Design Simplification by Analogical Reasoning

by
Marton E. Balazs

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

_____

December 1999

APPROVED:

_____
Dr. David C. Brown, Major Advisor

_____
Dr. Lee A. Becker, Committee Member

_____
Dr. Ashok Goel, External Committee Member
Georgia Institute of Technology, College of Computing

_____
Dr. Stanley S. Selkow, Committee Member

_____
Dr. Micha Hofri, Head of Department

Abstract

Ever since artifacts have been produced, improving them has been a common human activity. Improving an artifact refers to modifying it such that it will be either easier to produce, or easier to use, or easier to fix, or easier to maintain, and so on. In all of these cases, "easier" means fewer resources are required for those processes. While 'resources' is a general measure, which can ultimately be expressed by some measure of cost (such as time or money), we believe that at the core of many improvements is the notion of reduction of complexity, or in other words, simplification. This dissertation presents our research on performing design simplification using analogical reasoning.

We first define the simplification problem as the problem of reducing the complexity of an artifact from a given point of view. We propose that a point of view from which the complexity of an artifact can be measured consists of a context, an aspect and a measure. Next,

we describe an approach to solving simplification problems by goal-directed analogical reasoning, as our implementation of this approach. Finally, we present some experimental results obtained with the system.

The research presented is significant as it focuses on the intersection of a number of important, active research areas - analogical reasoning, functional representation, functional reasoning, simplification, and the general area of AI in Design.

Acknowledgments

First, I would like to thank my advisor, Dr. David C. Brown, for all his support and guidance during my tenure at WPI. The other members of my reading committee, Dr. Lee Becker, Dr. Ashok Goel and Dr. Stanley Selkow, provided many useful ideas that contributed to this research and raised several interesting issues that strengthened and broadened my understanding of analogical reasoning, design and complexity and improved this dissertation. Thanks also to all the faculty who during grand school helped me to build and broaden my knowledge of computer science.

Many people in the Computer Science Department at WPI contributed to my progress in the graduate school and to this dissertation in one way or another. My office mate Dan Grecu has been a great colleague and friend throughout my entire stay in Worcester. The discussions with him often helped me a lot in my work and in difficulties I often had faced

in my every day life, not to mention that the many things we share due to our similar past, allowed us many fun minutes to unwrinkle our foreheads. Other people of the AI in Design Group at WPI, especially Jon Kembel, Ming He, Peter Bastien, Janet Burge made the life in graduate school much more enjoyable.

Graduate school in general is stressful and the years I spent at WPI were not different. Fortunately I found some great friends especially George and Nina Galica, Bogdan Vernescu and Florin Frigioiu, who made me feel at home in Worcester.

My family has been always a great source of support in my life in general and in my work in particular. Most of my successes in school over the years are due to my parents, Balazs Marton and Margit, who taught me the right priorities and always encouraged me to learn and work hard.

Finally and most importantly, I want to thank my wife Aniko for her love and support, and for putting up with me and my life as a grad student (and not only). To her I promise that one day I'll settle down and stop going to school.

Table of Contents

Abstract . . . . . **i**

Acknowledgments . . . . . **iii**

Table of Contents . . . . . **v**

CHAPTER 1 Introduction **1**

The Goal of the Research **2**

The Importance of and Motivation for the Research **5**

Expected Benefits of the Research **7**

The Simplification Problem **7**

Specifying a Simplification Problem **8**

Possible Approaches to Solve a Simplification Problem **9**

Simplification using Analogical Reasoning **11**

The Problems Raised **11**

The Approach Proposed 13

Example 19

Measuring the Complexity of Designs 25

Simplifying Designs by Analogical Reasoning 28

Methods and Expected Results 29

Organization of the Dissertation 34

CHAPTER 2 The Problem 36

Simplification 37

The Simpler Relation 37

Measuring Complexity 38

Complexity of Designs 56

Why Count when Measuring Complexity? 67

The Simplification Process 69

Propagation of Simplification 71

Performing Simplification 72

Possible Approaches 73

Simplification by Analogical Reasoning 75

Difficulties Raised 79

Retrieving Useful Simplification Examples 79

Mapping Simplification Problems 85

Transferring Simplifications 87

Evaluating the Result of the Simplification 88

Storing new Simplifications 89

CHAPTER 3 Related Work 90

Work on Analogical Reasoning 90

Model-Based Analogical Reasoning 92

Goal-Driven Analogical Reasoning 94

Work on Abstraction 96

Work on Reasoning about Designs 99

Work on Design Optimization/Simplification 102

Suh's Information Content Reduction 102

Bashire & Thomson's Estimation of Design Effort 104

Boothroyd & Dewhurst's Complexity Factor 106

Reasoning about Designs from different Points of View 107

CHAPTER 4 The Approach:

Simplification by Goal-Directed Analogical Reasoning 109

Simplification as a Problem Solving Goal 110

Representing Simplifications 114

Explaining a Simplification 115

Elements Relevant to a Simplification 119

Relevance Calculation 121

Collecting the Elements that are not Absolutely Irrelevant 122

Propagating Relevance inside Objects 123

Organizing Simplifications 128

Organizing Simplifications for Retrieval 128

Organizing Simplifications for Knowledge Transfer 134

The Analogical Reasoning Process 138

Retrieving 138

Mapping 145

Transferring Simplification Knowledge 154

Evaluating the Result of the Simplification 164

Generalization and Storing 166

CHAPTER 5 Application: Simplification of Designs 168

The Door Lock Domain 169

Representing Designs 173

Representing Structure 173

Representing Behavior 175

Representing Function 178

Connections and Dependencies between the Different Aspects 181

Contexts, Aspects and Measures for Design Complexity 184

Contexts for Measuring Design Complexity 184

Aspects for measuring Design Complexity 185

Measures of Design Complexity 185

Structural, Behavioral and Functional Design Simplification 196

A Structural Simplification 198

A Behavioral Simplification 199

A Functional Simplification 205

CHAPTER 6 Implementation 208

The System 208

The Database of Known Simplifications 210

The Interface Module 212

The Data Management Module 212

The Simplifier Module 217

The Simplification Abstraction Module 223

Representation 225

Implementation of the Abstraction Mechanism 227

Implementation of the Analogical Reasoning Mechanism 229

CHAPTER 7 System Demonstration 230

Simplification of an Arithmetic Expression 230

The Sample Problem and Issued Raised 231

Operation of the System 233

Simplification of the Personal Fax Design 237

The Sample Problem and Issued Raised 238

Operation of the System 240

CHAPTER 8 Experiments 243

Demonstrating that the Simplification System is Effective 245

Setting up the Experiments 248

Results and Discussion 250

Measuring the Effect of Using Relevance 253

Setting up the Experiments 255

Results and Discussion 257

Measuring the Effect of Using Different Complexity Measures 259

Setting up the Experiments 260

Results and Discussion 261

Conclusions 263

CHAPTER 9 Conclusion 264

Contributions 264

Future Work 269

Performing Further Experiments with the System 270

Improve the Usability of the System by Building a GUI 270

Extending the System to other Types of Simplification 271

Adding New Application Domains 271

Studying the Simplification Propagation Problem 271

Studying the Possibility of Generating Creative Simplifications 272

Bibliography 274

Appendix A 282

Appendix B 295

Appendix C 304

Appendix D 307

CHAPTER 1

Introduction

Ever since artifacts have been produced, improving them has been a common human activity. Improving an artifact refers to modifying it such that it will be either easier to produce, or easier to use, or easier to fix, or easier to maintain, and so on. In all of these cases, "easier" means less *resources* are required for those processes. While 'resources' is a general measure, which can ultimately be expressed by some measure of *cost* (such as time or money), we believe that at the core of many improvements is the notion of *reduction of complexity,* or in other words, *simplification*. For instance, the less complicated an artifact is, as measured by the number of parts it consists of, the easier it will be to manufacture. It is clearly the case that the cost of the actual manufacturing process will depend on the technological sophistication of the manufacturer, experience and skill of the workers and so on. However, as opposed to cost, the complexity of an artifact gives an objective characterization of the difficulty of its manufacturing.

Simplification, as a very important method of artifact improvement, is a goal-driven activity. Such goal-driven activities have been the ‘motors’ of many creative acts. While great inventions and discoveries have often been made by noticing connections, drawing analogies or using metaphors, they almost always were driven by some goal, which quite often was to improve, or to simplify something.

Thus, studying simplification is a very important direction of research. On one hand, it targets the understanding and simulation of a basic human (cognitive) activity. This can lead to important results from both theoretical and applicative points of view. On the other hand, the study of simplification, may set a context for studying human creativity as a by-product of goal-driven reasoning processes.

1.1 The Goal of the Research

This dissertation is concerned with the representations and reasoning required for *Simplification* in general and *Design Simplification* in particular.

A design can be considered as *simpler* than another for a variety of reasons, such as shape, use, or ease of assembly. Simplification, as a consequence, can be done with a variety of goals, such as to simplify the shape, the use, or the ease of assembly of a design. In addi-

tion, simplification can be done in a variety of ways. For example, simplifications might be searched for (reasoned out) or retrieved.

The research presented in this dissertation studies simplification by analogy with stored simplifications. The designs are represented as function, plus behavior, plus structure. Simplifications may occur for each of these aspects, with consequences for the other aspects. For example, changing the behavior may make a component redundant. We call this *propagation of simplification*.

The two general hypotheses of our research are that simplification of designed objects is an important class of problems that is worth a special study, and that an effective and efficient approach to solving problems of this class is to reuse known simplifications by analogical reasoning. As a consequence of these hypotheses we proposed that the following problems need to be investigated:

- How to define and represent simplification problems.
- What special techniques are needed in the analogical reasoning process to make the design simplification problem solving effective and efficient.

The definitions we hypothesize as needed refer to a *complexity measure for designs* and a *simpler-than relation.*

To use analogical reasoning to solve design simplification problems we propose that the following subproblems - raised by developing an analogical reasoning model for design simplification - must be investigated and solved:

- use of the goal, as formulated in the simplification problem, to guide the phases of the analogical reasoning process;
- effective and efficient retrieval of source analogs (known simplifications) under the assumption that a simplification may refer to only some part of a design;
- effective and efficient evaluation of mappings between source analogs and the target under the assumption that those mappings are between deep structures as opposed to the shallow structures used to demonstrate analogical reasoning models in cognitive science research;
- transfer of simplification knowledge to the target whether the source analog (simplification) is described by the simplification process or just by the differences between the designs involved.

In the concluding chapter of this dissertation we will present the results of our investigation with respect to the hypotheses and subproblems described above.

We have no knowledge of any ongoing research concerning simplification or simplification of designs. The propagation of changes across levels of representation (e.g., from structure to function) is, as far as we know, a new research area, and one that appears to be very challenging. The design simplification problem area in general raises a large number of interesting research issues concerning functional reasoning and functional representation schemes, and their interaction with analogical reasoning.

1.2 The Importance of and Motivation for the Research

The research presented in this dissertation is significant as it focuses on the intersection of a number of important, active research areas — analogical reasoning, functional representation, functional reasoning, simplification, and the general area of AI in Design.

We investigate the process of design simplification at different *levels* (i.e., structural, behavioral and functional). While there is some simplification-related work in the Engineering community, such as Design For Manufacturing (DFM) and Design For Assembly (DFA), their work is mainly concerned with the structural view. We believe that design

simplification is a significant problem that hasn't been addressed by AI, and that our approach provides a fresh view.

Each of the levels at which a design may be simplified can set simplification goals. The dissertation examines simplification as a goal-based activity and proposes an analogical reasoning model to perform it. Goal-based analogical reasoning is one of the important areas of the research presented.

Simplifying a design from one level may affect the other aspects. Studying the propagation of simplifications across levels is also a significant aspect of our research on design simplification. This dissertation does not propose a solution to the propagation problem. It presents, however, the problems raised by the propagation of simplifications.

The dissertation presents both theoretical and practical results. We hope that these results will have significant theoretical impact on the field, as well as a strong potential to impact design applications, and possibly other application areas.

The results from this work have the capability of influencing the next generation of design systems. The techniques developed and implemented in our prototype system can be fur-

ther developed and refined, and included in practical CAD tools, which then will be able to assist designers to produce simpler designs and analyze proposed simplifications.

1.3 Expected Benefits of the Research

In addition to the obvious benefits of any new research results, better understanding of the simplification of designs might eventually lead to cheaper, better designed products.

Design has been chosen as the domain because it is a rich source of representations and problem solving activity. It is a type of human activity that is still not very well understood, and, consequently, is a natural target for AI.

1.4 The Simplification Problem

Simplifying an object means to reduce its complexity. We view complexity as a way of characterizing objects from a given *point of view,* that is *context*, *aspect* and *measure*. A *context* for characterizing an object by its complexity refers to a process that can be performed on the object (e.g., describing it, producing it, using it and so on). For a given context, an *aspect* is the collection of those elements of the objects which play a role in their characterization in the context considered. For a design an aspect can be its structure, that

is its components, relations between components and attributes. Finally, for a given context and aspect, a *measure* is a function that assigns to an object a numeric value that characterizes the complexity of the object in the given context and aspect (e.g., counting the components of a design can be a measure of complexity defined for the context of manufacturing, in the aspect of structure: it characterizes the number of components that have to be manufactured before the design can be completed).

Given an object, a point of view (context, aspect and measure), and a set of constraints on the object, the *simplification problem* is the problem of transforming the object such that the resulting object satisfies the constraints and such that its complexity, as measured for the given point of view, is less than that of the original object.

1.4.1 Specifying a Simplification Problem

A simplification problem is defined by three elements: the object that has to be simplified, the point of view of the simplification and properties of the object that the simplification has to preserve. These three elements correspond respectively to the *object*, *goal* and *constraint* of the simplification problem.

1.4.2 Possible Approaches to Solve a Simplification Problem

We view the simplification process as a search in some search space (e.g., design space in the case of design simplification). The goal of the search is to find a simpler object than the one given as the starting point. Note that we do not define simplification as an optimization problem (i.e., with the goal to find the least complex object), but rather an improvement problem. Also, simplification is a constrained search because all simplification problems require the preservation of some properties of the object (for example, design simplification is, or should be, a function-preserving process). There are several possible ways we could try to solve a simplification problem.

One possible approach to performing simplification is to view it as an optimization problem with a complexity measure as the objective function. For instance one could apply local transformations known to reduce complexity and organize them into a hill-climbing type of process. Structural simplification of a mechanical design could be approached by applying basic simplification operators, such as removing redundancy (e.g., removing two gears from a line of connected gears).

Another possible approach to performing simplification would be to perform some heuristic search. Having some knowledge of what operations and what sequences of operations

may lead the search towards “good” simplifications, would overcome the deficiencies of the local optimization approach, which is the major problem with uninformed search methods. One general problem of this approach is the lack of good heuristics. As far as we know there are no general (domain independent) heuristics for simplification and many specific domains also lack extensive simplification rules.

Finally, a third approach to performing simplification is reusing known simplifications to produce new ones. This could be done either by reusing known simplifications from the same domain as the problem is in (i.e., by case based reasoning), or from a different domain (i.e., by analogical reasoning). In this dissertation we propose this latter approach as the best one for solving simplification problems.

In our research we propose the study of using analogical reasoning for simplification in general, and design simplification, in particular. Our approach extends the general model of analogical reasoning with mechanisms for using the simplification goal to guide the processing.

1.5 Simplification using Analogical Reasoning

Approaching the simplification problem by analogical reasoning would have several benefits. First, a known simplification can be reused over and over for identical simplification problems. Second, even if a new simplification problem is not identical to any known one, if some (significant) similarity between the two can be discovered, the old simplification may be used as an "idea" for simplification. Finally, simplification by analogical reasoning also has the benefit that it may be capable of producing general simplification principles by learning and abstracting over the simplifications produced.

1.5.1 The Problems Raised

Simplification by analogical reasoning requires the solving of a number of problems. First a way of representing the known simplifications must be defined. Such a representation must contain all the elements that are needed for solving a simplification problem in general, as well as elements that would allow the application of analogical reasoning. The minimal set of elements that the representation of a simplification has to contain, consists of: a) the objects involved in the simplification (the original object and its simplified version) and b) the simplification process that has been applied to transform the original object. As we shall see, the second one is needed for the analogical transfer. In this disser-

tation we propose, that in addition to these two elements, the representation of a simplification also contains a description of those elements of the objects which were actually involved in the simplification. This will allow our analogical reasoning model to concentrate only on elements that are *relevant* to simplifications.

Another problem that needs to be solved is the design of the data structures for organizing known simplifications. These data structures must support all the phases of analogical reasoning. Designing these data structures needs to be done in parallel with building the model for analogical reasoning, since they will be strongly interdependent.

Finally, a model of analogical reasoning needs to be defined. For this a set of general and specific issues must be solved. To solve the general issues we need to answer the following questions:

- How might the retrieval of an analog occur?
- How will the retrieved analog be mapped onto the given problem?
- How will the mapping be used to transfer the problem solving knowledge?
- How can the solution to the problem be completed if needed?
- How will a solution to the problem be evaluated?

- How can a generalization over the analog and the solution to the problem be built?
- Will the generalization and/or the solution of the target problem be stored into the database of problems for later use?

Specific issues refer to particular aspects of the simplification problem which could be used by the analogical reasoning mechanism to improve performance and/or the quality of the result produced. Such issues are for example, whether the simplification goal could be used to improve retrieving, mapping and knowledge transfer, what role could the simplification goal play in producing and evaluating the solution, or how could the simplification goal be used to generate useful abstractions for generalizing over simplifications.

1.5.2 The Approach Proposed

The approach to solving simplification problems presented in this dissertation is based on what we call “goal-driven” analogical reasoning. Goal-driven means that the simplification goal stated in the problem will be used all through the analogical reasoning process to improve the performance of the processing and/or the quality of the result. We had to take the simplification goal into account both in defining the representation of simplifications and in designing the data structure for organizing known simplifications.

We propose that the representation of a simplification consists of the representations of the following elements:

- the two objects involved in the simplification (the more complex and the simpler one);
- the explanation of the simplification, which specifies either the difference between the two objects involved, or the process by which the more complex object was transformed into the more simple one;
- the set of those elements of the objects, which according to the explanation, played some role in the simplification (we call these elements *relevant elements*);

The explanation is used for two purposes. One one hand it is the basis for determining which elements of the objects involved in the simplification are relevant. On the other hand it can be used to build abstractions over simplifications, with the purpose of organizing simplifications into hierarchies. Such hierarchies are useful for the analogical transfer of simplification knowledge, as well as for generating general simplification rules and/or principles.

The relevant elements are central to this research because they are used to focus the processing on all the phases of the analogical reasoning process to only those portions of objects that are involved in some simplification.

The data structure we are proposing for organizing known simplifications was designed along three dimensions. First, the set of known simplifications is partitioned based on their goal (context, aspect and measure) into classes of simplifications having the same goal. Second, simplifications are organized along their “more complex object” component. The reason for this is that the retrieval of a known simplification similar to a problem is done by matching the object specified in the problem to an object that has already been simplified. Along this dimension, simplifications are organized into several hierarchies corresponding to the types of elements used in the representation of the objects involved. Finally, the known simplifications are organized into simplification hierarchies. These hierarchies are based on abstractions over the explanations of the simplifications.

