasks defined in the planning module PM5 and is under the current focus of the user. Retrieval of the case matching the current task is done using the methods for retrieval detailed earlier [Voß, 1994]. The retrieved case matches the current task and shows an elaboration. The retrieved case is analysed and the following information retrieved.

• The object types are extracted, some of these are task objects and are mapped to the PM5 task space to find their ordering.

• Next the spatial relations of containment and adjacency are extracted, and each object tagged with this information.

• The case is then directly copied to the current problem situation, and the objects instantiated into software agents.

• These agents use domain knowledge and the extracted spatial knowledge to adapt themselves in the new situation.

Adaptation here means creation of new instances, deletion of instances according to domain knowledge and their correct spatial alignment according to the derived spatial relations. The result of this is a layout which is similar to that of the retrieved case. The user accepts this with or without modifications or rejects it. Acceptance by the user results in a change in the current situation. The process continues with the user focusing on a new problem, for which a new case is retrieved and so on.

6.4 Example

A scenario is sketched graphically in Fig 6.3 and the description below refers to this.

The design situation is where the floor and rooms are defined, which are marked F and R respectively in Fig 6.3(a) and the user wants to design a layout for the supply air. A focusing agent is defined for the task \( \text{xeb}_4 \) whose scope or area-of-interest circumscribes the entire floor, and the default control passed to this agent. This is shown in Fig 6.3(a) as a solid square and this focusing agent controls objects inside its area of interest.

The root agent then retrieves a case using the standard Fabel retrieval module, with the task \( \text{xeb}_4 \) as the query. One of these is selected by the user and is shown in Fig 6.3(b). The
case shows the floor and its regular division into supply air areas, one for each room. Case analysis leads to the following sequencing of tasks $zeb_4$, $zeb_6$ according to the PM5 task space. For each of the object instances of $zeb_4$ and $zeb_6$, the containment and adjacency relations are derived.

The root agent copies the case into the current situation after a rotation, and all the objects in its scope are instantiated into agents and this is shown in Fig 6.3(c). It may be noted that the case does not match the situation exactly, neither in position and there are more $zeb_6$ agents than necessary, as there are only five rooms and nine $zeb_6$ agents.

This leads to the next step - that of adaptation of the case to the current situation. Adaptation begins with the focusing agent passing control to the the $zeb_4$ agent, as shown in Fig 6.3(d). Next $zeb_4$ makes a task announcement for align-action, and each $zeb_6$ agents make a bid for aligning themselves with one of the rooms. Bids are based on spatial proximity as well as fulfillment of spatial constraints of each $zeb_6$ agent. Bids are evaluated by $zeb_4$ agent and control passed to successful $zeb_6$. The $zeb_6$ agents with successful bids will align themselves and pass control back to the $zeb_4$ agent. The successful agents are shown shaded grey and the position where they will move to is shown by arrows in Fig 6.3(d).

The $zeb_4$ agent then makes a delete-action announcement and the unsuccessful $zeb_6$ agents are deleted. These are shown shaded black in Fig 6.3(e), and after their elimination control passed back to $zeb_4$ agent. The $zeb_4$ agent makes an announcement for evaluation and various $zeb_6$ agents notify spatial constraint violation by highlighting as shown in Fig 6.3(f). The user can now accept, reject or modify the solution and proceed on further
elaboration of the design by choosing a new focus.
In case the user wants to proceed further a new focus is set up and cases are retrieved according to this focus. Thus in the scenario, the user would probably choose a new focus which includes some or all *zebō* objects and retrieve a corresponding case. This case would then be applied concurrently to all or only some of the *zebō* objects in the users focus by the focusing agent.

### 6.5 Implementation

The language for implementation for agents is Oz [Smolka *et al.*, 1993] a concurrent, constraint programming language which combines ideas from logic, concurrent and object-oriented programming. The prototype is under development at the moment and is expected to be ready by the end of August 95.
Part IV

Construction Modules
Chapter 7

Analogical Layout Design (SYN)

Katy Börner and Roland Faßauer

7.1 The Idea

The module SYN (for Synthesis) partially implements the approach to Conceptual Analogy (CA) that was proposed in [Börner, 1995a] based on previous work reported in [Börner et al., 1993; Börner, 1994a; Börner, 1994b]. CA is an approach that integrates conceptualization based on prior experiences, i.e., case memory organization, and analogical reasoning. It was developed to support the design of geometrical layouts, a subtask of building engineering. SYN implements some of the main processes constituting CA and demonstrates its applicability to supporting geometrical layout design. This section reports the domain characteristics the approach was built upon and argues for CA, i.e., the integration of memory organization and analogical reasoning.

7.1.1 Domain Characteristics

A typical subtask occurring in building engineering is the layout design of interconnections between supply accesses and main accesses. Different tasks (e.g., the connection of supply air, return air, or electricity accesses) require different connection patterns. For example, used air supply accesses are connected using the shortest admissible way. On the contrary, supply air connections take curved tracks to reduce the noise caused by the flowing air etc.

As an example Fig. 7.1 shows a projection of the supply air layouts for one floor of the Swiss railway education center Murten (see section 1.1, Fig. 4.1). The left hand side of the DANCER drawing shows the outlet of supply air accesses. The right hand side illustrates their proper connections.

The knowledge needed to support such layout design task is generally too complex to be represented by rules. Architects browse through prior layouts to inspire and guide their
work. This working style points to case based reasoning (CBR) as the predominant problem solving method [Kolodner, 1993; Wess et al., 1994; Aamodt and Plaza, 1994]. Additionally, architects form concrete individual design experiences into generalized concepts. Later on, these general concepts, named "design prototypes" by [Gero, 1990], are used to guide the design process.