The reasoning process we propose to use for simplification was derived from a quite general model of analogical reasoning and proceeds as follows (Figure 1 presents the intermediate results of the different steps in the processing):

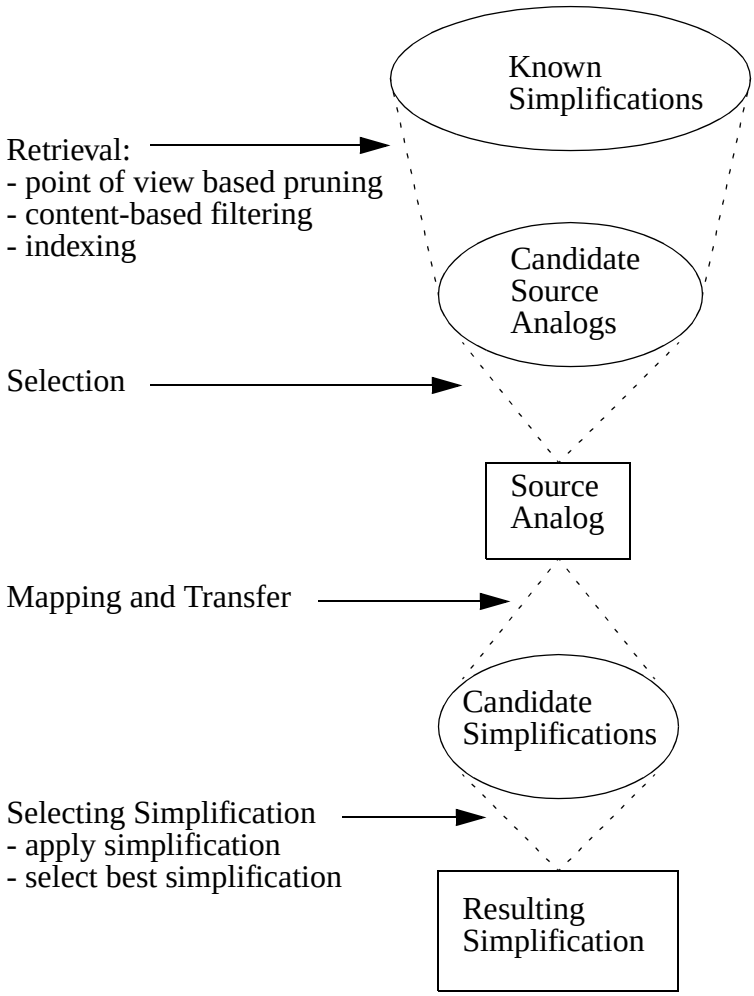


FIGURE 1. The process of producing a simplification. The ovals represent sets of simplifications, with larger ones containing more simplifications, the rectangle represents one simplification.

- *Retrieval of candidate source analogs*: this phase selects from the set of known simplifications those that have the same point of view as the problem, and which are "similar" to the problem. Similarity is measured in terms of the number and kind of elements (e.g., components, relations and attributes) they share.
- *Selection of the source analog*: each candidate analog retrieved has associated with it a score which measures its similarity to the object to be simplified. This score is used to select the simplification that is closest to the problem.
- *Mapping of the source analog onto the problem*: this phase will produce several "global mappings" that are consistent sets of correspondences between relevant elements in the source analog, and elements in the problem.
- *Selection of the best global mapping*: each of the global mappings obtained will be evaluated for quality by combining the scores of the member correspondences (e.g., correspondences between relations will assigned higher scores than correspondences between attributes for analogical reasoning). The scores of the member correspondences are assigned at the time of retrieval. The global mapping with the highest score will be selected to be used for transferring the simplification knowledge.

- *Transfer of simplification knowledge*: the best global mapping will be used to produce several candidate simplifications by associating the unmapped elements in the source analog with elements in the problem.
- *Application and evaluation of candidate simplifications*: all of the candidate simplifications are applied to the simplification problem, producing new objects. The objects produced will be evaluated against the problem constraints and for the simplification condition. If an object produced does not satisfy the constraint or is not simpler than the object specified in the object, it is dropped.
- *Selecting the solution*: the object that has the minimal complexity from among those which satisfy the constraint and are simpler than the object to be simplified, will be reported as solution to the simplification problem.
- *Generalization and learning*: if the simplification that was applied is significantly different than the source analog it has been derived from, it will be added to the database of known simplifications. Also, if a useful generalization over the new simplification and the source simplification can be built, it will also be added to the database.

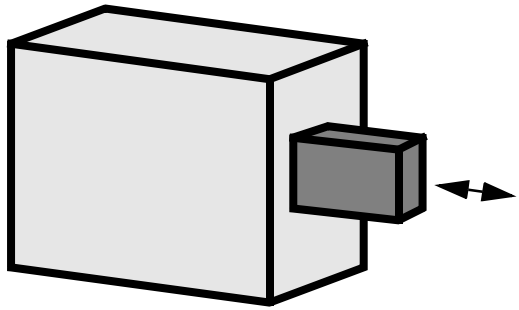


FIGURE 2. A schematic door lock

1.6 Example

In this section we will use a simple example to illustrate our ideas about the simplification of designs by analogical reasoning. The example will be drawn from the domain of designing simple door locks [Chakrabarti & Tang 1996].

We consider a door lock (Figure 2) to be a device that allows and prevents the opening of a closed door (or gate or window). It is composed of a *box*, and a *bolt* that can be fully retracted into the box as a consequence of some input applied. When the bolt is completely retracted it allows the door (together with the whole lock) to move into the open

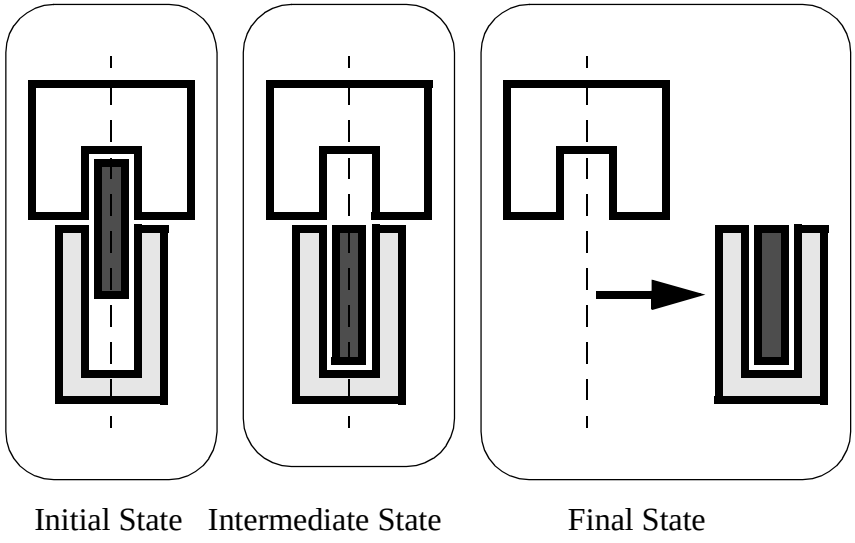


FIGURE 3. Behavior of a door lock

position. When the input is no longer applied the bolt returns to its initial (unretracted position).

For our purposes, the opening of the door lock is a three state process (Figure 3). The initial state corresponds to the closed door and is characterized by an unretracted bolt and a shut door. The second (intermediate) state is characterized by a retracted bolt and a shut door. The door lock may get into this state from the initial state as a consequence of applying some input. The final state corresponds to the open door and is characterized by a

retracted bolt and an open door. This state can be reached from the intermediate state by applying a second input to the door lock. The closing of the door lock can be described in a similar way.

In our examples we shall limit ourselves to door locks implemented using simple components with distinct functional roles (such as levers, cams, shafts, etc.). We shall also assume that the inputs to a door lock will be forces characterized by their directions.

Figure 4 (a) illustrates a design for the door lock device. We represent designs by their structure, behavior and function. Note that in this dissertation we do not study designs that achieve their function without a behavior. Figure 4 (b) is a graphical representation of the structure of the a door lock, consisting of components, relations (represented by thick, two-directional arrows) and attributes. Every device has at least one (intended) *function*.

A function of a device is defined in terms of its interaction with a given environment [Chandrasekaran & Josephson 1996] “We say that an object achieves its function if placed into the environment for which the function is defined, if it causes the interaction to happen, by virtue of certain of the properties of the device”.

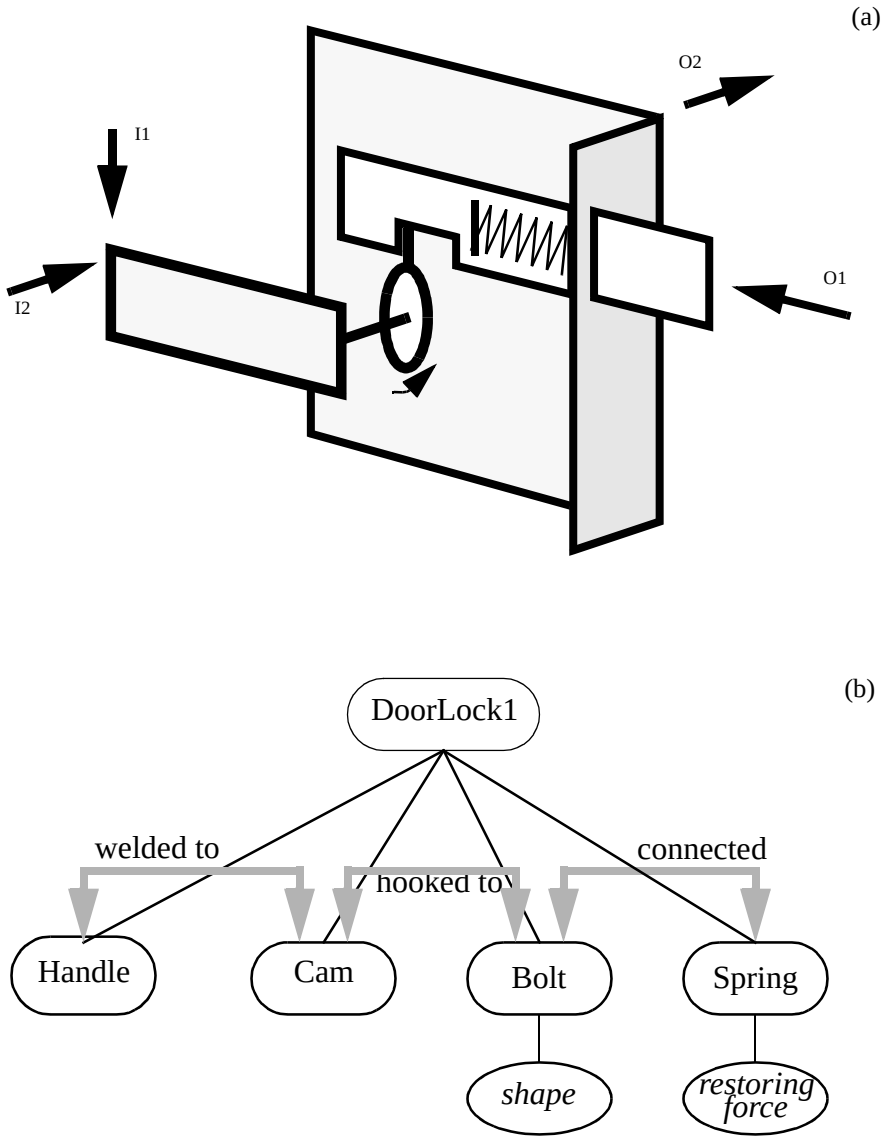


FIGURE 4. Door lock implementing using a cam: (a) schematic and (b) structural representation

```
Function: Open
 Environment:
    - input I1 applied to the handle
    - input I2 applied to the handle
 Interaction:
    - force_I1_applied > restoring_force →
      retract_bolt
    - bolt_retracted → apply_I2
 By (deployment):
    - Open_Behavior
```

FIGURE 5. The Open function of DoorLock1

Figure 5 gives a description of the Open function of the door lock represented in Figure 4. Figure 6 represents the behavior implementing the Open function. We view the behavior of a device as a process described by a sequence of state transitions. A state transition is specified by two (partial) state descriptions, the *initial state and the final stat*e, a *condition* and a specification of *how the state transitions are achieved*. A state transition may be achieved by a function or another behavior.

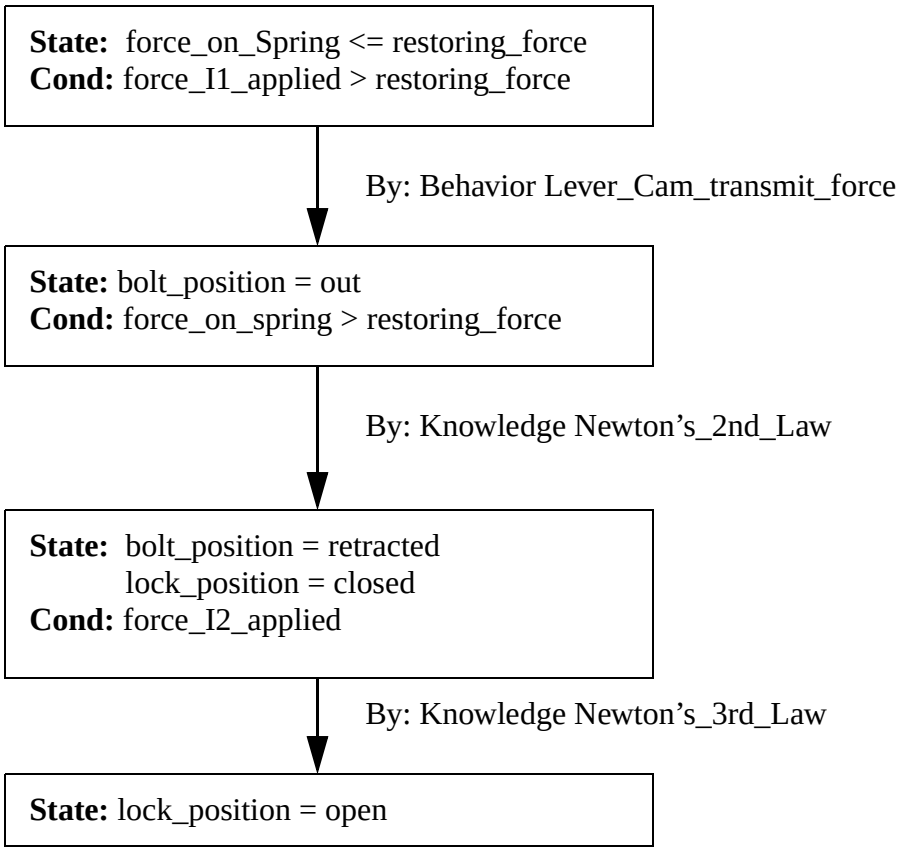


FIGURE 6. State transition graph for the top level behavior of DoorLock1

1.6.1 Measuring the Complexity of Designs

Let us now see how the complexity of a design, such as the door lock described above, can be measured. Since the definition of complexity is relative to a point of view, we have to start with defining contexts, aspects and measures for measuring complexity of designs.

The *contexts for measuring the complexity of designs* must be processes that can be applied to a design such as designing, manufacturing, using, repairing, maintaining and so on. Thus we may want to answer questions like “How complex is it to design the door lock?”, “How complex is it to manufacture the door lock?”, “How complex is it to use the door lock?” or “How complex is it to describe the door lock?” and so on.

For each of the contexts above there may be *aspects of a design* with respect to which complexity may be measured. In our research we will only be concerned with complexity for the aspects of structure, behavior and function of designs.

For each context-aspect combination that makes sense, we can define several *measures*. We will consider measures based on counting different elements of the design. What exactly needs to be counted will depend on those elements of the design in the aspect considered, on which the process corresponding to the context depends. For example, in the

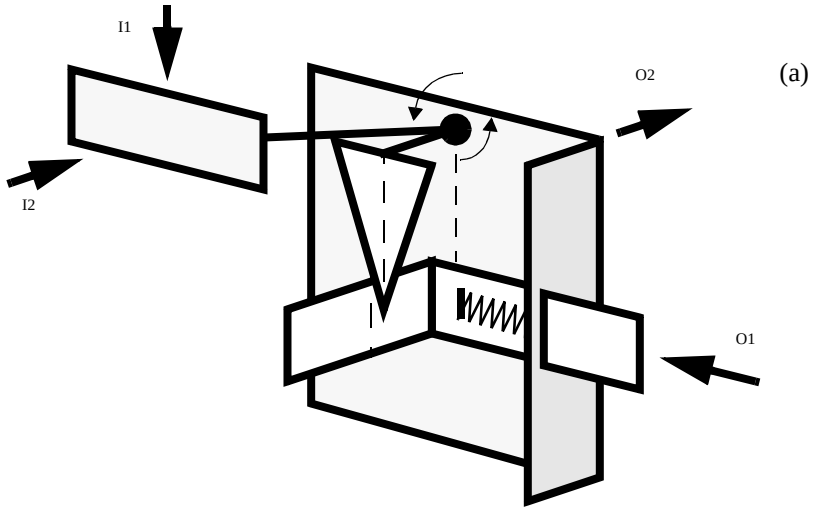


FIGURE 7. Door lock using a combination of two levers, a wedge and an L-shaped bolt

context of manufacturing the door lock, for the aspect of structure, a possible measure of complexity would be the count of components.

Let us illustrate some complexity measures that could be applied to our door lock example. Here we will restrict these examples to the context of *using the* door lock. A door lock that is *structurally* more complex than the door lock in Figure 4 is the one illustrated in Figure 7. This is true if we measure the complexity of the design by the count of components. The door lock in Figure 7 has 5 components (a handle, a lever, a wedge, a bolt and a spring), while the one in Figure 4 has only 4 components (a handle, a cam, a bolt and a

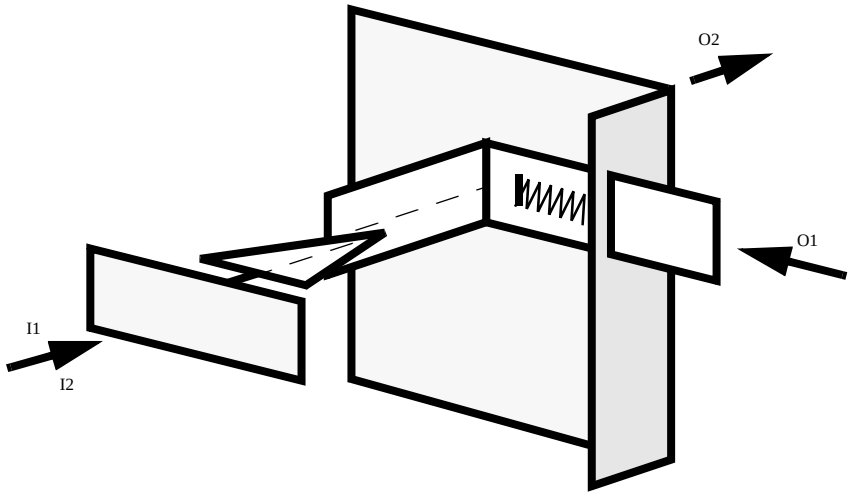


FIGURE 8. Door lock using a combination of a wedge and an L-shaped bolt

spring). In addition to the number of components, structural complexity may also refer to the number of relations between components as well as to the number of attributes of the design and/or its components. The door lock in Figure 4 is also *behaviorally* simpler than the one in Figure 7 because the state descriptions involved are simpler (this is obviously a consequence of having fewer components, but it is reflected in the behavioral representation as well). This behavioral complexity may refer, for example, to the count of states (or states transitions) in the behavioral description. Finally a door lock that is *functionally* simpler than the door lock in Figure 4 is shown in Figure 8. This is due to the fact that the

interaction with its environment contains fewer inputs (the same force will retract the bolt and push the door open).

1.7 Simplifying Designs by Analogical Reasoning

Simplification of designs by analogical reasoning relies on a collection of known simplifications. This collection is partitioned into three classes of simplifications, corresponding to the three aspects of designs, structure, behavior and function. This partitioning is done by marking (labeling) each simplification with the aspect to which it corresponds.

The analogical reasoning process is performed as described in Section 2. The most difficult problems raised by the simplification of designs are the transfer of simplification knowledge, the application of the candidate simplification and the evaluation of the resulting design. The first two problems occur when retrieving simplification knowledge from a different domain than the domain of the problem. The most straightforward solution is to use a hierarchy (or several hierarchies) of object classes and of simplifications. The evaluation of the object generated as the result of the selected simplification is difficult because the modification of one aspect of the design will propagate to other aspects. This propagation may lead to violation of the original requirements for the design. Thus the propaga-

tion has to be performed explicitly. The difficulty of propagation may range from no modifications needed to a complete redesign of the object. As a consequence we are not addressing it to any detail.

1.8 Methods and Expected Results

In this research we are proposing a way to solve simplification problems, in general, and design simplification problems, in particular. For this purpose we address two major questions: a) What is simplification? and b) How can simplification be performed?

To answer the first question, we are proposing a definition for the complexity of objects modeled by their structure, behavior, and function. We chose to use *structure-behavior-function* models as a basis for our definition of complexity because they are the most popular means of modeling physical systems, in general, and designs, our main domain of application, in particular. The criteria which we need to keep in mind for our definition are the following:

- Our definition of complexity must be operational, that is, the complexity of any object modeled by its structure, behavior and function must be *effectively* and *efficiently* computable. The requirement for efficiency is necessary because physical

systems may be very complicated and any process of evaluating them, including complexity, has to be performed efficiently, in order to be useful.

- We need to be able to explain the relation of our definition of object complexity to other definitions used in the literature and in practice, and clearly point out its advantages, and (possibly) disadvantages. For this we need to compare complexity measure, and relations between complexity measures obtained with our definition to complexity measures obtained with other known methods. We then need to explain the differences between the results of those measures (if there are any) as well as why those differences are useful for solving a simplification problem.

To answer the question “How can simplification be performed?” we did two things: a) built a model of our approach to solving simplification problems and analyzed it theoretically, and b) implemented a system based on that model and performed a set of experiments to demonstrate our approach.

We propose to solve simplification problems by using “goal-directed analogical reasoning”, that is, analogical reasoning in which the reasoning process is guided by the simplification goal. We base our problem solving model on an almost universally accepted model-based analogical reasoning process model [Bhatta et al. 1994]. The specifics of our model

consist in two aspects: First, the matching and mapping phases of the process are based on Falkenheiner's Structure Mapping Engine (SME) [Falkenheiner et al. 1993], and second, in each phase of the process, the relevance of object parts to simplifications is used to restrict the processing. We perform the theoretical analysis of our model by applying to it the known theoretical analysis results for the models from which it was derived (i.e., model-based analogical reasoning and SME) and estimating the impact on using relevance to restrict processing. We also perform experiments to measure this influence empirically.

We expect that the complexity of our model is not worse than that of known model-based analogical reasoning models and that using relevance significantly reduces the computation in the retrieving, matching and mapping phases.

Based on the model proposed we implemented a system for simplification by analogical reasoning. For the implementation we used the CLIPS language [CLIPS 1993]. Among other reasons, we chose CLIPS for our implementation because it supports rule-based programming, it supports object-oriented programming, it implements a set of powerful query operations, and it allows easy interfacing with other programming languages.

To use our system for demonstrating the goal-directed analogical reasoning approach to simplification we designed and performed a set of experiments. The goals of our experiments were the following:

- to demonstrate that the system is capable to produce simplifications using known simplifications from either within the application domain, or across domains;
- to measure how using relevance influences the resources required by solving simplification problems;
- to study how using different ways of measuring complexity from the same point of view (e.g. structural complexity) affects the results produced.

We present the results of these experiments in Chapter 8.

It should be clear that the effectiveness of any similarity-based problem solver heavily depends on the knowledge it can rely on. We wanted to make sure that for each problem presented to the system the simplification database contains enough knowledge to propose a solution. For this reason, in our experiments we used hand-coded simplification databases. This way we were able to demonstrate different aspects of the system and to test it for problems that we considered interesting or hard. Never the less, we also performed an experiment in which the simplification generated by the system was added to the simplifi-

cation data base and then reused in solving another simplification problem. For this experiment we have specially built the second simplification problem such that it would retrieve the newly added simplification as a good source analog.

We must note here that our implementation was only tested for structural simplification problems. However, the approach works for behavioral and functional simplification problems as well. We can argue for this based on our way of representing designs:

- A behavior of a given design is represented by a sequence of state transitions. This can be viewed as a decomposition of the behavior into *steps* which are connected by a *followed-by* relation. Each of these steps may be either an *elementary step* (in the case when the transition is achieved by a function), or may be a *composed step* (in the case when the transition is achieved by another behavior). Thus a behavior is represented by a tree structure with relations between sibling nodes. This representation of a behavior is similar to the structural representation of a design.
- A function of a given design is represented by the *environment* in which the design has to be placed into in order to achieve its function, the *interaction* of the design with the environment required to achieve the function, and the way the function is *deployed.* Functional simplification refers to either the simplification of its interac-

tion with the environment, or to the simplification of how it is deployed. We view an interaction as a sequence of input and output pairs and we represent it in a way similar to representing behaviors (i.e., using a structure similar to a sequence of state transitions). The mode of deployment of a function is represented by a behavior that implements that function. Consequently, based on the discussion in the previous paragraph, the representation of function of a design uses structures similar to those used in the behavioral and structural representation of designs.

In the concluding chapter we will describe what changes need to be made to the current implementation of our system in order to be able to solve behavioral and functional simplification problems as well.

1.9 Organization of the Dissertation

The remainder of this dissertation presents our approach to simplification by goal-driven analogical reasoning in more detail. Chapter 2 presents the simplification process by breaking it down into subproblems, as well as possible approaches to solving those subproblems. Chapters 3 relates the work presented to work in other fields, such as analogical reasoning, abstraction, reasoning about designs and design optimization. Chapters 4 and 5

describe our approach in detail. While Chapter 4 is a general description of our proposal to solve simplification problems, Chapter 5 specifically refers to the simplification of designs, the additional problems raised by it and our solutions to those problems. Chapters 6 through 8 present the validation of the approach. Chapter 6 gives a short theoretical analysis of the simplification process with emphasis on time complexity of the processing. Chapter 7 describes the implementation of a simplification system based on our approach. Chapter 8 presents the experiments performed with the implementation for demonstrating the approach. Chapter 9 summarizes the contributions of the research and sets goals for future work.

CHAPTER 2

The Problem

The problem addressed in this dissertation is: What is simplification and how can simplification in general, and design simplification in particular, be performed in an effective and efficient manner? This chapter gives a detailed description of the problem. First we present our views on what simplification is. We use examples from different domains to illustrate the issues raised when trying to define simplification. We conclude the first section with a definition of simplification. The rest of the chapter defines the simplification process, presents ways simplification could be performed, describes the general approach to solving problems by analogical reasoning, presents the issues raised by using analogical reasoning in general and for design simplification in particular. These issues constitute the subproblems our research proposes to solve.

2.1 Simplification

Simplification is a process by which an object is transformed into another, simpler object. Intuitively we say that an object A is *simpler* than another object B, if B is more *complicated* than A. Thus, to define what simplification is we need to define what we mean by "simple" (or equivalently "complicated") and how "degrees" of simplicity can be me measured.

2.1.1 The Simpler Relation

Using a more precise term for our definition we say that object A is simpler than another object B, if the *complexity* of A is smaller than the complexity of B. This definition suggests that to study the nature of the "simpler" relation we first need to define what we mean by the complexity of an object. As we shall see in the followings this is by no means a trivial problem.

Following the view adopted in Mathematics and Computer Science (see for instance [Brassard & Bratley 1996]), we define the *complexity of an object* to be *a measure of certain resource requirements for a given process performed on the object*.

2.1.2 Measuring Complexity

Our definition given in the previous subsection suggests that complexity is not an absolute measure. We may define several complexity measures for the same object, depending on different factors. In the followings we will use several examples, from different domains to illustrate what factors might be considered when defining the complexity of an object.

2.1.2.1 Measuring the Complexity of Mathematical Expressions

The purpose of a mathematical expression (or "expression" for short) is to describe a set of numbers. If the expression contains variables it describes the set of all numbers that can be obtained by *substituting* (legal) values for each of those variables and *evaluating* the resulting arithmetic expressions. Two expression which evaluate to the same number for all the possible substitutions of the variables are said to be *equivalent*. For simplicity, in this subsection we will only consider expressions built using constants, one single non-negative integer variable, the four arithmetic operation signs and parentheses.

One reason for the importance of measuring the complexity of an expression would be to estimate the effort needed to evaluate it. Based on such a measure one could decide whether a given expression can be evaluated in a reasonable amount of time or not, or which of two equivalent expressions to evaluate for a faster result. However, as we shall

see, evaluation is not the only process that can be applied to an expression. We will show that those other processes may require a different view on the complexity of an expression.

The following example will illustrate the problems raised by defining a complexity measure for expressions.

Consider the following three equivalent arithmetic expressions:

$$A(n) = \frac{(n+1)(n+2)}{(n-1)(n-2)(n+4)} \quad \text{(EQ 1)}$$

$$B(n) = \frac{n^2 + 3n + 2}{n^3 + n^2 - 10n + 8} \quad \text{(EQ 2)}$$

$$C(n) = \frac{\frac{1}{n} + \frac{3}{n^2} + \frac{2}{n^3}}{1 + \frac{1}{n} - \frac{10}{n^2} + \frac{8}{n^3}} \quad \text{(EQ 3)}$$

How do these three expressions compare from the point of view of their complexity? Unfortunately there may be several different answers to this question. In the following we will refer to the structures of the three expressions represented in Figures 1, 2 and 3 respectively.

First, we could measure the complexity of each of the expressions by the number of elements (i.e., constants, variables, operators and parentheses) needed to *write* them. Such a

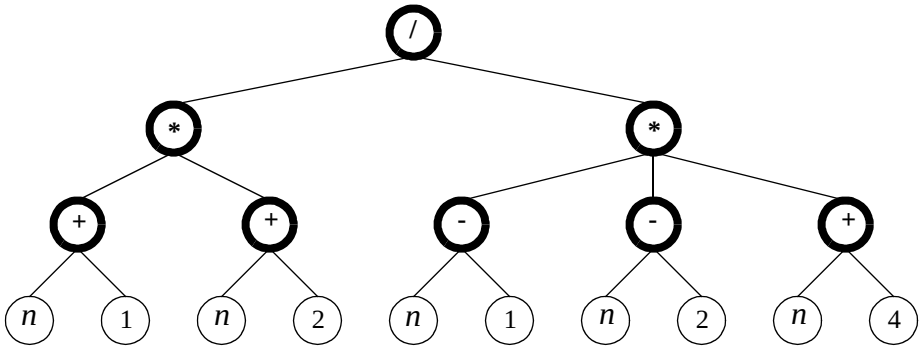


FIGURE 1. Structure of expression A(n)

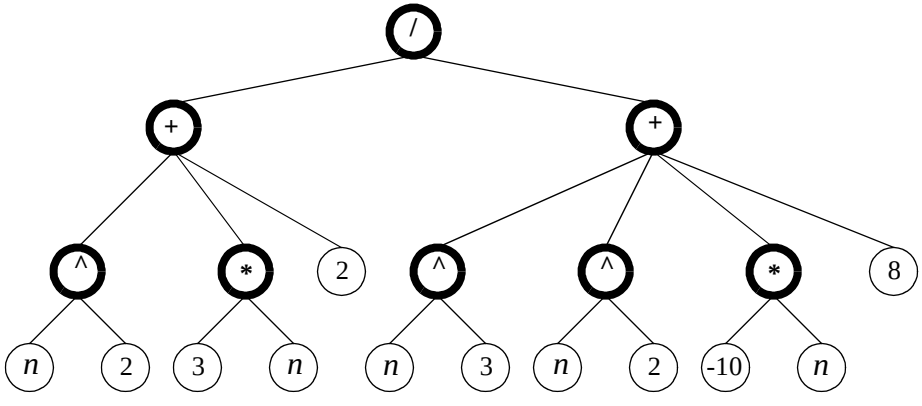


FIGURE 2. Structure of expression B(n)

measure would be a good basis for estimating the effort needed for the process of writing expressions. If we chose this count as the measure of complexity of an arithmetic expression, then we would say that expression $B(n)$ is the simplest (least complex) because it

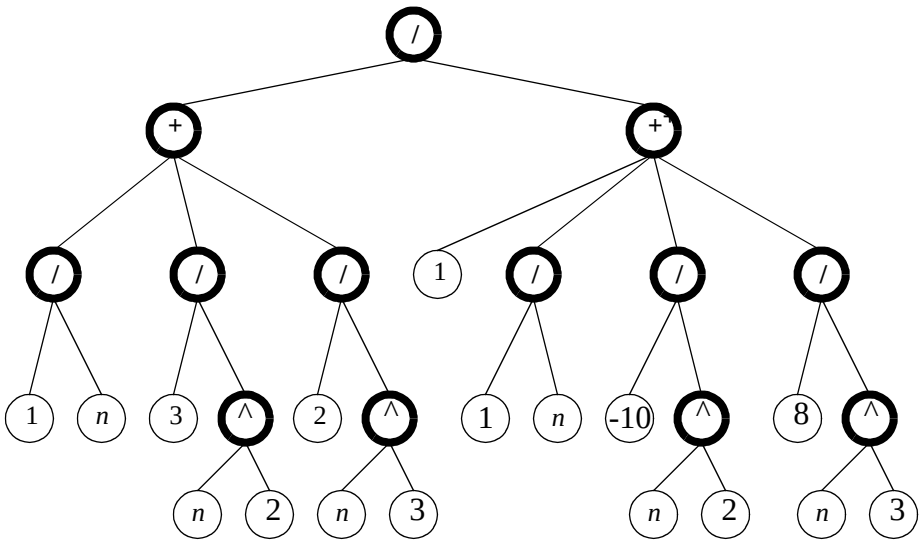


FIGURE 3. Structure of expression C(n)

only uses 18 elements, compared to 26 and 29 elements used by expressions $A(n)$ and $C(n)$, respectively.

Let us relate this measure of complexity to the structure of an arithmetic expression. We can describe the *process* of writing an expression based on its structure by the algorithm presented in Figure 4. Note that this algorithm will print out all the parentheses, even if they are not needed. To count the elements used to write an arithmetic expression using this algorithm we only have to count the elementary `write` operations performed by it. While this could be done using standard algorithm analysis techniques, it should be obvi-

```
WriteExpression (E)
begin
   switch type(E)
      case NUMBER:
         write E;
      case VARIABLE:
         write E;
      case EXPRESSION:
         O:= operator(E);
         A[1..n]:= arguments(E);
         write '(';
         WriteExpression(A[1]);
         for i:= 2 to n do
            write O;
            WriteExpression(A(i));
         end for
         write ')';
   end switch
end.
```

FIGURE 4. Algorithm for writing a fully parenthesized expression based on its structural representation

ous that the number of elements used for writing a given expression, represented by its structure can be found based on the *counting of terminal nodes and nonterminal nodes, plus their branching factors* (i.e., the number of descendants for a node in the structure tree) in the structural representation. Note that if we only use binary operations in our representation our measure of complexity would reduce to the counting of nodes in the representation tree and the consideration of the fact that the writing of each internal node requires three elements, the operator sign and two parentheses.

Another situation in which we may want to measure the complexity of expressions is if we

want to *evaluate* them. In this case the goal of measuring the complexity of an expression with respect to evaluation is to estimate the resources required during the evaluation process. One such resource, *time* could be estimated by counting the operations that need to be performed during evaluation.

Just by looking at our expressions we can tell that from this point of view expression $A(n)$ would be the simplest because it only requires 9 operations to be performed, compared to 11 operations for $B(n)$ and 13 for $C(n)$. However, let us again give a precise description of the process with respect to which we want to measure complexity (i.e. of evaluation). The algorithm in Figure 5 specifies this process. To calculate the complexity of an expression measured as the number of operations that need to be performed during evaluation we have to count the number of times the step of applying an operator, that is

```
result: = O(result,EvalExpresion(A[i]);
```

will be performed. Again, this can be easily done by *counting the internal nodes and their branching factors* in the structural representation. Also, a note similar to the one made at the end of the previous paragraph can be made, that is if the representation of the expression only uses binary operators the complexity from the point of view of the number of operations performed could be computed by counting the internal nodes in the representa-

```
EvalExpression (E): NUMBER
begin
 switch type(E)
   case NUMBER:
     return E;
   case VARIABLE:
     return value(E);
   case EXPRESSION:
     O:= operator(E);
     A[1..n]:= arguments(E);
     result:= A(1);
     for i:= 2 to n do
       result:=O(result,EvalExpresion(A[i]));
     end (for)
     return result;
 end (switch)
end.
```

FIGURE 5. Algorithm for evaluating an expression based on its structural representation

tion tree.

Finally, let us consider another process with respect to which the complexity of an expression may be of interest: *understanding* the "behavior" of an expression for large values of the variable *n* it depends on. This process is important because it allows a characterization of an expression without actually evaluating it.

It should be clear that all of the expressions we have considered will have values closer and closer to *0* as the value substituted in for *n* is bigger and bigger. What is different however about the three expressions with respect to this process is that some of them

require more effort for obtaining the same characterization. This difference in the effort required has its explanation in the different complexity of the three expressions with respect to the process considered. By applying simple limit calculation rules known from calculus we would decide that $C(n)$ is the simplest because we only need to apply a fixed number of limit calculation rules to see that

$$\lim_{n \to \infty} C(n) = 0,$$

while both *A(n)* and *B(n)* need first to be transformed into expression *C(n)* to produce the same answer.

Since this latter statement may sound more vague than the ones given for the previous examples let us once again give a precise description of the process involved.

Figure 6 gives an outline of an algorithm which computes the limit of an expression (depending on a variable, when the variable goes to infinity) using a limited set of rules of limit calculation and of expression transformation. The structure of the algorithm is very similar to the evaluation algorithm. Essentially the difference consists in the case when limit calculation of a subexpression returns 'UNDEFINED' (e.g., when trying to compute $\lim_{n \to \infty} \frac{0}{0}$). To compute the limit in this case the expression needs to be transformed into another (equivalent) expression, for which the limit calculation rules may yield a result.

```
LimitExpression(E): NUMBER
begin
 switch type(E)
   case NUMBER:
     return E;
   case VARIABLE:
     return value(E);
   case EXPRESSION:
     O:= operator(E);
     A[1..n]:= arguments(E);
     i:= 1;
     repeat
       limit[i]:=LimitExpression(A[i]);
     until (i > n) or (limit[i] = UNDEFINED)
     if i>n then
       result:=ApplyLimitRule (O,limit);
       if result = UNDEFINED then
         rule:= SelectTransformationRule(E);
         TransformExpression(rule,E);
         return LimitExpression(E)
       else
         return result;
       end (if)
     else
       return UNDEFINED;
     end (if)
 end (switch)
end.
```

FIGURE 6. Algorithm for calculating the limit of an expression based on its structural representation

This (significant) part of the algorithm is “hidden” in the statements

```
rule:= SelectTransformationRule(E);
TransformExpression(rule,E);
```

The procedure `SelectTransformationRule` will perform a search to find an applicable transformation rule that will lead to a useful form of the expression. `TransformExpression` will then apply the transformation rule to calculate the limit.

Obviously the effort required for performing the (process of) limit calculation depends on the number of limit calculation rules that have to be applied. For a given expression the maximum number of limit calculation rules that will be applied is given by the *number of nonterminal nodes of the structural representation* (i.e., there will be one limit calculation rule for each elementary operation). However if expression transformations are needed, additional limit calculation rules will be performed. Thus the total number of limit calculation rules performed by the algorithm will depend on the number of nonterminals in the structural representation and the performance of the transformation selection.

We can conclude that if we want to compare two arithmetic expressions for complexity (simplicity) we may consider at least three different *context*s: description, evaluation and understanding (its behavior for large values of the variable). Note that each of these contexts is essentially a process applied to the object for which the complexity is being evaluated. For each of these contexts in which the complexity of an expression can be measured, different views on what complexity is are needed.

One intuition resulting from these examples is that different processes will refer to different elements of the structure. Also, some processes with respect to which complexity is to be measured may introduce new elements into the complexity which depend indirectly on the structure of the expression.

2.1.2.2 Measuring the Complexity of Algorithms

We will use the domain of algorithms to illustrate two further “problems” raised by defining complexity of objects. As examples we will use comparison-based sorting algorithms, for which complexity results are very well known. For describing the algorithms we use a Pascal-like pseudcode language.