Another characteristic of geometric layout design refers to the importance of relations between objects rather than the attribute-values of single objects. Retrieval and adaptation of prior CAD-layouts mainly proceeds via the spatial relations of their constituents. Complex case representations, which allow not only the consideration of geometric attribute values of single objects but most of all of their topological relations, cause increased computational expense in retrieving, matching, and adapting cases. Short response times, however, are crucial to the acceptance and usage of case based design systems.

To increase the efficiency of design support, only adaptable cases should be retrieved and considered for solution transfer. That is, retrieval of prior CAD-layouts should depend on the adaptation knowledge available [Paulokat et al., 1992; Smyth and Keane, 1994; Börner, 1994a].

### 7.1.2 Memory Organization and Analogue Reasoning

To handle the problems surveyed, we propose the approach of conceptual analogy. CA integrates and automates memory organization and analogical reasoning on the basis of attribute-value representations of prior CAD layouts which are electronically accessible.

As for memory organization, CA acquires cases in a task-oriented way [Janetzko et al., 1994a]. Thus, the task-structure determines the grain size of cases (CAD-layouts). Cases are represented by the geometrical attributes of their constituent objects (e.g., accesses, pipes etc.). Every case is assumed to represent implicitly some the rules inherent in a CAD-layout. For efficiency reasons the entire set of cases is organized into task-dependent case bases. Subsequently, every case base is further divided into case classes containing cases with a similar common structure. Here, the topology of cases is important. Therefore, all cases are transformed into algebraic representations, which reflect the topological

Figure 7.1: Supply air layouts of Murten
relations among objects (e.g., which pipe connects which accesses). This algebraic representation also provides the basis for inductively determining the prototypical topology which characterizes cases belonging to one case class. In such a way, the entire set of cases may be represented by a few prototypical ones.

During analogical reasoning, this kind of memory organization restricts the search space to cases supporting an identical task and showing similar topology. Given a new problem \textsc{Syn} selects the appropriate task-dependent case base first. Next, it redescibes the new problem in terms of the prototypes representing this case base. The solution of the most similar prototype will be applied to solve the new problem. Similarity of complex case representations is determined in terms of adaptability.

The approach has been shown to be integrable into a highly interactive, adaptive system architecture that allows for incremental knowledge acquisition and user support (see [Börner and Janetzk, 1995; Börner, 1995b]). It is especially suited to tasks where explicit rules about the derivation of design solutions are not available but huge amounts of CAD-data are electronically available.

\textsc{Syn} has been designed to handle the geometric layout design of supply systems for air, water, electricity, etc. The subsequent sections introduce the knowledge used and its representation, explain the part of the CA approach implemented in \textsc{Syn}, and show an example run of \textsc{Syn}.

### 7.2 Knowledge and its Representation

The knowledge representation employed by \textsc{Syn} is suited to MIDI, ARMILLA, and A4 (see chapter 1). Thus, standardized objects occurring in MIDI/ARMILLA buildings are represented in the attribute-value representation A4 and may be visualized via \textsc{Dancer}.

For memory organization and analogical reasoning, the attributes defining the geometry and the type of objects are used exclusively. The grainsize of cases corresponds to the task-structure. We employ two different case representations, namely an attribute-value representation and an case tree representation, in considering geometric and topological attributes of cases.

**Attribute-value case representation:** Here, the case problem and its solution are represented by sets of A4-objects. A4-objects constituting the problem may be of a different type. The A4-objects constituting the case solution (the completed task) share one type per definition. Fig. 7.2 provides an example case.

**Case tree representation:** The three dimensional, orthogonal planning grid provided by ARMILLA and the knowledge representation / organization scheme A4 are the basis for uniform transformations of attribute-value representations into algebraic representations of object arrangements. Assuming the existence of a transformation function $\phi$ and its inverse, every case problem together with its solution are transformed into a tree representation (see Fig. 7.4). To arrive at uniform representations we represent the main access by the root-node, the supply accesses by the leaf-nodes and the pipes by the intermediate tree nodes. Labels at intermediate nodes denote the distances between the centers of connected
objects in three dimensions. The internal case tree representation, depicted in Fig. 7.3, corresponds to a list.

Design prototypes are represented by case trees in which the common features of cases are expressed by constants and the differences by variables. In such a way, the design prototype represents knowledge generalized from a set of similar design cases and form a class from which individual ones can be inferred.

7.3 Approach

This section does not present the general approach of conceptual analogy that was proposed in [Börner, 1995a]. Instead of this, it explains the part of CA implemented in SYN. Although the approach of CA is a general one, the case-tree representation and the corresponding transformation routine, which translates attribute-value representations into case tree representations, strongly depends on the domain selected and the tasks that are tackled. Different tasks may require different structural case representations.
Syn’s input for memory organization is either a task-structure and a project (representing a design by a set of A4-objects) or a case base. In the former case, the task-structure is applied to extract cases out of projects and to organize them into task-dependent case bases. To reduce the response time for design support, memory organization is preferably done as a preprocessing step. Subsequently, the cases are translated into case tree representations as was introduced in the preceding section. This representation provides the basis for diving task-dependent case bases into case classes containing cases of similar structure.