The study of algorithm complexity (known as Analysis of Algorithms) is a very well researched area of computer science. Analysis of algorithms takes the view that the complexity of an algorithm is a measure of the resources it uses. Several different kinds of resources may be considered, such as time, storage, number of statements (source code lines for programs) and so on. Algorithm analysis typically addresses only the first two of these resources, the others being studied mainly as part of the field of software engineering.

There are several different contexts in which the complexity of an algorithm could be con-

```
Execute (A)
 step[1..n]:= Steps(A);
 for i:=1 to n do
    switch type(step[i])
       case ASSIGNMENT:
          arg(1,step[i]):=eval(arg(2,step[i]))
       case CALL:
          call(step[i])
       case FOR_LOOP:
          for k:=arg(1,step[i]) to arg(2,step[i])
             Execute(arg(3,step[i](k)));
          end for
       case .. other loop constructs ...
       case IF:
          if arg(1,step[i]) then
             Execute(arg(2,step[i]))
          else
             Execute(arg(3,step[i]));
    end switch
 end for
end.
```

FIGURE 7. Algorithm for executing an algorithm based on its structural description

sidered. For instance one could define an algorithm's complexity in the context of writing the algorithms (e.g., counting the number of statements used). Alternatively, one could define algorithm complexity in the context of execution (e.g., measuring the time or storage required for performing the algorithm).

We will restrict our discussion of the complexity of an algorithm to the context of *execution* (Figure 7 gives the description of a simple execution algorithm). Unfortunately, even

within this single context for algorithm complexity, the question “what is the complexity” of a given algorithm may have more than one answer, depending on which *aspect* of the algorithm we want to measure. We could for instance be interested either in the amount of time or in the amount of memory required by the execution of the algorithms.

Consider for example the Selection Sort and Merge Sort algorithms. Figures 8 and 9 respectively present algorithmic descriptions of these algorithms, together with a graphical representation of their structures. Well known algorithm analysis results show that for sorting an array of n objects Merge Sort requires $O(n\log n)$ comparisons, while Selection Sort requires $O(n^2)$ comparisons. That is, in the context of execution, under the aspect of “time required”, as measured by the number of comparisons performed, the complexity of Selection Sort is higher than that the complexity of Merge Sort.

Intuitively this complexity measure can be done again by counting elements in the structural description of the algorithms. However during the counting we need to take into account that certain elements in this structure of an algorithm play special roles. These elements are the so-called control structures (if-then-else, while, calls and so on). They allow for a short description of groups of operations that are alternatives to each other or are repeated several times.

```
SelectionSort (A[1..n])
   for i:= 1 to n-1 do
      minIndex:= i;
      for j:= i+1 to n do
         if A[j] < A[minIndex]
            then minIndex:= j;
      end (for)
      temp:= A[minIndex];
      A[minIndex]:= A[i];
      A[i]:= temp
   end (for)
end.                                    (a)
```

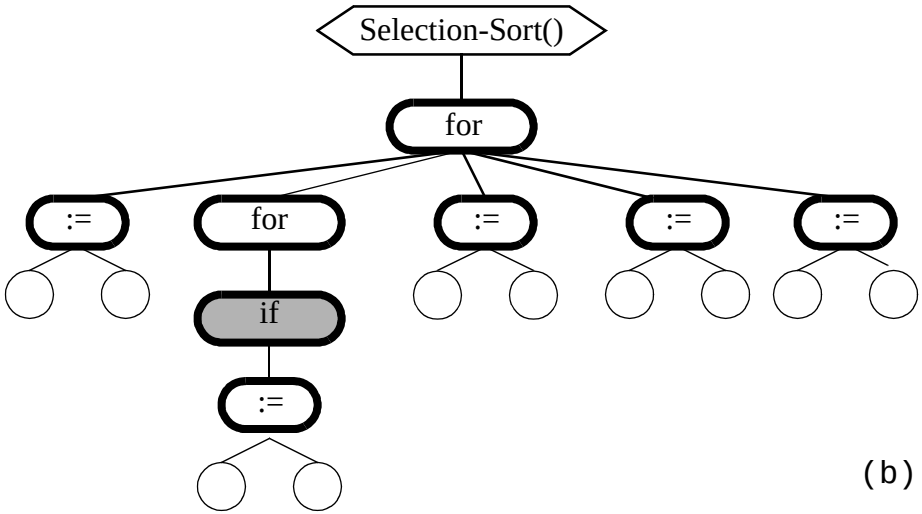
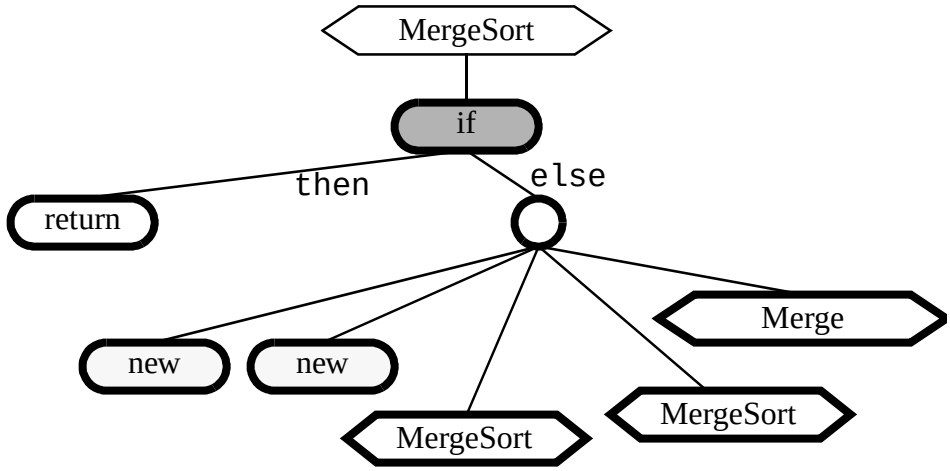


FIGURE 8. Selection Sort Algorithm (a) and its structure (b)

On the other hand, if we were interested in the aspect of “storage requirement”, we would conclude that the complexity of Merge Sort is higher than that of Selection sort. This is

```
MergeSort (A[1..n])
   if n=1
      then return;
      else
           new ALeft[]:= A[1..n mod 2];
           new ARight[]:= A[n mod 2+1..n];
           MergeSort(ALeft);
           MergeSort(ARight);
           Merge(A,ALeft,ARight);
           return;
   end (if)
end.
```

(a)



(b)

FIGURE 9. Merge Sort Algorithm (a) and its structure (b)

because each recursive call of MergeSort will allocate two new arrays, while SelectionSort sorts the array in place, without using any extra memory.

Again, the counting of memory allocations can be performed on the structure of the algo-

rithm, but taking into account the control structure elements.

Measures of Complexity

So far we have seen that to measure the complexity of an algorithm we have to decide on the context and the aspect of this measurement. Let us concentrate now on a third dimension of measuring the complexity of an algorithm.

Consider the problem of measuring the *time complexity of an algorithm* - that is, measuring the *time aspect* of the algorithm in the *context of execution*. As before our question is whether it is possible to measure the complexity with only the context and aspect specified. Unfortunately the answer is once again negative. The new problem we are facing here is deciding how exactly complexity will be measured. There are clearly several possibilities.

We could measure the physical time required for executing the algorithm. That is implement the algorithm in the form of a computer program, run it for the problem for which the time requirement has to be measured and measure the time of the run (more than likely with the computer's internal clock). This way of measuring the time requirement of an algorithm has several major disadvantages:

- The running time of a program will depend on how the algorithm was implemented (*implementation dependence*).
- Implementations in different programming languages may produce executable programs with different performances (*language dependence*).
- Even two implementation using the same programming language can take advantage of different features of that language (e.g. iterative versus recursive implementation), resulting in different time requirements (*coding dependence*).
- Different computers may have different execution speeds.The running time of a program will depend on the machine on which it will be executed (*machine dependence*).
- Even if the same program is run on the same computer several times, the execution times for different runs may be different due to the way the operating system manages the computer's resources (*operating system dependence*).

These disadvantages suggest that physically measuring running time is not appropriate for characterizing the time requirement of an algorithm, and, as a consequence it is not appropriate for comparing algorithm performances.

An apparently more precise way of measuring time complexity of an algorithm would be

to sum up the times (as given in the technical specifications) required by each of the machine statements in its implementation. While one could argue that this is a precise measure of the time required for running the program, the only disadvantage that it overcomes (of the ones listed above) is machine dependency.

Observations like the ones above led researchers in the field of algorithm analysis to approaches based on the following principles:

- Measure time complexity of algorithms by *counting the number of times certain operations will be performed*. We call these operations *significant operations*, suggesting that they significantly influence the time required by the execution of the algorithm.

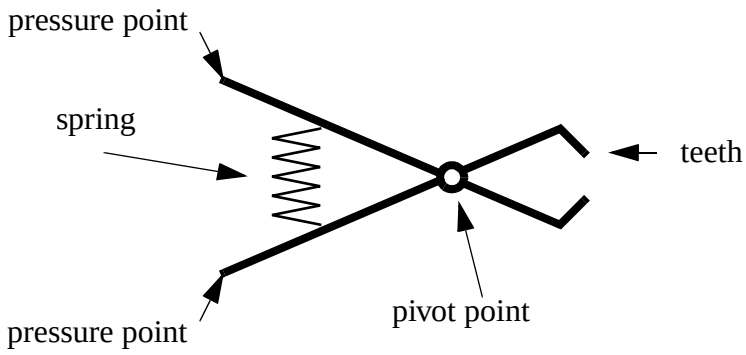
  An operation may be considered significant for several reasons, such as importance to solving the problem addressed by the algorithm (e.g., comparison is required to sort an array of objects in place), amount of time required by the operation is great compared to other operations (e.g., input/output in external sorting methods require much more time than any CPU operation), the operation is performed with high frequency compared to other operations used by the algorithm (e.g., loop control operation performed a great number of times will contribute to the execution time even if

they are neither specific to the problem, nor do they require individually a significantly greater amount of time than other operations in the algorithm).

- It may happen that for a given algorithm more than one significant operation is identified. It is always possible to concentrate on one of those operations at a time, however such an approach will disregard the possible dependencies between the different significant operations. A better idea is to *combine the numbers of different significant operations into a single expression*. For instance the average time complexity of the Selection Sort algorithm in Figure 8. can be given as $O(n^2)\boldsymbol{C} + O(n)\boldsymbol{S}$, where $\boldsymbol{C}$ and $\boldsymbol{S}$ stand for Comparisons and Swaps, respectively. Such a formula can then be used to compare the complexity of two algorithm either from the point of view of one of the measures, or from the point of view of some derived measure built from the formula.

2.1.3 Complexity of Designs

In this subsection we will use a simple example of mechanical design (a clothespin), meant only to illustrate our views on measuring complexity of designs. This example was taken from [Sticklen & al. 1989] and is shown in Figure 10. Figure 10 (a) is a schematic



(a)

(b)

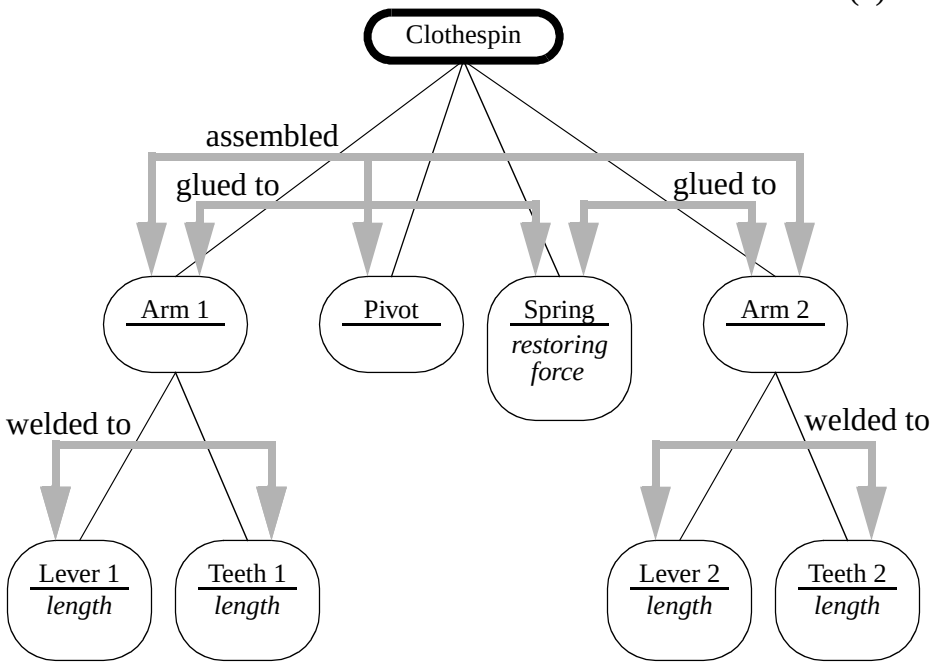


FIGURE 10. Simple Device - A Clothespin: (a) a schematic and (b) a structural representation

representation of a clothespin. It consists of two arms, each of them having teeth, a pivot connecting the two arms (and providing them with lever functionality) and a spring, connected to both of the arms and which provides a restoring force that will keep the clothespin in a "closed" state (that is, with the two sets of teeth touching). There are two intended functions of the device: *to maintain* [Keneuke 1991] the "closed" position, and *to achieve* (move to) and then temporarily maintain the open position.

Figure 10 (b) is a graphical representation of the *structure* of the device, consisting of components (the elementary components are represented by thin framed boxes) and relations (represented by thick, two-directional arrows).

Every device has (at least) one (intended) *function*. A function of a device is defined in terms of its interaction with a given environment [Chandrasekaran & Josephson 1996]. We say that an object achieves its function if placed into the environment for which the function is defined, it causes the interaction to happen, by virtue of certain of the properties of the device. To describe a function of a device the following elements need to be specified:

- the environment;
- the interaction of the device with the environment;

- the mode of deployment, that is, what properties and relations of the device, and relations between the device and the environment determine the causal interactions between the device and the environment.

If the mode of deployment assumes a sequence of state transformations of the design we say that the device achieves its function by a *behavior.* Obviously there are devices which achieve their functions without a behavior, or as it is (maybe improperly) said, by static behavior (i.e., not via state change). For instance, a chair doesn't behave, in the sense of changing its state, while achieving its function, that is to support a person sitting on it. In this dissertation we will be only concerned with devices that achieve their function through some behavior. In this case we also need to present our view on behavior and provide a way of describing behaviors.

We view the behavior of a device as a process described by a sequence of state transitions. A state transition is specified by two (partial) state descriptions, the *initial state and the final stat*e, a *condition* and a specification of *how the state transitions are achieved*. A state transition may be achieved by a function or another behavior. This approach allows a decomposition of the function of a device into a hierachy of behaviors and functions of its components.

```
Function: Open
 Environment:
    - force_applied to the pressure_points
 Interaction:
    - force_applied > restoring_force causes teeth_more_open
 By (deployment):
    - Open_Behavior
```

FIGURE 11. The Open function of the clothespin

Note, that in Chapter 5 we will give more precise definitions for the structure, behavior and function of a device as well as of the representations we use in our research.

For the clothespin a description of the *Open* function is shown in Figure 12, while Figure 11 represents the behavior implementing this function. In the literature on design representation behavior is usually described by graphs, similar to the one in Figure 11.

Let us now see how the complexity of a device, such as the clothespin described above, can be measured. In our proposal for measuring design complexity we build on the intuition presented in the previous examples from other domains.

Thus we have to start with defining contexts, aspects and measures for measuring complexity.

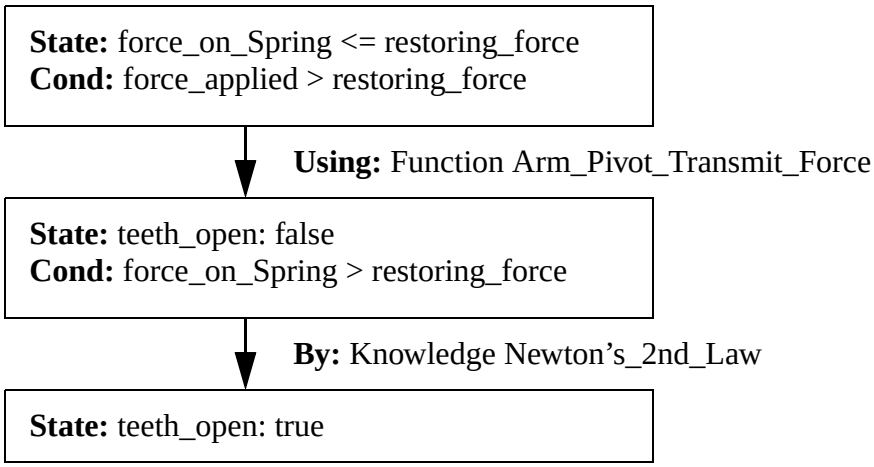


FIGURE 12. Open behavior of the clothespin

The *contexts for measuring designs* must be processes that can be applied to a design such as designing, manufacturing, using, repairing, maintaining and so on. Thus we may want to answer questions like "How complex is it to manufacture the clothespin?", or "How complex is it to use the clothespin?" and so on.

For each of the contexts above there may be *aspects of a design* with respect to which complexity may be measured. In our research we will only be concerned with complexity for the aspects of structure, behavior and function of designs. Note here that some of context-aspect combinations may not make sense. For instance it doesn't seem to make sense

to talk about functional complexity in the process of assembly. This is because the process of manufacturing doesn't refer at all to the function of the design.

For each context-aspect combination that makes sense, we can define several *measures*. We will consider measures based on counting different elements of the design. (In Section 2.1.4 we will explain why we believe that counting-based measures are the most appropriate for measuring the complexity of designs.) What exactly needs to be counted will depend on those elements of the design in the aspect considered, on which the process corresponding to the context depends.

Let us illustrate our ideas about measuring the complexity of designs using our clothespin example.

When *designing* the clothespin all the aspects (structure, behavior and function) can be considered for measuring complexity. Consider function first. To measure *functional complexity* of the clothespin when designing it we may take into account some or all of the following elements:

- the *complexity of the environment*, that is how many elements in the environment the design must interact with and how complex those elements are. (e.g., in order for the

clothespin to achieve its function, it must be placed into an environment where mechanical forces can be applied to its arms),

- the *complexity of the interaction with the environmen*t, that is the number and complexity of inputs and outputs that have to be applied to the design to achieve its function (e.g., in order for the clothespin to achieve its function, two linear forces of opposite directions must be applied to its arms and one output, the opening of the teeth, will be generated: the number of inputs is 2 and the complexity of both inputs is that of a linear movement),
- the *complexity of deployment*, that is how complex the decomposition of the deployment into behaviors and functions is (e.g., the `Open` function of the of the clothespin is deployed by the behavior `Open_Behavior`, which is implemented, decomposed into the function `Arm_Pivot_Transmit_Force` and the physical principle `Newton's_2nd_Law`).

Let us note here that other designs may have more than one intended function, in which case another way to look at functional complexity is to count the different functions the design is intended to achieve.

We can conclude that when measuring functional complexity we can count (or measure the complexity of) objects in the environment, inputs and outputs, functions and behaviors and possible relations between these (e.g., synchronization relations between inputs: the two forces have to be applied at the same time to the arms of the clothespin).

The statement above seems to be circular because it explains the measuring of complexity in terms of itself. However, it is meant to be interpreted in a recursive way, rather than circular one. By this we mean that the complexity of a function may depend on the complexity of other elements it is in relation with. For instance, we stated above that the complexity of the interaction of the clothespin with its environment depends both on the number of inputs and on the complexity of those inputs. The latter requires that there is a way to measure the complexity of inputs. In this case the inputs are forces that are applied linearly. We view such a linear input as simpler (less complex) than a *rotational* one, that is one applied by winding, rotating, etc. Obviously this recursive way of defining complexity will only be correct if there is a well defined set of elementary objects, behaviors, functions, inputs and outputs for which the complexity is postulated. In the rest of this subsection we will use similar recursive definitions for the behavioral and structural complexity of designs.