As for analogical layout design, Syn takes a new problem and delivers its solution(s) at runtime. There are four parameters which control Syn’s search for similar cases and the new solution. Firstly, reformulation-variants, which restrict the maximum number of problem redesciptions in terms of the prototypes available. Secondly, similarity (%), which specifies the similarity of prior cases to the problem needed to consider the cases for solution transfer and adaptation. Thirdly, adaptation-variants, which fix the maximum number of different adaptation variants to be tried. Fourthly, max-solutions, which restrict the maximal number solutions to be proposed. Reasoning proceeds as follows. In a first retrieval step, the task-dependent case base appropriate to solving the new problem is selected. This is done by attribute-value matching from the type attributes of the constituting A4-objects of the problem with the problem types of the available set of case bases. Note that the solution type of the selected case base corresponds to the type of the new so-
solution objects. If no appropriate case base is available, the problem is rejected. In a second retrieval step, the actual problem (represented by a list of A4-objects) is internally redescribed in terms of prototypes which represent the selected case base. That is, the problem is transferred into its case tree representation. The value of reformulation-variants restricts the number of attempted reformulations. Fast preselection compares the number of supply accesses of prototypes to the number of accesses in the actual problem. If a prototypical case with the same number of supply air accesses as in the problem exists, all reflected and rotated versions of this case are computed and compared to the actual problem. All (max. eight) reflected and rotated case versions are sorted corresponding to their similarity to the actual problem. The transfer and adaptation of prototypical solutions to the actual problem proceeds corresponding to their similarity. The lower the similarity threshold the more "innovative" adaptations are proposed. The maximum number of trials applying solutions to the problem is fixed by adaptation-variants. All solutions created from this process are sorted in decreasing order corresponding to their structural similarity to the given problem, and are returned as a list. The number of proposed solutions is restricted by the value of max-solutions. A retransfer routine guarantees the transformation of internal data into data output format, i.e., the A4 attribute-value representation of the solution.

The search complexity strongly depends on the value of max-solutions and similarity. The worst case is max-solutions=100 and similarity=0, i.e., a search for all solutions by adapting all available cases. The resulting complexity depends on the number of accesses $n$, this being $O(n!)$. By restricting max-solution and similarity and letting $k$ be the number of nodes in a case tree, the complexity may be reduced to $O(k^2)$.

Note that the adaptation knowledge used by the system may considerably differ from the strategies of its user. For that reason, it is not recommended to use Syn as a retrieval module. Preselection of candidate cases by retrieval modules is advantageous.

### 7.4 Implementation and Example Run

**Syn** is implemented in Allegro Lisp 4.2 on Sun SPARC Station. It is front end connected to the design editor and navigator Dancer running on Next Cube via TCP/IP communication. A tk/tcl-based X11 user interface is provided by Fabel-Idea-prototype 2.1 [Schmidt-Belz et al., 1995], Whereas the overall system provides interfaces (see the Fig. 7.5 for the Fabel-Idea control panel), databases etc., Syn supplies the system with solutions for actual design problems.

In a typical usage scenario, the user first starts the memory organization process to provide the knowledge needed for design support. To illustrate this, we apply Syn to the layout design of supply air connections. For this task, we exploit the relatively simple, but easily accessible data base of the supply air layouts in the education center Murten. These are 42 in number (called '42 cases') and are graphically depicted in Fig. 7.1. Every case consists of a set of A4 objects representing its problem and another set representing the case solution. Objects the types zab8, zab6, and zvh6 constitute the problem. A set of supply air connections (zvh7-objects) represents the case solution. Fig. 7.2 depicts an example case, more exactly case no. 42. Its graphical representation in Fig. 7.4 shows, that the root node represents the main access (zvh6-object). Leaves denote supply accesses (zah8-
objects). Intermediate nodes refer to connections (ZH7-objects). The nodes are labeled by their placement and extension in three dimensions.

To start knowledge organization the SYN module needs to be selected in the Fabel-Idea control panel together with the supply-air case set containing the 42 cases from Murten. The command create case base organizes the entire case set into three case classes which are represented by three prototypical connection patterns. These patterns, depicted in Fig. 7.6, connect two, four, or six air accesses to the corresponding main access. They are represented by \texttt{42cases.42}, \texttt{42cases.36}, and \texttt{42cases.28} respectively. Memory organization is by far the slowest operation of SYN but may be executed in non-working times.

![Figure 7.5: Fabel-Idea control panel](image)

![Figure 7.6: Three general supply air connection patterns](image)
As for analogical reasoning, Syn constructs proper solutions for air access patterns, i.e., air access connections. Aiming at graphical man-machine interaction, problem formulation proceeds by object selection via mouse click in DANCER (see Fig. 7.7). A valid problem provides the supply air area, the main access, and the accesses which need to be connected. Otherwise, the module responds with an error report problem is invalid. Corresponding to the task-structure, the particular subgoal, i.e., the object type the user wants to concentrate on, is the connection of air supplies.

![DANCER](image)

**Figure 7.7: Problem formulation via DANCER**

To run Syn the appropriate case base (here supply-air) needs to be selected in the Fabel-Idea control panel. Pushing the **execute** button, brings up the Syn dialogue box (see Fig. 7.8). Four parameters may be selected corresponding to user preferences. The number of attempted problem reformulations and adaptation variants may be restricted by the values for reformulation-variants and adaptation-variants respectively. The possible range of the similarity (%) value, stating the required similarity of cases in order to be considered for adaptation, is 1% –100%. The higher the threshold value, the more conventional the corresponding solutions will seem. Low threshold values lead to more creative solutions but cause immense effort in case adaptation and long response times. The maximum number of suggested solutions (max-number) offers a possible range of 1–100 (with 100 standing for all). Default values are provided for all four parameters.
After the selection of the caseset-name, here simply called supply-air1, a mouse-click on the OK button starts the construction of appropriate solutions. The output of SYN is the case set supply-air1. The solutions are represented in A4 and graphically via the FABEL-Idea preview panel, see Fig. 7.9.