To measure the behavioral complexity of the clothespin in the context of designing, we may take into account the following:

- the *complexity of implementation* of the behavior, that is, how the behavior is decomposed into other behaviors and functions (e.g., the decomposition of the `Open_Behavior` of the clothespin),
- the *complexity of the behavior process*, that is how many steps (state transitions) the behavior consists of and how complex those steps are, in terms of the complexity of the partial states and conditions involved in the description of those steps (e.g., the Open_Behavior of the clothespin is described as a process consisting of two state transitions and the initial state of the first transition is specified as a value constraint of a single attribute of the clothespin - Figure 11 (b)).

Thus, when measuring the behavioral complexity of a design we may count (or measure the complexity) of functions, behaviors, states and conditions (i.e., of processes).

Consider finally the *structural complexity* of the clothespin when it is being designed. The elements that contribute to this aspect are:

- the *attribute complexity* of the design (or of its components), that is the number and complexity of attributes (e.g., both the arms of the clothespin have one attribute,

length, that can be expressed by a single numerical value, while the spring has one attribute, the restoring force, which can be represented as a formula that depends on several values),

- the *compositional complexity*, that is, the number and complexity of the components, subcomponents, sub-subcomponents and so on into which the design is decomposed (this decomposition of the clothespin is illustrated in Figure 11 (b)),
- the *relational complexity*, that is the number and complexity of the relations between the components of the design. These relations for the clothespin are illustrated in Figure 11 (b) by the thick, two-way arrows.

To measure the structural complexity of a design we may count or measure the complexity of attributes, component objects and relations.

When *manufacturing* the clothespin all its components need to be manufactured and then they have to be assembled. It should be clear that in this context measuring the functional or behavioral complexity of the design does not make sense. To measure the structural complexity of the clothespin, with respect to manufacturing, we may take into account some or all of the following:

- the *compositional complexity,* that is, the number of components that have to be manufactured and the (structural) complexity of those components in the context of manufacturing (e.g., for the clothespin six components have to be manufactured, of which the spring may require a more complex manufacturing process),
- the *relational complexity,* that is the number and complexity of relations between the components which have to be physically realized (e.g., for the clothespin relations between components are realized by gluing, welding, assembling, and each process may have different complexities).

We can conclude that for measuring the structural complexity of a design in the context of manufacturing we may count (or measure the complexity) objects, relations and processes.

2.1.4 Why Count when Measuring Complexity?

We will conclude this section on measuring complexity by explaining why we believe that *counting is the appropriate way to measure complexity* in general. Our motivation is similar to the one given for the analysis of algorithms: measuring complexity by counting allows the estimation of resource requirement in terms of elements that only depend on the intrinsic characteristics of the design. Such measures can be used further as the basis for estimating costs in different, concrete environments (see for instance [Bashir & Thomson

1999]).

To illustrate this let us consider once again the problem of measuring the complexity of our clothespin in the context of manufacturing. To manufacture the clothespin according to the design description in Figure 11 one will have to manufacture two levers, two teeth, a pivot and a spring, weld the teeth onto the levers, assemble the two arms and the pivot and glue the spring to the two arms. Counting these steps (of manufacturing and assembly) gives an exact measure of what needs to be done. We can then use this measure of complexity to estimate the cost of manufacturing the clothespin by combining it with measures for the technological sophistication of the manufacturer, technical difficulty of the processes in the manufacturing environment, experience and skill of the workers and so on. Similar arguments can be made for measuring complexity in other contexts ([Bashir & Thomson 1999 express similar ideas)

This two-step approach to estimating the cost of a design would have the following benefits:

- it provides an objective measure of complexity, depending only on intrinsic aspects of the design;

- it provides an absolute basis for comparing the resource requirements of different designs;
- it provides a common basis for computing the costs of a design in different environments;
- it provides a better ground for certain kinds of decisions concerning designs (e.g., whether it is feasible, in what environment is it less expensive to realize, and so on).

2.1.5 The Simplification Process

By *simplification* of an object we mean a transformation of the object into a different object, such that the complexity of the result is lower than the complexity of original.

The *complexity* of an object can be evaluated by computing (or physically testing) it in some *context*, relative to some *aspect* and according to some *measure*. *Contexts* refer to processes that can be applied to the object considered. In the case of design simplification, contexts include assembly, manufacturing, use, aesthetics, etc. *Aspects* refer to different points of view on the object. For instance, in the case of design simplification, aspects include structure, behavior and function. Corresponding to this we can speak about structural, behavioral and functional simplification. *Measures* involve counting, or some measure of complexity or information content. In our research we argue for the primary use of

counting, however other complexity measures have also proven useful (see for instance [Suh 1990]). Lower counts, complexity or information content imply ‘simpler’. What is counted depends on the context: for example, it may be assembly operations, components, surfaces, the potential for manufacturing mistakes, internal states, inputs, or the number of terms in an equation that describes a surface.

There is a tendency for people to *assume* that certain types of changes are simplifications. An example of this is the belief that ‘simpler = fewer components’. Such examples have been compiled into peoples’ knowledge by repeated use, and appear to be context-free to those individuals. However, we feel that they can be traced back to contexts such as manufacturing or use. We suspect that the common ‘assumed’ simplifications are those that lead to being evaluated as simpler in common contexts and measures, or that lead to being evaluated as simpler in the important majority of contexts and measures.

One primary goal of our research is to give an operational definition of design simplification. For this purpose we will consider a fixed set of the most important contexts in which complexity of designs may be measured. We expect that our definitions can then be extended to other contexts. We select only a fixed number of contexts for pragmatic reasons. The only aspects of designs for which we will study simplification will be structure, behavior and function. There are two major reasons for this. On one hand these are the

most important aspects of designs considered in the process of designing (our main area of research). On the other hand these three aspects address the problems of simplifying structured objects, relations and processes. We believe that any design simplification is a combination of these three *types of simplification* (i.e., structure simplification, relation simplification and process simplification).

2.1.6 Propagation of Simplification

Simplification of an object for a given combination of context, aspect and measure will need to propagate to the other aspects, as different aspects of an object may be interdependent. For example, removing redundant links in a causal chain of motion or force flow might cause two gears in a three gear train to be removed, i.e. a behavioral simplification propagates to the structural aspect. Unfortunately it is not always the case that simplification in one aspect will propagate to a simplification in all the other aspects. For instance, if a design is simplified in the context of its use by making its interaction with the environment less complex (e.g., less inputs, or less complex interaction process), the structure may need to be modified by adding new components to it, thus making it more complex.

In our current research we are not addressing to any depth the problem of propagation of simplification across aspects. We merely acknowledge its importance and illustrate the

problems it raises by some examples.

2.2 Performing Simplification

Once we have defined how to evaluate whether a transformation of an object is a simplification or not, we need to study what kind of transformations can be used to produce simplifications, and compare them for effectiveness and efficiency. We will start our discussion on this topic by presenting possible ways a simplification process could be achieved.

We must view the *simplification process as a search* in some search space (e.g., design space in the case of design simplification). The goal of the search is to find a simpler object than the one given as starting point. Note that we do not define simplification as an optimization problem (i.e., with the goal to find the least complex object), but rather an improvement problem. Also, simplification is a constrained search because all simplification problems require the preservation of some properties of the object (for instance, design simplification is, or should be, a function-preserving process).

2.2.1 Possible Approaches

One possible approach to performing simplification is to *view the simplification problem as an optimization problem* with a complexity measure as the objective function. For instance one could apply local transformations known to reduce complexity and organize them into a hill-climbing type process. Structural simplification of a mechanical design could be approached by applying simple simplification operators, such as removing redundancy (e.g., removing two gears from a chain).

While we do not view simplification as a (global) optimization, an approach of this kind would have all the draw-backs of global optimization methods. The simplification can quickly "get stuck" at a point where no more improvements can be obtained (local optima). Although we can say that even in this case some simplification has been achieved, in general the "big picture" will be missed. For instance removing two small gears from a very complex device may have very little or possibly no impact on its complexity. Those two gears may however be parts of a more complex context in which some higher level, more conceptual simplification could have been performed.

Having some knowledge of what operations and what sequences of operations may lead the search towards "good" simplifications, would overcome the deficiencies of the local

optimization approach. Such heuristic knowledge would in general avoid local optima. Also, and more importantly, the *heuristic search* approach to simplification will allow for simplification processes that temporarily create more complex objects with the purpose of setting the context for a more significant (higher level?) simplification (e.g., adding additional, but redundant structural components may trigger a simplification where the whole object can be made from a single molding).

One general problem of this approach is the lack of good heuristics. As far as we know there are no general (domain independent) heuristics for simplification. In some domains there are certain principles of what kinds of transformations lead to simplifications. For instance there are some good heuristics on how to perform simplification on arithmetical expressions. Some other domains, such as design for instance, have very few or no such heuristics (e.g., principles for DFM [Stoll 1991]). For such domains the heuristic search approach for simplification is not appropriate.

We believe that *reusing known simplifications* to produce new ones is the best approach to the simplification problem. First, a known simplification can be reused over and over for identical simplification problems. Second, even if a new simplification problem is not identical to any known one, if some (significant) similarity between the two can be discovered, the old simplification may be used as an “idea” for simplification. These two situa-

tions suggest the use of *case based reasoning* as a possibly good approach to simplification. Finally, reusing known simplifications can also be done across domains. This assumes discovering some abstract similarities between a given simplification problem in a domain (target) and a known simplification in some other domain (source), and using that similarity to transfer the "simplification idea" to the target domain. This suggests that *analogical reasoning* could be used as a good approach for performing simplifications, especially in domains where simplification is not a well understood problem. Simplification by analogical reasoning also has the benefit that it may be capable of producing general simplification principles by learning and abstracting over the simplifications produced.

In our research we propose the study of using analogical reasoning for simplification in general, and design simplification, in particular.

2.2.2 Simplification by Analogical Reasoning

In this section we give a brief description of what analogical reasoning is. In this we follow the definitions in Bhatta & Goel [1994].

Analogical reasoning is the process of retrieving knowledge of a familiar problem or situation (called the source analog) that is similar to the current problem or situation (called

the target) and transferring that knowledge to solve the current problem.

Analogies can be of different types: within-problem, within-domain and cross-domain analogies. Within-problem analogies involve the transfer of knowledge from one subproblem to another subproblem within the context of solving the same overall problem. Within-domain analogy involves the transfer of knowledge from one problem to another in the same domain. Cross-domain analogies involve the transfer of knowledge from a problem in a domain to another problem in a different domain.

Although several different models of analogical reasoning have been proposed, one can identify in most of them the following stages [Gentner 1983]: retrieval of source analog, mapping of the source analog to the target, transfer of relevant knowledge from the source to the target, evaluation of the solution to the target problem, generalization over the source and the target, and storage of the solution to the target problem, and of the generalization (Figure 13).

Bhatta and Goel [1994] list a set of important issues raised when applying analogical reasoning to problem solving:

- What should be the content and representation of source analogs?
- How is the target problem specified?

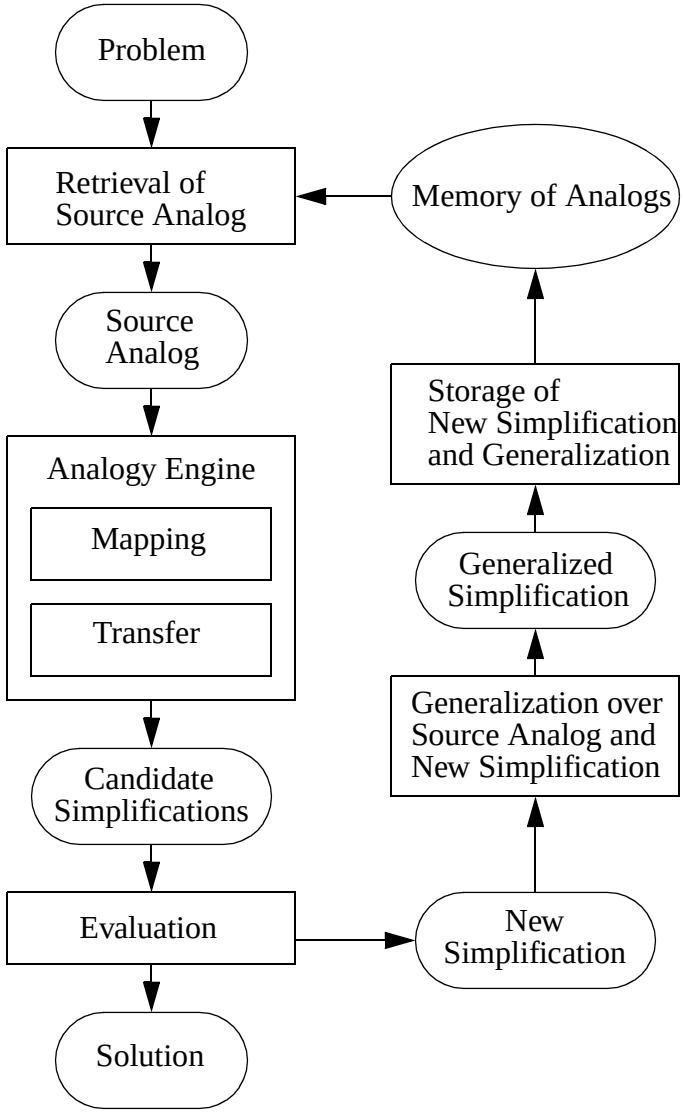


FIGURE 13. A process model of analogical problem solving

- Given a target problem, how might the retrieval of the source analog occur? This question gives rise to several sub-questions: What features (superficial or deep) of the target problem will determine the retrieval? How will it be determined whether source analogs will be retrieved from the same problem, the same domain or some different domain?
- Once a source analog has been retrieved, how can it be mapped onto the target problem and how will this mapping be used to transfer the problem solving knowledge?
- Since the transfer of knowledge from the source to the target may not satisfy the requirements of the target problem completely, how can the solution to the target problem be completed?
- How will a solution to the target problem be evaluated?
- How can it be decided whether a useful generalization over the source problem and the target problem can be built. How can such a generalization be built?
- How can it be decided whether the target problem and its solution are different (novel) enough to be worth storing for later use?
- How can the generalization and/or the target problem be stored into the database of problems for later use?

Most of the models of analogical reasoning are based on one of the following two computational frameworks: transformational analogy and derivational analogy. The two computational models are distinguished from each other by the way the knowledge transfer is performed. Transformational analogy involves the transfer by direct mapping of the source problem's solution to the target problem. Derivational analogy, on the other hand, involves taking the problem solving process of solving the source problem and replaying it in the target domain.

In the previous section we argued for approaching the problem of simplification by using analogical reasoning. To do so we will have to answer to the questions above in the context of the simplification problem. This will be done in the next chapter.

2.3 Difficulties Raised

In this section we describe how the issues raised by applying analogical reasoning in general translate to the application of analogical reasoning to simplification.

2.3.1 Retrieving Useful Simplification Examples

A simplification consists of a relation connecting two objects, a simpler one and a more complex one (called the *simpler relation*) and an *explanation* of the simplification (that is

of how the simpler object can be obtained from the more complex one). The explanation may be given as a description of the simplification process, or simply as a description of the difference between the two objects. The latter case may occur when the process of simplification is not known (e.g., the simpler relation was discovered by evaluating the complexity of the two objects, but there is no evidence of a simplification process by which one of the objects was transformed into the other). Thus to represent a simplification we need to represent a relation, a process and possibly a set of differences between objects.

As discussed earlier simplification can be performed from different points of view (that is with respect to different combinations of context, aspect and measure). One primary organization of simplification has to be made along these dimensions because simplification problems are specified with respect to some point of view.

An important problem is organizing simplifications for fast retrieval. Theoretically the most appropriate way to do this would be by building a hierarchy of simplifications and using it for fast indexing. The problem with this approach is that since at this point there are no general principles of simplification and no classifications of simplifications into types, building such a hierarchy would be either impossible or would result in very shallow hierarchies. As, hopefully, the system will produce new simplifications and generali-

zations over them, the building of such a hierarchy and of indexing schemes based on it will become possible.

An alternative solution would be to organize simplifications around the objects involved. Since a new simplification problem will be checked against the more complex object in the source simplifications, it would seem appropriate to organize known simplifications around hierarchies of these objects.

There are at least two problems with this approach. On one hand, while a simplification refers to two objects, the actual simplification process may only involve a small portion of those objects (for instance replacing three gears by two gears in a complex device will only affect the set of gears). Thus building a design hierarchy based on the *entire* objects involved in the simplification may not be useful and will definitely be unnecessarily complicated. On the other hand there may be several independent simplifications connecting the same two objects. This might require that the same object occurs in several different places in the hierarchy.

We are proposing the organization of simplifications around those portions of the objects involved which are relevant to those simplifications. This approach poses other problems: we need to define how is it decided what is relevant to a given simplification, when this

decision will be computed and how these relevant portions will be used to organize simplifications. Rather than computing which portions of an object involved in a simplification are relevant, we will compute the portions which are *not absolutely irrelevant*. A portion of an object involved in a simplification is 'not absolutely irrelevant' to that simplification if it is referred to in the simplification process. We call the process of deciding which portions of an object involved in a simplification are not absolutely irrelevant to that simplification *relevance calculation*.

Fortunately relevance calculation is quite straight-forward in this case. It assumes the collection of all the elements (e.g., components, relations, attributes) of an object involved in a simplification, which are directly or indirectly (i.e., through a function or relation) mentioned in the explanation of that simplification.

We acknowledge here that if, as a result of generalizations, abstract simplifications (that is simplification principles or rules) will be added to the database of simplifications, the relevance calculation for those simplifications may require a more sophisticated mechanism.

It is commonly accepted that, in analogical reasoning, higher level relations, or systems of relations are more useful for solving problems. However retrieving source analogs has to be a fast process and as a consequence it must rely on simple, surface aspects of the

objects inspected (such as attributes). Such surface similarities are in most of the cases not very useful and are sometimes even misleading. Even more, when performing cross-domain analogical reasoning the domains involve may not even share common attributes. Consequently we are faced with two apparently conflicting requirements: fast retrieval of simplifications, based on simple criteria, and retrieval of useful simplifications.

We are proposing to use for this our hierarchy of relevant portions of objects described in the previous section. This approach will reduce the search for source analogs to only relevant portions of objects involved in some simplification.

We must note here that, even with this two-level organization of simplifications for retrieving source analogs, the process of retrieving may be quite complex. This will happen if either there is a great number of known simplifications, or if the object to be simplified is complex. The second part of this statement is true because the relevant portions of an object involved in a simplification may be similar to different portions of the target (Figure 14). This suggests that the retrieval process should be further improved, if possible by pruning as much as possible from the space of possible source analogs. We are proposing to use a counting scheme similar to the feature-vector described in [Gentner & Forbus 1991].