In the same way SYN may be used to support the design of supply systems for return air, water, electricity etc. Even the design of layouts showing a very different topological structure may be supported. Here a new structural (case tree) representation must be specified. However, this does not influence the basic approach and mechanisms underlying SYN.

Current work aims at the reimplementation of SYN in the FABEL-Idea-prototype3. The final version of SYN implements the larger part of CA. Among others it will allow

- not only for graphical problem formulation but also for graphical solution presentation via DANCER,
- the incorporation of background knowledge representing knowledge about the equality of structural case representations as well as about geometrical transformations.
- the facility to change the representation language and the transformation procedures required to represent cases structurally.
- optimal case base organization, whereby optimality is defined via a quality measure to be computed from the required solution confidence and the available response time.
- explicit statements about the solution confidence.
- explanation capabilities, i.e., the system will be able to graphically present the prototype that was applied to design a suggested solution.
Figure 7.9: Solution preview panel
Chapter 8

Exact and Correct Placement of Pipes (ANOPLA)

Wolfgang Gräther

8.1 The Idea

The exact placement of pipes in the service plenums of buildings is a particular design problem. In this design step the horizontal co-ordination of the various subsystems takes place. Such subsystems are typically supply-air, return-air, water-supply, and water-return systems etc. The spatial units for this kind of co-ordination are floors or part of floors.

ANOPLA\textsuperscript{1} uses domain knowledge provided by the ARMILLA system [Haller, 1985] to solve this design problem. ARMILLA provides us with generic templates, rules, and heuristics for installations. More than that ARMILLA forces a particular model of usage. This model envisages an increasing resolution of the design solution; this is achieved by way of stepwise refinement. \textit{Starting point for ANOPLA} is a strategic plan with the principle layout\textsuperscript{2} for the various subsystems. Such a strategic plan connects the vertical pipes of the various subsystems with the corresponding areas of connection for the inlets and outlets. The connection network satisfies the special requirements of the different subsystems and has typical access patterns (cf. [Haller, 1985, pp 20]).

\textit{Case adaptation} is a second task ANOPLA supports. A complex case adaptation is necessary when the pipes in the retrieved case do not fit the installation templates derived from the actual service plenum. ANOPLA is interactive in the sense that the user can stop and restart the execution at every time. The architect can also fix pipes at definite positions. Resizing and movement of pipes during the problem solving process will be visualized.

The knowledge provided by ARMILLA – together with the component based building system MIDI [Haller, 1974] –, heuristics acquired from our expert, and the representation

\textsuperscript{1}ANOPLA is an acronym of the german term ANOrdningsPLAnung (arrangement planning).

\textsuperscript{2}The bounding boxes for pipes are placed on grid lines.
scheme A4 ([Hovestadt, 1993a] and [Hovestadt, 1993c]) for design objects are prerequisites for structuring the problem of spatial arrangement of pipes. Given this structuring the problem can be formulated as a spatial assignment problem. In the domain of building design an agent approach seems to be very natural and promising. Our work is strongly influenced by Ludger Hovestadt and other related work of FABEL, especially [Bhat, 1995] and [Morgenstern, 1993].

The next section will point out the knowledge needed and its representation. We then outline the concepts, structures and functions used in ANOPLA. A detailed example is given in section 8.4.

8.2 Knowledge and its Representation

ANOPLA uses data and knowledge from different knowledge sources. These sources are:

- design objects in A4 representation,
- strategic plans or cases as a starting point,
- templates for pipe placement, and
- rules and heuristics concerning the process of arranging and the quality of the achieved arrangement.

All these are described in more detail in this section.

8.2.1 Design Objects

The A4 representation locates each design object in a 13 dimensional design space [Hovestadt, 1993a]. Besides the dimensions x, y, and z all design objects have values for the dimensions aspect, morphology, resolution, size, and other dimensions like time and user. Thus design objects have spatial, temporal, and semantic information.

ANOPLA needs only parts of the data of design objects. The data for the pipes shown in Fig. 8.1 are listed below. We use a list representation with keywords.

```lisp
((:x 120 :dx 255 :y 105 :dy 30 :z 730 :dz 30
   :aspect return-air :morphology connection :resolution bounding-box :size 6)
 (:x 345 :dx 30 :y 105 :dy 390 :z 700 :dz 30
   :aspect return-air :morphology connection :resolution bounding-box :size 6)
 (:x 345 :dx 255 :y 465 :dy 30 :z 730 :dz 30
   :aspect return-air :morphology connection :resolution bounding-box :size 6))
```
Figure 8.1: Planning grid with 3 return-air pipes (bounding boxes) in 2d-view on x and y axis. Note that pipes from west to east and pipes from south to north are placed on different layers according to ARMILLA.

8.2.2 Strategic Plans and Cases

When planning with the ARMILLA system a strategic plan is an intermediate design solution. Such a strategic plan fixes the principle layout for the pipes of the various subsystems. Special layout requirements of the different subsystems are satisfied. The diameters of pipes are also fixed. For example, sizing is done according to domain knowledge about flow rates at outlets. An example of a strategic plan is shown in Fig. 8.2.

Figure 8.2: A simplified strategic plan with 2 different subsystems (cf. [Haller, 1988]). Only a few bounding boxes of pipes are placed on the same grid line.