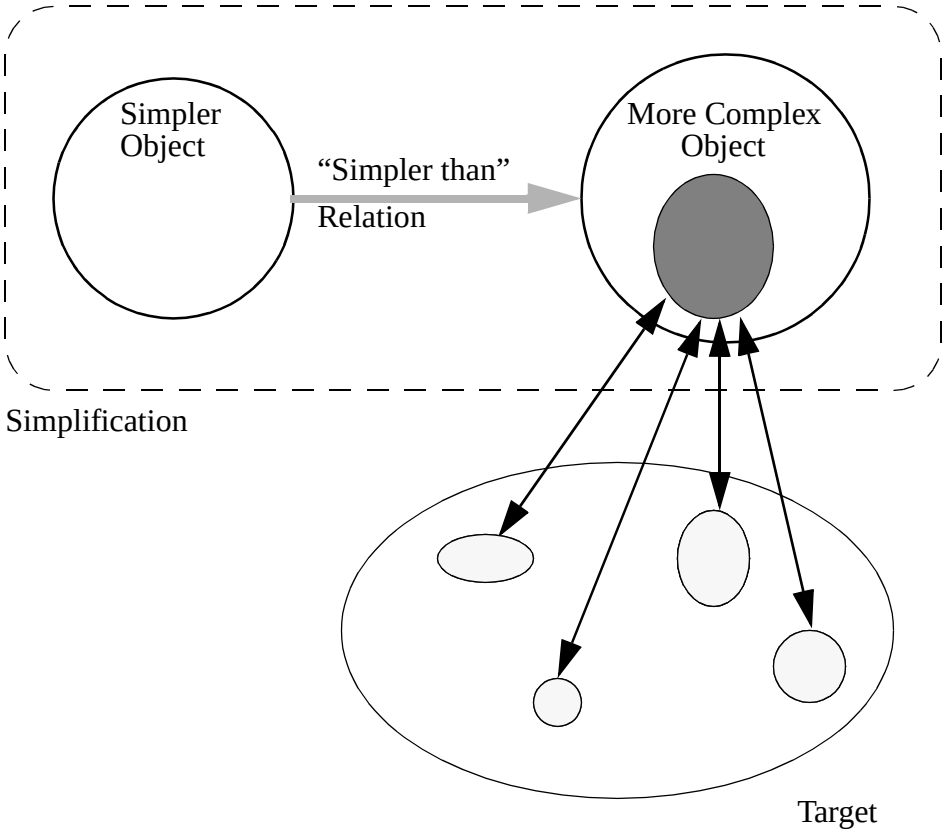


FIGURE 14. The relevant portion of an object may be similar to many portions of the target

The process of retrieving source analogs may produce several candidate source analogs, each of them having different degrees of similarity to the object specified in the simplification problem. This degree of similarity needs to be measured for each of the candidate source analogs retrieved. The resulting measures will then be used for selecting the best

candidate source analog to be considered in the next phases of the analogical reasoning. Defining a measure of similarity between a relevant portion of an object and another object (the one to be simplified), in the context of simplification has to take into account two main factors. On one hand the relevant portions which are parts of the retrieved candidate source analogs, may have different degrees of relevance, resulting from their role played in the corresponding simplification process. On the other hand, they may be similar to portions of the new problem with different degrees. For instance two relations with the same name (that is identical) are “more similar” that two relations that only have the same signature (that is, the same number and types of arguments, but different names).

2.3.2 Mapping Simplification Problems

Once a source analog has been retrieved, it has to be mapped onto the new simplification problem. Due to its clarity and efficiency from a computational point of view we will use an adaptation of Falkenheiner’s Structure Mapping Engine (SME) [Falkenheiner et al., 1986] for performing this mapping. The SME gets as its input a set of elementary *match hypotheses*. A match hypothesis is a pair of elements (i.e., objects, attributes and functions), one from the source analog and the other one from the target. Based on these elementary match hypotheses, the SME builds consistent *systems of mappings* (or simply

mappings) between attributes, functions and relations of the source and the target, working "upwards" in the hierarchy of relations in the source and target.

The SME builds these systems of mappings quite efficiently. However, as acknowledged by Forbus and Oblinger [1990] it has two significant draw-backs:

- it *constructs all* structurally consistent interpretations of an analogy,
- it *contains no mechanism for focusing on interpretations relevant to the goals* of the reasoner.

At this stage of our research we are primarily interested in using our problem solving goal, that is 'simplification', to guide the building of mappings. The way we have proposed to do the retrieval of candidate source analogs already focuses on portions of the objects involved in simplifications, which are relevant to our goal. The same relevant portions of objects can be used when building the mappings. We can restrict the building of mappings to portions of objects relevant to simplifications. This emphasizes the importance of the relevance calculation in our approach.

During the mapping phase, several mappings between the source analog and the target may be generated. Some of these mappings will be better than others. Transferring knowledge from the source analog to the target using a better mapping is more likely to produce

a solution to the target problem. To select the best mapping generated, a measure of mapping quality has to be defined. Such a measure needs to take into account both the quality of the component matches and the structure of the mapping. The quality of a match within a mapping expresses the level of confidence of placing the two members of the match in correspondence. The structure of the mapping refers to the systems of relations from the source analog and the target that are placed in correspondence by the mapping.

There are several possible ways to define a measure for mapping quality. Each of these definitions is essentially a computation process that, when applied to a mapping, will produce a measure of the mapping quality. These computation processes work by accumulating the measures of quality of the matches constituting the mapping along the structures connecting those matches. The measures can be classified into two classes, depending on how the processes defining them accumulate the measures of the matches: top-down or bottom-up.

2.3.3 Transferring Simplifications

Once a mapping is selected, it will be used to produce a solution to the new simplification problem. This is achieved by adapting the simplification corresponding to the mapping to the new problem. How this adaptation will be performed depends on how the simplifica-

tion is described, more precisely how the explanation of the simplification is given.

If the simplification is explained in terms the difference between the two objects involved in the simplification, then a similar difference must be built for the new problem. This difference can then be applied to the new problem. On the other hand, if the simplification is explained in terms of the simplification process, that process has to be adapted to the new problem by abstraction and instantiating, and then replayed in the context of the new problem. Thus, the way a simplification is given, naturally selects the type of analogy (transformational or derivational) to be used.

2.3.4 Evaluating the Result of the Simplification

After a new simplified object is produced it must be evaluated. This evaluation will necessarily refer to at least two aspects of the simplification. As simplification must preserve some properties of the object being simplified (e.g., function in the case of design simplification), the first thing that needs to be verified is whether the object generated as the solution to the simplification problem satisfies this constraint. Depending on how the constraint is specified, this part of the evaluation can be performed either by computation, or by simulation. For instance simulation may be used to verify that a simplified design satisfies the functional requirements imposed on the original one. The other aspect that

needs to be evaluated is whether the object generated as a solution to the simplification problem is indeed simpler than the original. This is done by measuring the complexity of both the object that had to be simplified and the one produced as the result of the simplification, and comparing the two measures. It may happen that the adaptation of simplification required some modifications of the object which rendered it more complex that the original one. If either of these evaluations fail, the result has to be discarded.

2.3.5 Storing new Simplifications

Any new simplification produced can be added to the database of simplifications. The question is whether it is new enough (different enough) from existing simplifications to be worth storing it. What may produce new simplifications are the adaptations of solutions obtained by analogical transfer. The more adaptation is needed, the more likely it is that the result will be different from the source simplification used.

CHAPTER 3

Related Work

We have no knowledge of any ongoing research in the area of "goal-based analogical reasoning for design simplification". However, there is certainly a rich body of research results in the relevant domains — analogical reasoning in general, and purpose-directed analogical reasoning in particular, abstraction, reasoning about designs and design optimization. This chapter relates our research to those domains.

3.1 Work on Analogical Reasoning

Our broader area of interest is *creativity* in *Artificial Intelligence* (AI) with emphasis on technical creativity in AI [Dasgupta 1994, 1996]. Many of the case studies presented in the literature revealed that creative acts had been results of some goal-driven activities. In the

case of technical creativity, quite often, the goal driven creative acts come in the form of improving, in general, and simplifying, in particular, some artifact or process. As a consequence we think of design simplification as a potentially creative activity.

Many researchers studying creativity agree that one of the most important ingredients of creative acts is using (new, surprising) analogies [Dasgupta 1994], [Perkins 1997], [Boden 1994], [Holyoak and Thagard 1995], [Finke et al. 1992]. To our knowledge most of the domains in which simplification is a common activity or a desirable goal, lack general simplification rules and principles. This is why we believe that a reasonable way to approach simplification problems is by reusing previously produced similar simplifications. These suggest the possibility of using analogical reasoning for solving simplification problems both as a way to reuse previously accumulated simplification knowledge, and potentially to generate creative solutions.

Using analogical reasoning for simplification would allow the reuse of simplification knowledge from the same problem (*within problem analogy*), the same domain (*within domain analogy*) or from a different domain (*cross-domain analogy*) [Bhatta & Goel 1994]. Simplification by analogical reasoning can be done either by transferring and adapting the result of a known simplification to the new simplification problem (*transformational analogy*), or by transferring, adapting and applying the simplification process to

the new simplification problem (*derivational analogy*). The generalization phase [Gentner 1988] of the analogical reasoning process may result in the generation of simplification rules and principles. The simplification goal can be used to guide the analogical reasoning and as a consequence improve its performance and/or the result it produces [Forbus & Oblinger 1994].

In the following subsections we will take a more detailed look at some work on analogical reasoning relevant to our research.

3.1.1 Model-Based Analogical Reasoning

Model-based analogical reasoning refers to using mental models of the underlying domain in the analogical reasoning process. Mental models [Gentner 1983] are characterized by the types of knowledge they capture, that is, structural, behavioral (causal) and functional knowledge of a physical situation or a physical device. The Structure Mapping Theory, used by many analogical reasoning approaches is based on this idea of mental models. By using mental models in analogical reasoning, many of the issues (raised by analogical reasoning) listed in Chapter 2 are answered through the definition of those models. For example, defining a mental model of a physical device or physical situation clearly specifies what may be the contents of source analogs, what their representations need to contain and

how these representations should be designed to allow the processing required by the analogical reasoning process, and so on.

Most of the work on analogical design [Qian & Gero 1992] [Bhatta & Goel 1994] [Goel 1997] relies on mental models of designs. These mental models of designs fall into two categories: *case-specific models* and *case-independent model*s (e.g., models of prototypical devices, physical principles, physical processes and generic mechanisms) [Bhatta et al.1994].

Probably the most popular case-specific models used in model-based analogical reasoning are the *structure-behavior-function* (SBF) models. This kind of model explicitly represents the structure of a design in some *object(-substance)-attribute-relation ontology,* representing its internal causal behavior as well as its function. Case-independent models used in model-based analogical reasoning are built such that they are compatible with the case-specific ones. For example, a case-independent model of designs used in conjunction with an SBF model could be defined in terms of behavior and function [Bhatta et al. 1994]. Such compatibility allows the “application” of case-independent situations (e.g., physical principles) to specific cases, as well the abstraction of specific cases to case-independent models.

Model-based analogical reasoning has been strongly criticized by part of research community (e.g., [Hoftsadter 1995]), due to the rigidity of predefined models for analogues. Forbus [Forbus et al. 1997] defends the SME approach by arguments referring both to psychological soundness and experiments.

We believe that model-based analogical reasoning indeed has strong limitations in its flexibility and, as a consequence, limitations on the possibility of creating certain kinds of distant (and interesting) analogies. This is due to the predefined structure of what can be inferred. We believe however that the kinds of analogies that a model-based approach would most likely fail to find fall into the category of ‘discovery’. The results produced by work on model-based analogical design show that (many of) the kinds of analogies needed for transferring design knowledge from one design case to another (within domain or across domains) can be produced using this approach. For this reason we consider that for our purposes the model-based approach to analogical reasoning is appropriate.

3.1.2 Goal-Driven Analogical Reasoning

Standard structure-mapping postulates that goals help determine both what gets matched and how the match gets evaluated [Gentner 1993]. This idea is incorporated in some of the research on analogical reasoning. Holyoak and Thagard [1989] use a blend of structural,

semantic and pragmatic consideration in their approach of finding a best mapping by constraint satisfaction. Forbus & Oblinger [1990] refer to another approach used by Falkenheiner in what he calls “contextual structure mapping”. The idea described there suggests the relaxation of the relation identicality and one-to-one constraints of structure mapping according to the goals of the analogical reasoning. In the same paper Forbus & Oblinger propose a new approach, called “pragmatic marking”, for incorporating the analogical reasoning goal into the operation of the SME. Their idea is to filter what subsets of local matches are considered, by whether or not they can support candidate inferences relevant to the analogizer’s stated goal.

Our approach is related to both Holyoak & Thagard’s and Forbus & Oblinger’s work. Similar to Forbus & Oblinger, we propose goal-based filtering. However, in our approach the filtering doesn’t only refer to local matches considered, but to designs and design parts based on relevance of their components, attributes and relations. Considering relevance as driving criteria for guiding the analogical reasoning process relates our work to that of Holyoak and Thagard’s. However, we discuss relevance in the context of design simplification problems. As a consequence we define precisely what is relevant to a design simplification and how relevance will be computed. We call our approach *goal-directed analogical reasoning*.

Goal-directed analogical reasoning is not to be mistaken for *purpose-directed analogical reasoning* [Kedar-Cabelli 1988]. The goal-directed analogical reasoning refers to using the problem-solving goal to guide the analogical reasoning process. On the other hand purpose-directed analogical reasoning, as used by Kedar-Cabelli refers to using the purpose of using (the function of) an artifact to guide the analogical reasoning about its structure.

3.2 Work on Abstraction

Abstraction is a very important ingredient of analogical reasoning, but it has developed into an area of research of its own, because of many other applications, such as hierarchical problem solving, planning or model-based reasoning. In this dissertation we are more interested in the role abstraction can play in analogical reasoning in general and in goal-directed analogical reasoning in particular.

In general we can think of abstraction as the “process which allows people to consider what is relevant and to forget a lot of irrelevant details which would get in the way of what they are trying to do” [Giunchiglia &.Walsh 1992].

There are two major problems raised by abstraction: *what to abstract from* (i.e., what to

forget) and *how to build an abstraction*. What to abstract from has to do with determining what is relevant or, equivalently, what is irrelevant with respect to the problem that is being solved.

Levy [1994] proposes a way to compute the "absolutely irrelevant"' elements of a set of queries. It essentially builds a set of elements that are referred to in the queries (relevant elements) and considers everything else absolutely irrelevant. The problem with this approach is that the so called "relevant elements" are only *syntactically relevant*, that is they may not be actually needed for solving the problem (e.g. they may be redundant). While this is not a problem for well formed representations such as query languages, for other domains, such as reasoning about designs, it may introduce limitations (e.g., some actually relevant elements, that are in relation with other relevant elements, but are not actually referred to, will not be considered). The approach we are proposing for defining (and computing) relevance to a simplification of designs also takes into account elements of designs that may be relevant due to some relations which connect them to other elements already determined as being relevant.

There are two general approaches to performing abstractions: a purely syntactic one and one based on a domain model.

The purely syntactic approach has a nice theory developed [Giunchiglia &.Walsh 1992] as well as several applications especially to theorem proving [Knoblock 1994] [Bacchus & Yang 1994]. The other general approach is based on the idea that what needs to be abstracted from has a semantic (and not just a syntactic) value. As a consequence, if the application domain is described in terms of a model, the abstraction needs to be applied to the model first to produce an abstracted domain model, and then the abstraction is built in this abstracted domain model [Levy 1994].

We are interested in using abstraction in the analogical reasoning process, in such a manner that what is abstracted from and how the abstraction is performed is determined by the problem solving goal, that is, simplification. To our knowledge there is no work on goal-driven abstraction in analogical reasoning. Since we use abstraction in the context of model-based analogical reasoning, we decided to use an approach similar to the one proposed by Levy. That is, we first perform abstraction on the model of a design, based on the relevance corresponding to a simplification the design is involved in and only then perform abstract reasoning over the abstracted model.

3.3 Work on Reasoning about Designs

Another relevant area to our research is reasoning about designs. Reasoning about how physical systems work has for some time been an important area of research in AI. Some of the major problems that arise in this domain are understanding how a particular physical system "works", diagnosing why a given system doesn't perform according to some expectations, and designing a physical system that would be usable for a certain purpose. From among these problems the last one is the most relevant to our research because simplification can be viewed as a *redesign problem*, in which the complexity reduction is added to the original design requirements. Within the (re)design problem, representing designs and reasoning about designs are central for us.

Although the literature shows some variety in approaching the problem of representing and reasoning about devices, it is usually discussed in the framework of reasoning about physical systems in general. As a consequence, the majority of these approaches refer to some or all of the notions of structure, behavior and function of an object and use some function-behavior-structure model to represent designed objects.

Our approach to representing designs follows the ideas concerning the structure-behavior-function representation scheme initially introduced by Sembugamoorthy & Chandraseka-

ran [1986] and further developed in work by Chandrasekaran [1994], Goel [1992] [1998], Bhatta [Bhatta et al. 1994] and others. In this approach, designs are represented by their structure, behavior and function and the relations between these. Structure is expressed in terms of components (sometimes substances as well [Bhatta et al. 1994]), attributes and relations between components. The behaviors of a device are represented as sequences of state transitions between behavioral states. Each state transition is characterized by the structural and causal context in which it can occur and the state variables that will be transformed. Finally, function is represented as a "top level behavior", and is characterized by an initial (input) state, a final (output) state and an internal causal behavior that "achieves" the function. Chandrasekaran and Josephson [1996] proposed an extension to representing function, by considering objects embedded in an environment. In their approach a function is defined in terms of the effects on its environment. This allows function to be specified without reference to the behavior it is implemented by. This idea of defining functions was used by Prabhakar & Goel [1998] to define an extension of the SBF model called Environmentally-bound Structure-Behavior-Function (ESBF). This new model allows reasoning about function without reference to the underlying behavior or, consequently to the structure.

Umeda & Tomiyama [1994] propose a slightly different way of modeling designed

objects. Their proposal is based on the observation that “function cannot be modeled objectively because functions are intuitive concepts depending on the intentions of designers and users” and as a consequence “it is difficult to distinguish clearly function from behavior and it is not meaningful to represent function independent of the behavior from which it is abstracted”. As a consequence they propose a two-level representation of designed objects consisting of two connected levels: the functional level and the behavioral level. Functions are represented in a knowledge base of function prototypes collected from existing designs. Behavior and state are represented in terms of Qualitative Physics Theory as a network of individuals, individual views and processes. A function-behavior relationship describes behaviors that can perform a given function in the form of views. A behavior-structure relationship describes the possible behaviors of an entity in terms of physical laws.

While at a first glance the FBS model used elements similar to the SBF model, it has a few disadvantages that make it unsuitable for our problem. For example, it does not support multiple levels of abstraction in describing behavior, and it does not allow reasoning from structure to behavior and from behavior to function.

There are several applications of functional reasoning about designs, such as conceptual design [Umeda & Tomiyama 1994], diagnosis [Chittaro 1995], blame assignment

[Stroulia & Goel 1995].

3.4 Work on Design Optimization/Simplification

Design simplification, and simplification (as a cognitive activity) in general, is a less researched area. Most designers and design researchers, including those we have spoken to about this topic, see simplification as reducing the number of components and/or connections between components. This view characterizes simplification as a process applied to the structural level of designs.

We find it reasonable to also think about the possibility of simplifying the behavior of a design (for instance, to involve less friction) or the function of a design (for instance, to need less input). Also, simplification of designs from the points of view of different life-cycle phases (manufacturing, maintenance, recycling and so on) has been extensively studied but not in the context of the general simplification problem. One of the most clear formulations of (what we may interpret as) design simplification principles are Stoll's [1991] design rules for efficient design for manufacture.

3.4.1 Suh's Information Content Reduction

The only general approach to design simplification we know of is Suh's [1990][1999]

"Reduction of the Information Content of a Product". This work gives a formal definition of the *information content* of a design, which we may interpret as a 'measure of complexity'.

The definition of information content of a design is based on the functional decomposition of the design and the following definition: "Information is the measure of knowledge required to satisfy a given FR (functional requirement) at a level of the FR hierarchy". The quality of a design is then formulated in terms of its information content (The Information Axiom): "The best design is a functionally uncoupled design that has the minimum information content". This together with the Independence Axiom, form the basis of the design principles formulated by Suh. We can interpret some of these principles as rules for reducing the complexity of designs, such as:

- Minimize the number of functional requirements;
- Integrate design features in a single physical part;
- Use symmetrical shapes and/or arrangements.

Suh's axiomatic way of measuring design quality, and within that context design complexity, is a nice theoretical approach. The problem with it is on one hand that it is not clear in what context the complexity or cost is being measured (most likely manufacturing). It also

has the draw-back of all universal measure approaches in that it does not focus on different aspects in response to the primary goal of the designer.