With respect to ANOPLA there are four types of knowledge that can be derived from strategic plans:

1. Knowledge about *forms*; these forms should be kept.
2. Knowledge about *connections* of pipes; these connections should also be kept.
3. Knowledge about sizes, especially diameters of pipes; these diameters constrain the placement of the pipes in the service plenum.

4. Knowledge about the number of pipes; the number is necessary for checking if the space reservation in the service plenum is sufficient.

Cases as a starting point deliver also the four types of knowledge as strategic plans do. Additionally, we get information about correct placement of pipes in a specific service plenum. Sometimes this information can be used to easily adapt the case (most of it) to the design problem at hand.

8.2.3 Templates

The ARMILLA system structures the service plenum horizontally into layers and vertically into lines for branches and lines for pipes. All pipes in layer O1 run in one direction, pipes in layer U1 run in the other direction. Layers O2 and U2 are used for twig lines. Twig lines connect pipes to inlets and outlets. The number of layers and number of lines depend on the size of the planning grid and the type of construction. The generic structure of a grid element is shown in Fig. 8.3 (cf. [Haller, 1985, p 17]).

![Diagram of ARMILLA planning grid and spatial structure](image)

Figure 8.3: ARMILLA: Planning grid and spatial structure of the service plenum. Thick black lines mark the grid, the broad areas near the grid are spaces for branches.

The ARMILLA structure of the service plenum is also used to define definite axes for pipes. These axes depend only on the diameters of the pipes (small: up to 50 mm; medium: up to 150 mm; large: up to 350 mm). Thus a finite number of patterns exists; we call such patterns templates. Figure 8.4 shows three generic templates; for a detailed motivation and a more complete overview please refer to [Haller, 1985, pp 22].
Figure 8.4: ARMILLA: Three generic templates for pipes with small, medium and large diameters. Only templates for the layers O1 and O2 are shown. A horizontal flip generates the corresponding templates for U1 and U2.

Figure 8.5: MIDI: Two templates for different layers.

The generic ARMILLA templates have to be adapted according to the construction system. Using the component based building system MIDI leads to specialized templates like the one shown in Fig. 8.5. Note that the steel beams are also placed in the service plenum. Therefore, a few lines for pipes are not available and positions of axes change compared to the generic templates.

8.2.4 Rules and Heuristics

So far we know that:

- Pipes have to be placed in different layers according to their direction; this is not true for twig lines.

- Pipes placed on a grid line in a strategic plan can only be moved ‘one grid’ to the left or right (top or bottom).

- Pipes with large diameters should be placed first.

- Pipes should be placed compactly so that large portions of free space become available. This space can then be used for other installations or equipment.

This list is probably not complete and could be changed after our first experiments with the implementation of ANOPLA.
8.3 Approach

From an artificial intelligence point of view the problem of correct arrangement of pipes falls into the category of assignment problems. The complexity of these problems is well known; in general such problems are NP-complete. There are a lot of ‘classical’ approaches and problem solving methods. An inference structure is given in [Karbach and Vöß, 1992]. Simulated annealing as a problem solving method could be a second type of approach [Aarts and Korst, 1989]; metaphorically spoken one can think of shaking the pipes from the grid lines into correct places. Shaking is done with decreasing physical power – the starting power should be so large that the pipes were moving at maximum ‘grid size’ distances.

Our approach is different. We try to solve the problem of arrangement with a distributed-agent-based strategy including a decentralized control structure. Each pipe is controlled by one agent attempting to assign a legal position. Our decentralized control structure is comparable to distributed hill climbing with no backtracking (cf. [Luo et al., 1994]).

The ANOPLA system will become one module (tool) in FABEL’s next prototype. The use of ANOPLA should be naturally integrated into the architect’s workflow. There is a distributed data server for A4 design objects; the different tools are also part of the distributed environment. Typical purposes are editing, 3d-visualization, calculating of costs, retrieving of cases etc. These tools can easily be attached to designs or parts of designs.

In summary there are the following requirements for ANOPLA:

- the arrangement process should be visualized,
- the arrangement process should be interruptible, and
- the arrangement process should be fast.

Input for ANOPLA are templates and a problem which could be a strategic plan (that is part of the actual design) or a case. According to rules and heuristics ANOPLA tries to find a correct solution. The steps from input to output are listed below.
1. Decompose the problem. Figure 8.6 shows a strategic plan with two separate problems P1 and P2. These subproblems can be solved one after another or concurrently.

2. Check 'solvability' of the problem. A necessary condition is that the templates provide enough space for the pipes. We are not quite sure if this condition is sufficient too in our domain.

3. Determine for each pipe the connecting pipes.

4. Instantiate each pipe as software agent that can perform the behaviour listed below:
   - *Move* pipe from grid line to a correct place.
   - *Move* pipe from one place to a better empty place.
   - *Move* pipe from one place to a better not empty place while pushing away another (thinner) pipe.
   - *Resize* (shorten or lengthen) pipe according to movement of connecting pipes.
   - *Send new* position to connecting pipes.
   - *Look* around for better places.
   - *Deactivation*.

5. If no agent is active then the problem is solved.

All changes should be visualized immediately. The user will see cycles should they occur and could then stop the process. This approach has one problem: due to the absence of global control and a backtracking mechanism we cannot guarantee that all possible combinations of placements will be tried. We hope that in our domain this circumstance does not lead often to serious problems.

8.4 Example

The example below (cf. Fig. 8.7) shows a strategic plan that could be an input for ANOPLA. This intermediate design contains bounding boxes for pipes of the subsystems supply-air, water-return, and water-supply. Note that some of the pipes are overlaid by others. This strategic plan is taken out of [Haller, 1988].

Unfortunately we cannot show the intermediate steps with moving and resizing pipes. In Fig. 8.8 a solution is shown. This solution is correct with respect to the generic ARMILLA templates.

8.5 Implementation

The implementation of ANOPLA is currently under development. For testing this approach the prototypical implementation will be done object orientedly in Common Lisp (CLOS); maybe we later change to OZ [Smolka et al., 1993]. In more detail:
Figure 8.7: A strategic plan as input for ANOPLA.

Figure 8.8: Solution to the problem given in Fig. 8.7. All pipes are placed correctly.
- Agents are implemented as instances of classes.
- Behaviour of agents is realized in methods of the corresponding classes.
- Templates formulate the constraints.
- Decomposition, solvability check and determination of connecting pipes are methods of a problem manager.

With this first implementation step we test our approach. In a second step we extend it to case adaptation.

---

3 The asynchronous behaviour of agents can only be simulated.
Chapter 9

Operators Used in Routine Design (Roude)

Oliver Jäschke and Dietmar Janetzko

9.1 The Idea

In planning buildings an initial state is incrementally transformed until a goal state is accomplished. In this process a number of intermediate states or sub-goals are traversed, and a number of requirements have to be met. This is equivalent to saying that a number of tasks have to be tackled. If tasks have to be carried out that can be described with respect to

- the objects involved,
- the requirements to be obeyed, and
- the problem solving rationale to be pursued;

these tasks may be described as routine design tasks [Brown and Chandrasekaran, 1989]. Even in very complex domains, e.g., building design, there are some sub-tasks that qualify as routine design. Dividing a given area into subareas according to some criteria (e.g., minimal and maximal size or number of the subareas) is an example of a routine design task. To find out which tasks are of this type a task analysis may be conducted [Janetzko et al., 1994a]. A task-analysis does not only help identifying which tasks are routine design and which are not. In addition it provides guidelines for developing operators that may be applied to accomplish routine design tasks. We specified and implemented three exemplary operators that accomplish routine design tasks. The usage of these operators depends very much on the integration with other problem solving approaches. In a word: Using routine design operators works fine whenever routine design is concerned. However, there are other subtasks in a complex task domain like building design that require more
complex problem solving methods or that ought to be allocated to the user.

We will now give a very simple example that shows how operators are used to carry out routine design tasks. The example is taken from the domain of building design; more specifically, it is a task concerned with designing a supply air system. The description of the example makes use of a specific notational system that is called A4 (cf. section 1.1).

9.2 Example

Description of the Example Task. On a very general level, one may say that this task is concerned with dividing an area into subareas according to a number of requirements. This is a routine design task that recurred a number of times in more complex design endeavors. The output of this routine design task is embedded into other design tasks that are concerned with linking the resulting areas to other objects according to some requirements etc. The area that has to be divided into subareas is a zone that defines possible installation places for supply-air on the level of a floor of the building. This zone is abbreviated by zab4. This zone has to be divided into subzones. A subzone of this kind is abbreviated by zab6. The subzones define possible installation places for supply-air on the level of the rooms of a building.

![Diagram of zab4 and zab6](image)

Figure 9.1:

9.3 Knowledge and its Representation

Knowledge Needs. To carry out this task knowledge concerning the placement and extension of the zab4 is needed. In addition we need the size of the rooms that provide the length of the square-sized zab6. This information is needed since the zab6 have to fit the size of the rooms.

Requirements. A partition of a zab4 only qualifies as a solution if it meets two requirements:

- The zab6s should be squares
- The zab4 should be exactly covered by zab6s
9.4 Approach

Procedure. We start calculating the zab6 by the vector used to represent a zab4. Instead of the full-blown vector we only use the first four entries that is called the reduct of the vector. The reduct is denoted by $(x, dx, y, dy)$. If $c$ is the length of the square-sized zab6 then we simply have to ask how many square-sized zab6s fit into a zab4. This can be calculated by referring to the extension of the zab4, which is encoded by $dx$ and $dy$. We assume that the extension of all objects is respecting the so-called MIDI-raster that is used to achieve a standardization of the lengths of all objects used in A4. For this reason, we may assume that both $dx$ and $dy$ are multiples of $c$. The number of zab6 that fit into a zab4 is $m \times k$ with $m$ being defined by $m := dx/c$ and $k := dy/c$.

Thus, the result of the operator can be denoted by a list of $m \times k$-many elements $(w_{ij} \mid 1 \leq i \leq k, 1 \leq j \leq m)$, with $w_{ij}$ being defined by

$$w_{ij} := (x + c \times (j - 1), c, y + c \times (k - i), c).$$

9.5 Implementation

An elaborated documentation of the rationale of employing operators used in routine design and its implementation has been published elsewhere [Jäschke and Janetzko, 1994a].
Part V

Comparison
Chapter 10

Comparison of the Modules

*Katy Börner and Gerhard Strube*

The chapters in this volume testify to the variety of support FABEL now supplies for design. The different approaches cater to different tasks, as well as different modes of problem solving, e.g., knowledge-lean algorithms for routine design in the FABEL domain (ch. 9), or support for assessment of designs based on rich domain knowledge (ch. 3), or support for adaptation of cases in CBR by parallel algorithms in the style of agents (chs. 5 and 6). The architecture of the FABEL system [Schmidt-Belz, 1995] allows these diverse approaches to coexist side by side in the same system, operating on the same data, and making themselves available to the user of FABEL by means of the same interface. This wealth of approaches, and the different philosophies behind them would, of course, not be adequate in a finished product, but defines the range of state-of-the-art approaches that can now be tested and compared with respect to practicability and efficiency in the third FABEL research prototype. The very diversity of the approaches taken makes them difficult to compare theoretically, however. As many discussions among the members of FABEL have shown\(^1\), a comparison has to take at least the following characteristics into account: the domain of support, i.e., the *functionality* and the *scope of application* provided by the modules and the *domain knowledge* employed, i.e., its *content*, *representation formalism*, and *dynamics*.