3.4.2 Bashire & Thomson's Estimation of Design Effort

In their work on methodologies for estimation of design projects Bashir and Thomson [1999a][1999b], suggest that the accuracy of estimating the time required by a design project depends on the accuracy of effort estimation. They refer to a general view according to which the effort required by a design project depends on three factors: the *size of the project* (the number of components), *the complexity of the task* (the relative difficulty of the task in a particular environment) and *the productivity* (the rate at which the task progresses). According to them, the problem with this view is that it is not clear how project size should be defined. Measures such as counting may not be appropriate because the reduction of the number of components of a device may increase the complexity of other parts.

They are proposing to measure the size of a project in terms of its functionality. The method they propose for computing the complexity of a design project is based on a canonical decomposition of the top level function of the design and summation of the number of subfunctions at each level weighted with the corresponding level number. The

design effort is then computed by combining the complexity of the design project with factors that influence the productivity (e.g., technical difficulty; experience, skill and attitude of the team members, and so on).

In conclusion, Bashir and Thomson, proposed a way of measuring the cost of designing which clearly distinguishes between the complexity of the design on one hand, and the productivity factors on the other.

Essentially they propose a way to measure complexity of a project in the context of designing it and from the aspect of its function. The measure used is based on summing up the number of subfunctions at each level weighted with the corresponding level number.

While the idea of measuring functional complexity is a very interesting idea, considering a set of canonical functional decompositions will render the (conceptual) design phase quite inflexible (routine). Also, the functional decomposition will have to rely on a set of elementary functions. Those elementary functions may in theory be implemented by different structural elements, the design of which will also have possibly significantly different complexities. This will not be reflected in the model proposed.

3.4.3 Boothroyd & Dewhurst's Complexity Factor

Bothroyd and Dewhurst [1991] developed a set of principles for reducing the manufacturing and assembly cost of a product. Their work is in the domain of Design for Manufacture (DFM) and Design for Assembly (DFA).

According to them DFM may mean different things to different people. For example, for individuals, whose task is to design of a single component, DFM may mean to avoid the use of features that are unnecessary expensive to produce, or it may mean minimizing material costs. The important point for us in their work is the distinction they make between *part (components) DFM* and *product DFM*. They suggest that the "key to product DFM" is product simplification through DFA. Part DFM is only the fine-tuning process undertaken once the final form has been decided upon".

For DFA they propose a quantitative method, known as the *Boothroyd and Dewhurst method*. This method consists of two steps: a) minimizing the part count, and b) estimating the handling and assembly costs for each part. Based on this method they define a measure of complexity of products, called *complexity factor*, computed as the cube root of the result of multiplying together the number of parts, the number of part types and the number of relations between parts.

In conclusion, Boothroyd and Dewhurst view simplification as the reduction of a complexity measure in the context of manufacturing. They argue based on a practical point of view that simplification is a major components of cost and that in fact simplification is the goal of product DFM, and reduction of cost is the result. As a general principle for simplification they propose the reduction of the number of parts.

3.4.4 Reasoning about Designs from different Points of View

Simplification at one level may cause modifications of the design at other levels. The study of causal reasoning about the effects of a simplification is a new area of research. To our knowledge, considering simplifications from different points of view is also a new research area. Some relevant work refers to reasoning about designs from different points of view.

Chittaro, Tasso and Toppano [Chittaro et al. 1994] introduced *multimodelling* in representing and reasoning about physical systems. The key idea in their work is that the task of reasoning about physical systems can be viewed as a cooperative activity, which exploits the contribution of several separate models of the system, each one encompassing only a specific type of knowledge. The different models of a physical system are interconnected.

The task of problem solving is based on two fundamental mechanisms: *reasoning inside*

the model and *reasoning across models*. The models they are proposing can represent structural knowledge (knowledge about the topology of the system), behavioral knowledge (knowledge about the potential behavior of the components), functional knowledge (roles components may play in the physical processes in which they take part), teleological knowledge (knowledge about the goals assigned to the system by its designer) and empirical knowledge (knowledge concerning the explicit representation of system properties). The different models are integrated by using associations between structure and behavior, and links between function and behavior implementing the function, as well as between function and teleology.

Another piece of work relevant to reasoning about designs from multiple points of view is Manfaat's [Manfaat et al. 1996], [Manfaat 1997] work on viewpoint-based abstraction. This work proposes building multiple models of a physical system by building abstractions from different points of view. These abstractions are interconnected through the model of the whole system. These interconnections are used for managing the interdependencies between the different models.

CHAPTER 4

The Approach: Simplification by Goal-Directed Analogical Reasoning

In this chapter we describe the approach we are proposing for solving simplification problems. It is based on using analogical reasoning to transfer knowledge from known simplifications to the new problem. Since the *goal* of the reasoning, (i.e., "to simplify"), is known it will be used to guide all the phases of the analogical reasoning process. We call this kind of analogical reasoning process *goal-directed analogical reasoning*. Note that this is different from the purpose directed analogy proposed by Kedar-Cabelli [1988]. There "purpose" does not refer to the purpose of the analogical reasoning, but rather to the purpose of the physical device that is being reasoned about. We will see later that in the application of our approach to the simplification of designs, this corresponds to using device function to guide analogical reasoning.

To illustrate our method we will use examples from the domain of mathematical expressions. The expressions we will be using consist of constants, variables and the four basic arithmetic operators. It is true that arithmetic expressions have well-formed structures, and that the same problems we will be addressing in this chapter will become much more complicated when applying our approach to designs. However, we believe that they are adequate to present the most important problems raised and the solutions we are proposing to them.

4.1 Simplification as a Problem Solving Goal

A simplification problem is defined by three elements: the *object* that has to be simplified, the *point of view* (context, aspect and measure) of the simplification and *properties* of the object that *the simplification has to preserve*. These three elements correspond respectively to the *object*, *goal* and *constraint* of the simplification problem.

Under certain circumstances the goal and the constraints of a simplification problem can be implicit. As an example consider the following problem:

Problem 1: Reduce the number of multiplications in the following expression:

$$E(u, v) = \frac{((uv)^2 - 16) + 2u}{u(1 + v)}$$

The object of this problem is the expression E. The goal is to simplify the expression E in the context of evaluation, with respect to the aspect of its structure, as measured by the number of multiplications. The constraint is that the result has to be equivalent to E (i.e., it has to represent the same set of numbers). Note that part of the goal (the context and the aspect) and the constraint are implicit.

In the rest of this chapter, for all the examples using arithmetic expressions, we will assume that the simplifications they are involved in, or the simplification problems of which object they are, all have the same point of view as the problem above (i.e., evaluation as context, structure as aspect and count of multiplications as measure).

We can conclude that a simplification problem can be specified by the followings:

- a specification of the *object* of the simplification.
- a specification of the *goal* of the simplification. This has to be given a point of view of the simplification, consisting of a *context*, an *aspect* and a *measure*.

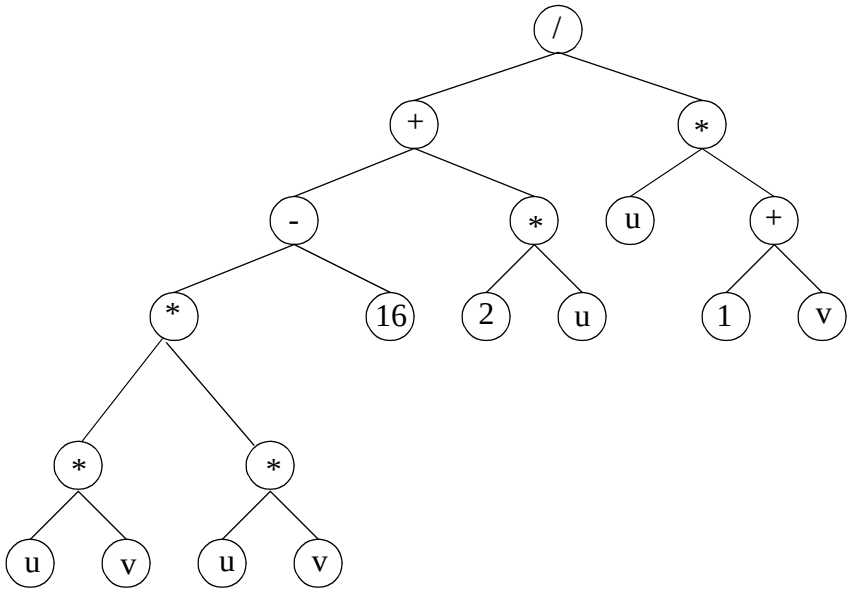


FIGURE 1. Structural representation of expression E(u,v)

- a specification on the *constraint* of the simplification. This can be given for instance as a set of logical propositions that has to be satisfied by the result of the simplification.

For our example the object of simplification can be specified as a representation of the structure of expression E (Figure 1.). Note, that for the sake of simplicity, in this chapter we will only use binary trees for structural representation of expressions. The point of view can be specified by the names of its components, that is, “evaluation”, “structure”

and “number of multiplications”. Finally, the constraint can be specified by the following condition:

If $F(a, b)$ is a solution of the problem, than

$$\forall(a, b)F(a, b) = E(a, b)$$

A simplification problem should be validated for consistency. This will involve at least the verification of whether the point of view specified as goal makes sense (see Section 2.1.3 for an example of a context, aspect, measure combination that does not make sense).

A simplification problem has to, explicitly or implicitly, specify how its solution(s) will be evaluated. Evaluating the result of a simplification problem means *verifying* whether the result satisfies *the simplification constraints* and *comparing* its *measure of complexity* to the corresponding measure of the original object. Evaluating the result of a simplification can be performed by either *evaluation* of formulae or by *simulation*, or by a combination of the two.

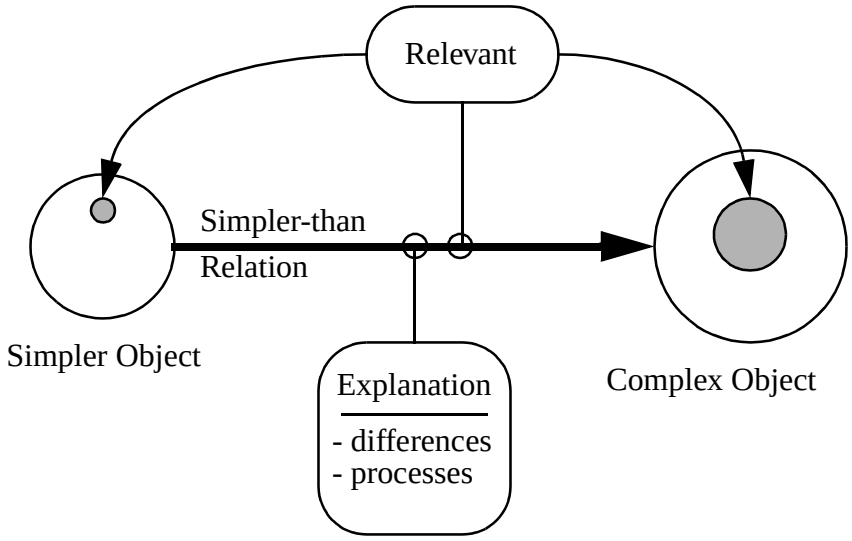


FIGURE 2. The structure of a simplification.

4.2 Representing Simplifications

To be able to perform analogical reasoning (or any kind of reasoning, for that matter) on simplifications we need to define how simplifications will be represented as well as how those representations will be organized to support the reasoning process.

We will represent a simplification by what we call a *"simpler-than relation"* (Figure 2). A simpler-then relation is a binary relation connecting two objects, a simpler one and a more

complicated one. A simpler relation may have two attributes: an *explanation* of the simplification it represents and a description of the aspects of the two objects that are *relevant to the simplification*.

4.2.1 Explaining a Simplification

The explanation of a simplification can be given in either of the following two ways: a) specifying the difference between the two objects involved in the simplification, or b) specifying the process by which the simplification was achieved.

Specifying the difference is needed when the fact that an object is simpler than another one was "discovered" (e.g., by comparing their complexity from some point of view), but no process for transforming is known. How the difference can be specified depends on the ontology used for representing the objects. For instance if the objects are represented using an objects, components, relations and attributes ontology, the difference can be represented by two sets: a set of elements (components, relations and attributes) that are part of the more complicated object, but not part of the simpler one (elements removed), and a set of elements that are part of the simpler object, but not of the more complicated one (elements added). In this dissertation we will assume that the differences are not very complicated and will represent both of these sets as lists.

As an example let us consider the following simplification of expressions:

Simplification 1:

$$\frac{1+xy}{x+10} \texttt{ is simpler than } \frac{1+xy}{x+2\cdot 5}$$

The simplification can be explained by "$2 \cdot 5$" being removed and "10" being added.

When the process by which the simplification was achieved (*simplification process*) is known, the description of this process can be added to the simpler relation as explanation. A *simplification process* will be represented as a sequence of transformations. Each transformation involves two objects (one being the object before the transformation, and the other the one after the transformation), the transformation operation applied, and the precondition which had to be satisfied in order for the operator to be applicable. To represent such a transformation we need the following:

- *a partial representation of the objects*, containing only those elements to which the operator refers and the ones involved in the precondition,
- a *representation of the operator* and the *arguments* it was applied to,
- a *representation of the preconditions* under which the operator was applied.

Let us consider the following simplification of expressions:

Simplification 2:

$$x-1 \text{ is simpler than } \frac{x(x-3)+2}{(x-2)}$$

The simplification process used in this simplification could be the following:

$\frac{x(x-3)+2}{(x-2)}$ is transformed into $\frac{x \cdot x-3x+2}{(x-2)}$

$\frac{x \cdot x-3x+2}{(x-2)}$ is transformed into $\frac{x \cdot x-x-2x+2}{(x-2)}$

$\frac{x \cdot x-x-2x+2}{(x-2)}$ is transformed into $\frac{x(x-1)-2(x-1)}{(x-2)}$

$\frac{x(x-1)-2(x-1)}{(x-2)}$ is transformed into $\frac{(x-1)(x-2)}{(x-2)}$

$\frac{(x-1)(x-2)}{(x-2)}$ is transformed into $x-1$.

The process explaining this simplification can be represented as a transformation graph, with the nodes representing intermediate forms of the expressions and the rounded boxes representing the transformations applied (see Figure 3). Note that not each transition in the simplification process has to be a simplification. For instance, the first transition produces a more complex expression. This sets the context for a simplification to be applicable.

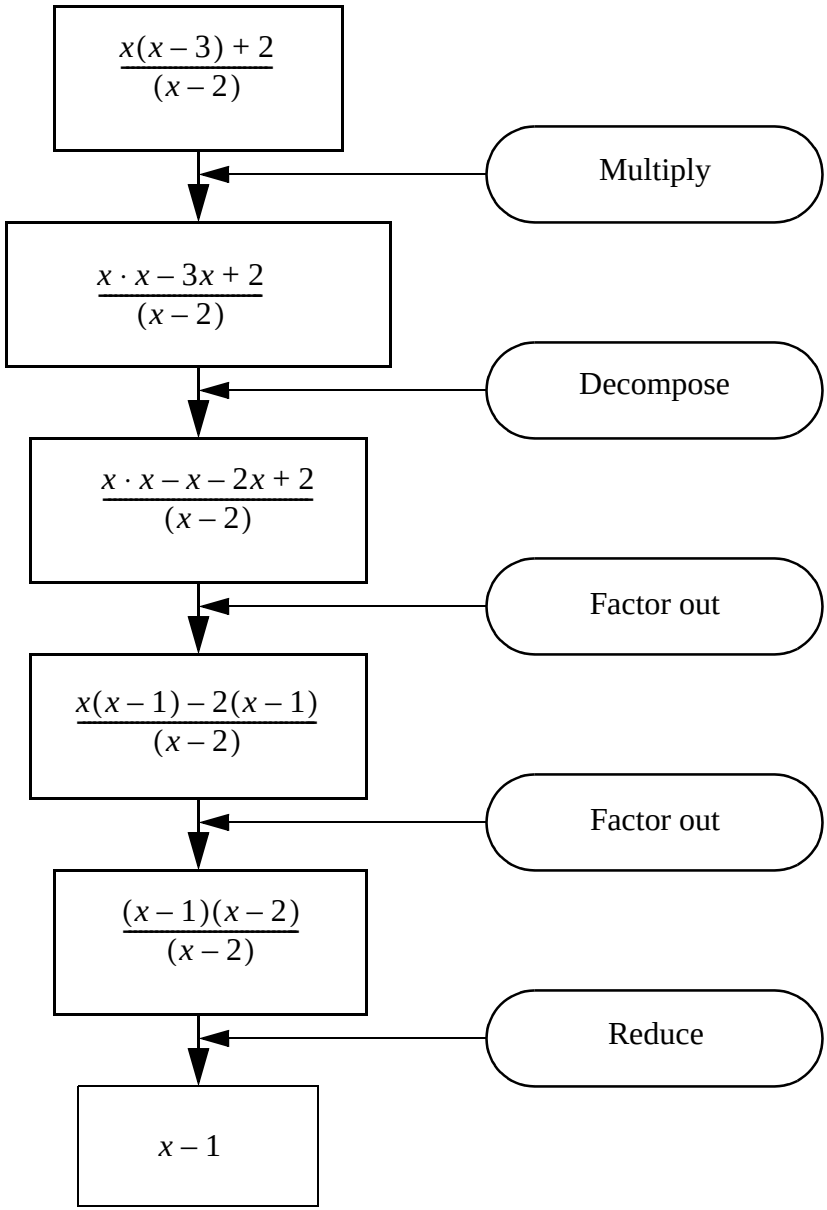


FIGURE 3. The transformation graph for the explanation of Simplification 2.

Note that since the process explaining a simplification *has* been performed, the preconditions are not absolutely required for representing the process. The reason we are including them into the representation is that they will play a role of rationale during the process of analogical transfer.

4.2.2 Elements Relevant to a Simplification

The second attribute (after explanation) that can be associated with a simplification, specifies the elements in the two objects involved, which are *relevant to the simplification* (we will call them *relevant elements*). Relevant elements are useful for two purposes. On one hand, they allow building *abstractions* of the objects involved in simplifications. These abstractions will not contain those portions of the objects that are irrelevant to the simplification. On the other hand, the relevant elements can be used as a basis for building indexing schemes over the set of objects involved in known simplifications.

The set of relevant elements corresponding to a given simplification can be computed automatically from the two objects involved in the explanation of the simplification and the explanation itself. In Section 2.3.1 we called the process of finding the relevant elements *relevance calculation*. Relevant elements of a simplification can be computed at the

time the simplification is created and can be used thereafter whenever needed.

As an example let us consider the following simplification:

Simplification 3:

$$\frac{3(x+y)-1}{(x+y)(x-y)} \texttt{ is simpler than } \frac{3(x+y)-1}{x^2-y^2}$$

The more complicated expression has 3 multiplications, while the simpler one only has 2. The explanation can be specified as a single step process consisting of the transformation: "replace $(x \cdot x - y \cdot y)$ by the (equivalent) expression $(x+y)(x-y)$ ". It should be clear that, while the simpler relation holds between the two expressions, only part of them is referred to in the explanation of the simplification. For each of the two expressions involved in the simplification, the parts (subexpressions) referred in the explanation constitute the relevant elements. Figure 4 presents a graphical representation of Simplification 3, with all its components. The shadowed portion of the structure represents the relevant elements of the two objects.

In Section 4.5 we will describe our proposal for performing relevance calculation.