10.1 Domain of Support

By *domain of support* we denote the *functionality* supplied by a module and the main *scope of application* it was designed for.

\(^{1}\)This chapter summarizes several discussions among the contributors of this report as well as Andrea Enzinger, Barbara Schmidt-Belz, Elisabeth-Ch. Tammer, Angi Vöß, Markus Knauff, and Eberhard Pippig.
10.1.1 Functionality

The functionality required for design design (usually for 'routine' work) may be characterized as follows:

(i) Retrieval of 'similar' cases which delivers relevant solutions from the case base that may be useful in solving the present design problem. Retrieval methods may be, but need not be combined with other CBR techniques like adaptation. In fact, retrieval is covered by some of the modules that do adaptation. Some modules require the user to select a case by other means (e.g., manually, or by means of some other retrieval module; see the Fabel report on retrieval by similarity: [Voß, 1994]).

(ii) Assessment in order to check designs, either the present one, or designs from the case base with respect to the present problem. Assessment procedures may be used to check parts of the design of a building with respect to a set of predefined criteria, or constraints, e.g., checking for completeness and connectedness of pipes (DOM), or checking against 'valid' arrangements of A4 objects (CHECK-UP). CHECK and DOM present the result of their evaluation via textual representation (listing predicates that have been left unsatisfied), or graphically (highlighting problem zones by blinking).

(iii) Adaptation of solutions (which have usually been retrieved from the case base, but could also be provided by the user) to the problem at hand. Some adaptation modules (TOPO and AAAO) take the result of an assessment of the design as input. TOPO provides adapted solutions by reduction of a layout (retrieved from the case base) to a common subgraph with respect to the problem at hand. AAAO and AgentEx use an agent-based approach, were the agents iteratively move and negotiate in order to arrive at an adapted solution. Steps taken in the adaptation process are graphically depicted on the screen.

(iv) Construction, employing general knowledge (like rules cast into algorithms, as in ROUDE, or templates according to the ARMILLA schemas and instantiated by agents, as in ANOPLA, or schemas abstracted from cases, as in SYN) to produce solutions for (mostly routine) design problems.

Table 10.1 provides an overview about the support modes (i) to (iv) offered by the modules introduced in this report. The main support mode of each module is marked by x. Here, the solution is presented to a user or to another module. Support modes (like retrieval or assessment) are distinguished according to the accessibility of their result. Support modes which may be requested and used by an external user or module are denoted by e. Support modes which proceed invisible to an external user are referred to by i.

We would like to emphasize that all the modules presented in this report are integrated into the architecture and the user interface Dancer of the Fabel prototype, and hence provide...
graphical interaction with the user. They take layouts as an input and provide graphical output (except for assessment). Thus, it is easy to incorporate them into the workflow of architects. The user is any time in a position to change or reject design proposals or to select the support mode and module by themselves.

### 10.1.2 Scope of Application

Most of the modules are specialized to support a specific task, like the distribution of pillars in basic construction, or the layout of supply air pipes. This is what we call the scope of application. It relies on the task decomposition [Janetzko and Börner, 1993; Janetzko et al., 1994a] and has already been mentioned and graphically illustrated in chapter 1, Tab. 1.4 to Tab. 1.6.

The scope of the modules essentially depends on the domain knowledge and the knowledge representation employed as well as on the reasoning mechanisms used to support subsequent tasks. The less task specific the selected knowledge representation formalism, the broader the scope a module can cover. However, often these representations are of less expressive power than the ones that have been especially selected for a certain task.

For instance, the primitive topological relations in TOPO occur between building objects of any subsystem, and a statistical analysis can automatically elicit usual and unusual relations between certain types of objects. Other modules use more special knowledge, like statical constraints on the distribution of columns in AAAAAO, constraints on supply air systems in DOM and AgentEX etc. Here, the scope of the modules is limited by the scope of the knowledge used so far. Note, however, that other kinds of knowledge or different knowledge representations can be employed to extend the scope of application of a module. For instance, CHECK can use other kinds of complex topological predicates and the ontology of DOM can be extended either.

### 10.2 Domain Knowledge

In order to execute or support some functionality on the basis of these data, each module has to employ domain-specific knowledge which may be characterized as being truly episodic knowledge (e.g., cases) or more general knowledge, like the generalized episodic knowledge (e.g., the 'prototypical cases' abstracted by SYN), and knowledge which originates from the architectural philosophy applied in our domain (i.e., building according to
MIDI, layout of installations according to ARMLa). The general v. episodic distinction is but one of the relevant characteristics of domain knowledge. Equally important are what is represented (i.e., the content), and how (i.e., the selected representation formalism).

10.2.1 Content

Content is defined as the set of domain knowledge characteristics that are applied by a module. For instance, a module may ignore the task decomposition, or utilize it for its own operation. If more general knowledge is used, it may come from different sources: From cases and whole projects (by machine learning, or through knowledge acquisition), from the architectural models ARMLa and MIDI, or even from geometry (e.g., using algorithms for reflection and rotation), and statics as rules of construction engineering. MIDI and the task decomposition structure are models that comprise both domain knowledge for design and control knowledge which helps to plan the steps of design. Grids, rules and templates provided by MIDI and ARMLa may be regarded as general domain knowledge.

As for episodic knowledge, three kinds of sets of A4 objects may be distinguished:

(i) projects which comprise the set of all A4 objects in a building,

(ii) cases according to the task dependency analysis [Janetzo et al., 1994a], which may be 'cut' from projects automatically, and

(iii) cases that are sets of A4 objects arbitrarily defined by the user.

Considering the number of objects in a project (usually some ten thousands), projects as such are clearly unsuitable for CBR. Therefore, comparably small selections have to be used. While arbitrarily selected subsets may result in a creative and hardly repeatable, basis for, e.g., adaptation, the task decomposition structure provides a rationale for the selection of A4 objects which rests on knowledge engineering analyses of the problem-solving process in design.

<table>
<thead>
<tr>
<th></th>
<th>Check</th>
<th>DOM</th>
<th>TOPO</th>
<th>AAAO</th>
<th>AgentEx</th>
<th>SYN</th>
<th>ANOPLA</th>
<th>ROUDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i)</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(ii)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(iii)</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 10.2: Survey of the episodic knowledge employed by the modules

Depending on the content of domain knowledge applied, a module may use only part of the A4 values of the user's selection of A4 objects that constitutes the 'problem', e.g., only topological information, or features like aspect, scale, etc., or both, according to the knowledge it uses.

As for generic knowledge, we may distinguish the following categories of content:
• specialized declarative knowledge, i.e., knowledge of typical design patterns like (iv) case schemata from previous projects (e.g., acquired by learning techniques from cases), (v) templates, (vi) grids, or (vii) rules taken from MIDI or ARMILLA,

• procedural knowledge, like (viii) geometrical algorithms, e.g., for rotation, reflection, or scaling of a pattern, or algorithms for computing the maximum common subgraph.

• control knowledge, e.g., the (ix) MIDI model or the (x) task-structure.

<table>
<thead>
<tr>
<th>generic</th>
<th>Check</th>
<th>DOM</th>
<th>TOPO</th>
<th>AAO</th>
<th>AgentEx</th>
<th>SYN</th>
<th>ANOPLA</th>
<th>ROUDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(iv)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(v)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>(vi)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>(vii)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>o</td>
</tr>
<tr>
<td>(viii)</td>
<td>x</td>
<td>o</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(ix)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>(x)</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>o</td>
</tr>
</tbody>
</table>

Figure 10.3: Survey of the generic knowledge employed by the modules.

Table 10.2 and Tab. 10.3 provide surveys of the domain knowledge applied. A fundamental distinction is between knowledge explicitly and knowledge encoded in algorithms. The former is marked by x. The latter is referred to by o.

10.2.2 Representation Formalism

Knowledge, both declarative and procedural, may be cast in different forms. Episodic knowledge in FABEL comes in different varieties, too: (i) attribute-value lists, (ii) relations or predicates, (iii) algebraic ground terms, and (iv) graph structures. Generic knowledge takes the form of (v) algebraic terms representing structural commonalities of case structures, (vi) geometric constraints, (vii) rules of different content (e.g., formulae for computing similarity of cases, or rules for routine construction ('operators') in ROUDE, or substitution rules denoting proper instantiations of case schemas in SYN etc.), and a representation of the task-structure similar to a (viii) semantic network.

Tab. 10.4 provides an overview of the knowledge representation formalisms used by the modules.

10.2.3 Dynamics

The effort needed to acquire the huge amount of data for design support is enormous. Here machine learning methods may be employed to acquire and organize knowledge automatic-
By 'dynamics of domain knowledge' we mean the degree to which a module can change the knowledge it uses. While general domain knowledge acquired from the ARMIL IA book and other sources, and encoded manually, may be considered as static (i.e., it does not change unless the programmer will change it), knowledge may also be acquired by means of machine learning techniques, e.g., generalized case schemata, or rules for assessment based on statistical characteristics of cases. CBR in itself may be considered to be a form of learning, if cases are added to the case library, or if the case library is reorganized from time to time.

There are three modules which apply learning techniques to preorganize knowledge and thus to increase the efficiency of support. Check-L learns the predicates which are valid for a certain combination of objects. The predicates are used to assess layouts designed subsequently. TOPO derives a statistics about the frequency of topologic relations in specific layouts. This knowledge is used to assess the result of adaptation. SYN reduces the manual knowledge acquisition effort by generalizing the topology of specific cases into prototypical topologies. During reasoning these prototypes are used to derive solutions for new problems.

Two modules, namely Check and SYN, offer another interesting feature. With the help of a task decomposition structure, the modules are able to take a whole project as input and determine ('cut') cases automatically. Therefore the task-structure is projected on the projects. According to the logic of the task decomposition, the objects which correspond to the preconditions of a task constitute the problem part of a case, whereas the subset of A/4 objects defined by the task (e.g., A/4 indexes like zab6) may be regarded as the solution part of the case.

---

5Our intention is not to have machine learning techniques replace knowledge engineering. Rather, the task decomposition structure (a result of knowledge engineering) provides the knowledge that enables automatic acquisition of cases from architectural projects.
10.3 Concluding Remarks

The overview given in the present report shows the wide range of ideas employed in FABEL to converge on a common goal, i.e., the development of approaches for design support. At present, however, implementation and testing has not been yet completed for all of the modules. Integration into the third prototype of the FABEL system and ensuing tests with realistic case bases are, however, necessary conditions for a well-founded comparison and evaluation of the modules presented here. Therefore, we have to leave statements about the validity of claims concerning the range of applicability and judgments of usefulness in practical work to a future FABEL publication.
Bibliography