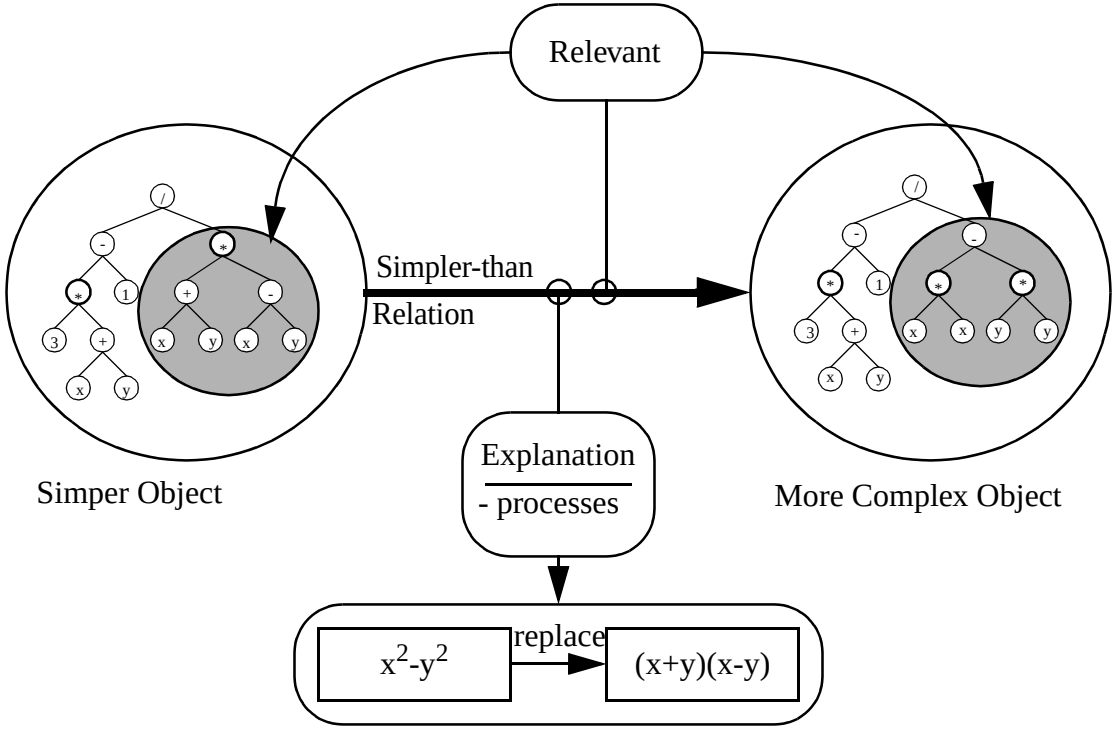


FIGURE 4. The representation of Simplification 3. The shadowed portions of the two objects involved in the simplification represent their relevant elements

4.3 Relevance Calculation

For every simplification added to the database of known simplifications the set of relevant elements will be computed. We called the process of computing the relevant elements of a

simplification relevance calculation. The relevance calculation can be decomposed into two phases: a) c*ollecting the elements that are not absolutely irrelevant* (with respect to the explanation) and b) *propagating relevance* along relations in the objects.

4.3.1 Collecting the Elements that are not Absolutely Irrelevant

Elements not absolutely irrelevant (see [Levy, 1994]) with respect to an explanation are elements that are explicitly mentioned in the explanation of the simplification. They may occur in the description of differences, in the case that the explanation is given in terms of differences, or in partial descriptions of objects, specifications of preconditions and arguments of operators, if the explanation is given as a process. These elements are said to be not absolutely irrelevant because, while mentioned in the explanation, they may not be absolutely needed. However there may not be any basis for discarding them as irrelevant.

The algorithm for computing the not absolutely irrelevant elements of an object involved in a simplification works by iteratively building a set of elements (objects, relations and attributes). The set is first initialized to the empty set. Next all the elements occurring in a difference or in a partial object description part of a transformation are added to the set. Finally, the relevance calculation is applied recursively to predicates, functions and objects in the preconditions and operators in the transformations. This algorithm builds a maximal

set of not absolutely irrelevant elements.

Figure 5 illustrates how relevance is computed for the more complex expression involved in Simplification 3. For better understanding the (functional) prefixed notation of the expression is used. The relevance calculation proceeds recursively: first, the top level operator (-) is added to the set of relevant element, and then the addition is repeated recursively for its arguments.

Note that only the elements explicitly present in the explanation will be added to the set of relevant elements.

4.3.2 Propagating Relevance inside Objects

In the previous section we described the first phase of the relevance calculation. We noted there that only the elements which were explicitly present in the explanation are collected in the set of relevant elements. However, the elements collected this way may be related to other elements in the object which were not explicitly present in the explanation, but which may bear some relevance to the simplification. This could happen for instance when the explanation of the simplification is given by the difference between the two objects

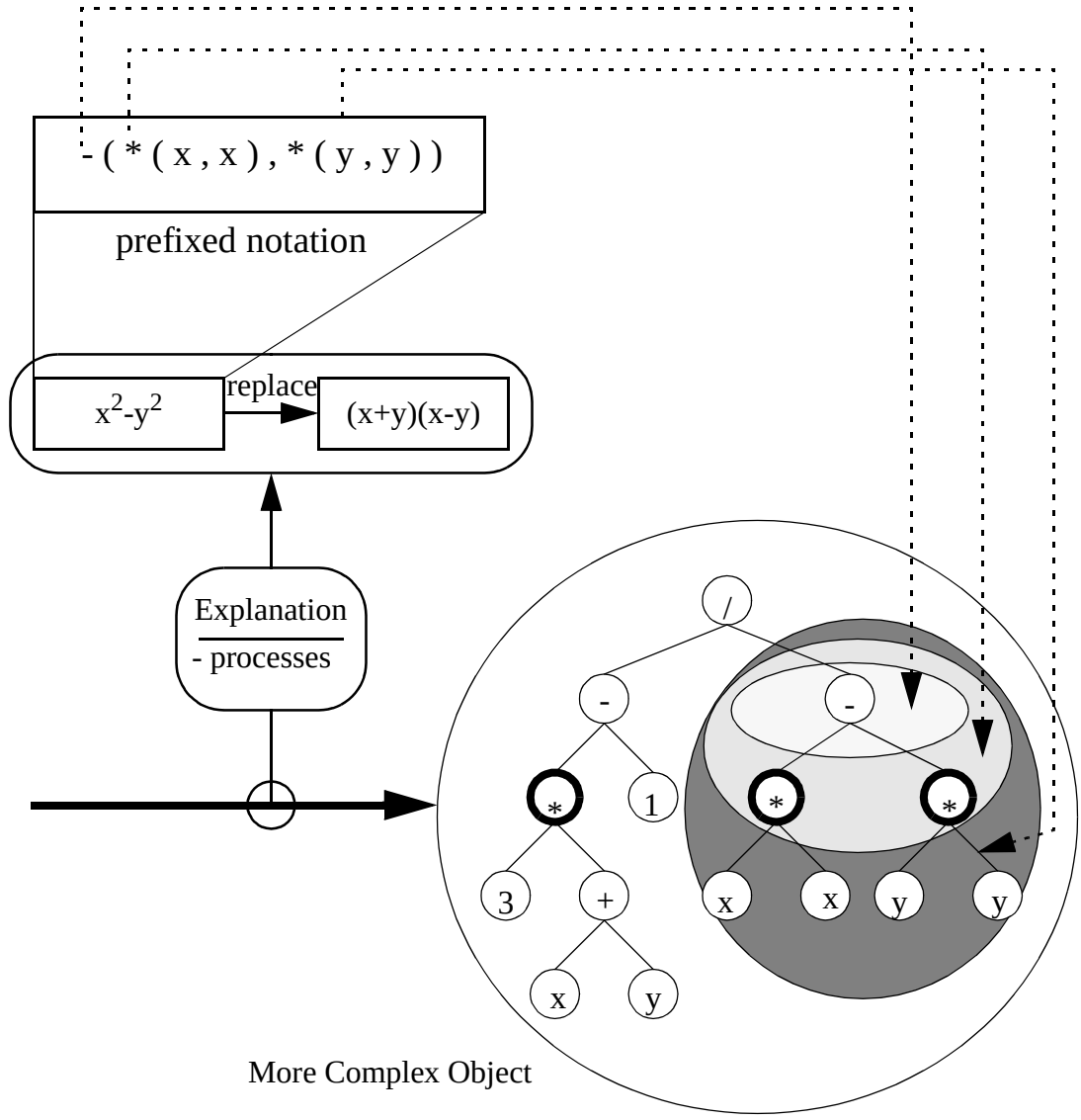


FIGURE 5. Computing the relevant elements of the more complex object involved in Simplification 3 the top level operator in the explanation is added first, than its arguments, recursively

involved. In this case only the removed and added elements are specified, without any reference to relations between them.

Consider for example the following simplification, very much similar to Simplification 3:

Simplification 4:

$$\frac{3(x+y)-1}{(A+B)(A-B)} \texttt{ is simpler than } \frac{3(x+y)-1}{A^2-B^2},$$

where $A = x - y$ and $B = 1 + xy$. The more complicated expression has expression has 4 multiplications, while the second one only has 3. As there is no simplification process specified as explanation, the only assumption that can be made is that an expression $A^2 - B^2$ has been replaced by the expression $(A+B)(A-B)$. Since x and y are not explicitly present in the explanation, they will not be added to the set of relevant elements. Figure 16 illustrates the result of the first phase of the relevance calculation by placing the elements collected into a dark filled shape. The question is whether the elements in the subexpressions involved in A and B, and possibly the operator '/' (of which the relevant portion of the expression is an argument) should be added to the set of relevant elements. In other words, should relevance be propagated *inside the object*? For the example considered, "downward propagation" - that is, propagating from relations to their arguments (or

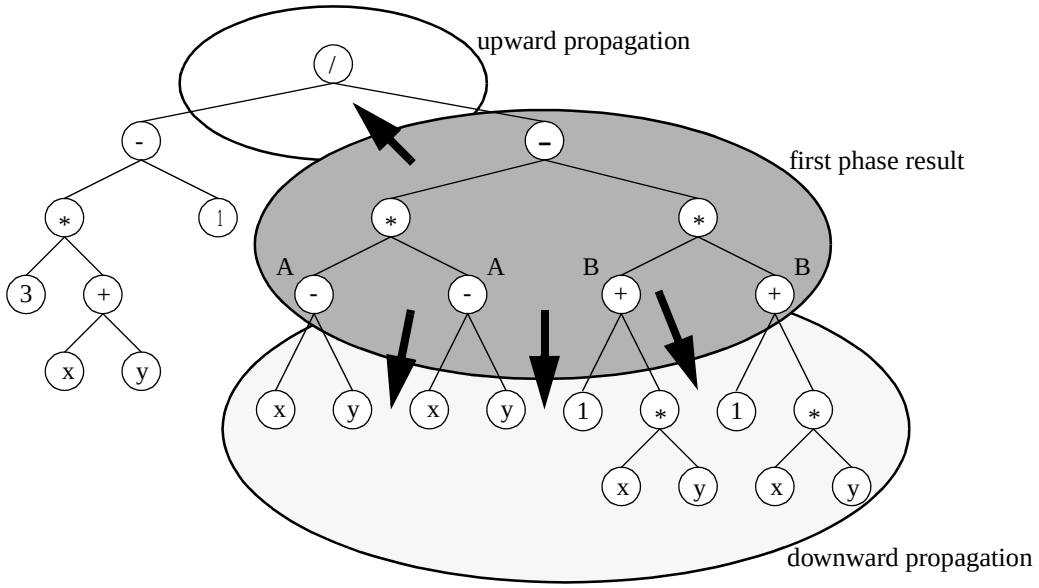


FIGURE 6. Propagation of relevance inside the more complex object involved in Simplification 4.

components to their subcomponents) - is desired, because it is important that subexpressions are identical. For this example it may not be so obvious whether the operator '/' should be added to the set of relevant elements. On one hand we could argue that it plays no role in the simplification, and thus it should not be added. On the other, however, the 'context' of the '/' operator sets some conditions on its arguments (e.g., the second argument cannot evaluate to 0) which suggests that it should be added to the set of relevant elements. Thus relevance may need to be propagated 'upwards' inside an object. We can conclude that relevance propagation is a knowledge-based activity. In Figure 6 the shapes

filled in lighter colors indicate the elements to which relevance can be propagated during the second step of relevance calculation.

We conclude from the above discussion that relevance may also need to be propagated inside the more complex objects involved in the simplification, along the decompositions, relations and attributes. This propagation can be done either downwards (i.e., from component to subcomponents, from relations to arguments, or components to attributes), or upwards (i.e., from subcomponents to their “parent” component, from components to relations they are arguments of, and from components to relations of which attributes they are).

This raises two further questions: a) Are both of the kinds of propagations needed? b) How far does relevance need to be propagated inside the object?

We propose that downward propagation be always performed. Upward propagation will only be performed if it is supported by several sources. For example, a relation not member of the set of relevant elements at the end of the first phase of propagation, will only be added to the set of relevant elements if all its arguments have already been added. In the next chapter we will describe propagation of relevance corresponding to our representation of designs.

Answering the question of how far relevance needs to be propagated is harder. There are two factors that should be taken into account when deciding whether or not relevance should be further propagated: the *length of the propagation* (because the farther we get from where the propagation inside the object started, the lower the relevance is likely to be), and the *strength of the connection* along which the propagation would happen (because, for instance propagating relevance from a relation to its arguments is more important than propagating it from an element to the relations in which it is involved). We will propagate relevance both downwards and upwards as far as it is possible.

4.4 Organizing Simplifications

Simplifications will have to be organized mainly for two phases of the analogical reasoning process: *retrieving* and (simplification) *knowledge transfer.* In the next two subsections we propose how these organizations should be done.

4.4.1 Organizing Simplifications for Retrieval

Simplifications have to be organized such that the resulting structure supports the retrieval of candidate source analogs. There are two important ways of approaching the problem of

the retrieval of candidate analogs: *indexing* and *spreading activation*. In our research we use indexing rather than spreading activation because the application of spreading activation requires the representation of knowledge by (complex) conceptual networks.

To build indexes over the set of simplifications we have to first decide what to use for indexing. Since the retrieval of a candidate analog can only be based on the elements defining a given simplification problem, we propose to build indexing around these elements. A simplification problem is defined by three elements: the *object* that has to be simplified, the *point of view* (context, aspect and measure) of the simplification and *constraint* (properties of the object that the simplification has to preserve). We are currently not considering the constraint part for organizing simplifications for retrieval. Therefore, we have two dimensions along which simplifications will be organized, and consequently two possible ways of indexing into the collection of known simplifications.

The first dimension along which simplifications can be organized for retrieval is their *points of view*, that is context, aspect and measure. While it is possible to exploit common features of simplifications with different point of views (for instance looking at simplification of structures, or simplification of processes), in this research we only use points of view for partitioning a given set of simplifications into classes of simplifications with the

same points of view. In the simplification of designs we partition the set of known simplifications into three classes corresponding to the structural, behavioral and functional aspects of the designs.

The other dimension along which simplifications can be organized for retrieval is the *relevant elements*. Relevant elements of a simplification are elements (e.g., components, relations, attributes, etc.) of the objects involved in the simplification, as obtained from the relevance calculation, based on the explanation of the simplification. For retrieving candidate source analogs we are only interested in the relevant elements of the more complex object involved in a simplification. The reason for this is that the object given to be simplified has to be matched with an object for which a simplification is known (that is, an object which is the more complex member in a known simplification).

Using relevant elements for indexing has the advantage of concentrating the retrieval process on elements that play some role in simplifications. What kinds of relevant elements of objects will be used for building indexes for retrieval will depend on the ontology used for representing the objects in the domain. For example, for organizing designs involved in structural simplification, if design structure is represented in terms of objects, components, relations and attributes, indexes can be build for some or all of the element types: objects,

relations and attributes. Objects could be organized by their classes into a hierarchy. Relations can be organized into hierarchies of relations and sub-relations (general to specific), or along features of relations such as signature (types of arguments), arity (number of arguments). Attributes can be organized into hierarchies of attributes. For example we may have numeric and non-numeric attributes. Numeric attributes may refer to sizes, intensities, etc.

Consider the example Simplification 3 shown in Figure 4. The more complex object involved in that simplification, which is the one used in the retrieval process, has only 7 'relevant' elements out of a total 15 elements in its representation. Only those 7 elements will be used to build the indexes for retrieval.

Arithmetic expressions are described in terms of *elementary expressions* (constants and variables) and *operators*, which combine expressions into other expressions. Thus, there are three types of elements for which indexes could be built. The set of constants and the set of variables cannot be decomposed into a hierarchy of classes. The set of operators, however can be decomposed into a (very simple) hierarchy of classes as shown in Figure 5. We acknowledge that, for this example, this indexing scheme is not very useful because there are too few classes in the hierarchy and, as a consequence, a possibly large number

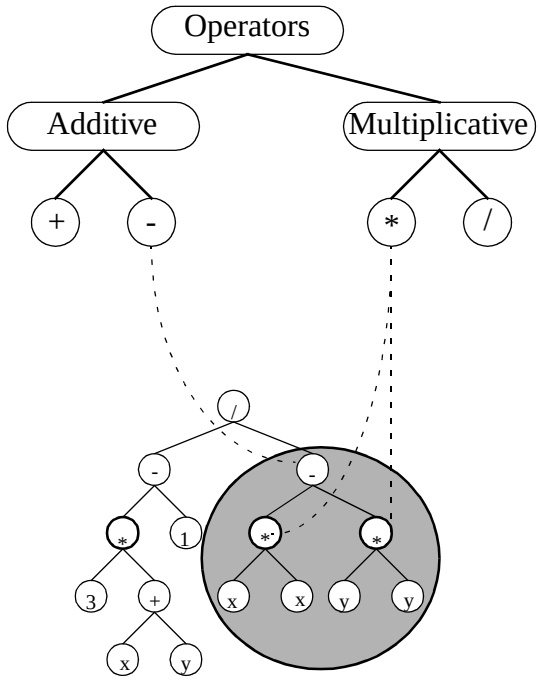


FIGURE 7. Indexing into the relevant aspects of expressions involved in a simplification, using a simple operator class hierarchy

of operators will go into every class. It is however useful to illustrate our proposal of building indexes over the types of elements which are parts of the relevant elements of a simplification

Note that, in general, in analogical reasoning certain kinds of features may be preferred over others for retrieving similar analogs. As a consequence retrieving is done based on either “surface similarities” (such as same attributes), or on “deep similarities” (such as

same relations), or on “literal similarity” (both deep and surface features).

Due to the nature of simplification, it is hard to prefer any of these over the others. First, it should be obvious that literal similarity is too strong because it will only find identical objects to be similar. On the other hand both deep and surface similarities may be important for a given simplification. Deep (relation based) similarities are important because in most of the cases simplifications can be performed because a certain system of relations hold. On the other hand a simplification may refer to only an attribute, or an attribute value. For instance a structural simplification of a cam mechanism may consist only in changing the shape of the cam to a simpler shape.

We propose that simplifications from a given point of view be organized for retrieval by the relevant elements of the objects involved, into several indexing hierarchies. These indexing hierarchies will be based on any of the types of relevant elements (e.g., objects, relations and attributes). If more than one such indexing scheme is used, different weights (levels of importance) may be given to them, as relations my lead to “more significant” simplifications than attributes.

As we shall see later, the proposed two-stage retrieval of candidate source analogs, using *point of view based pruning* first, and then *indexing into the relevant elements* of simplifi-

cations, may still be quite inefficient. In Section 4.4 we will propose additional operations for speeding up the retrieval.

4.4.2 Organizing Simplifications for Knowledge Transfer

Simplifications should also be organized into *abstraction hierarchies* based on their explanations. This, as we shall see later, will allow the analogical transfer of simplification knowledge by abstraction. Depending on how the explanation is specified (by differences or by simplification process), two kinds of abstraction hierarchies can be built: one along *object abstractions*, that is, along the classes of objects removed and added (as given in the specification of a difference), and another one along *process abstractions*.

For simplicity, we prefer to take a unified view of the two abstraction hierarchies. To do so we view the difference between two objects as representing a “replace” operator with preconditions that are not explicitly known. Figure 8 shows the representation of Simplification 1 where the explanation as difference was replaced by a one step process. Thus we view a difference as a single step simplification process.

Abstraction over simplification processes can be defined and build by using a “reduced

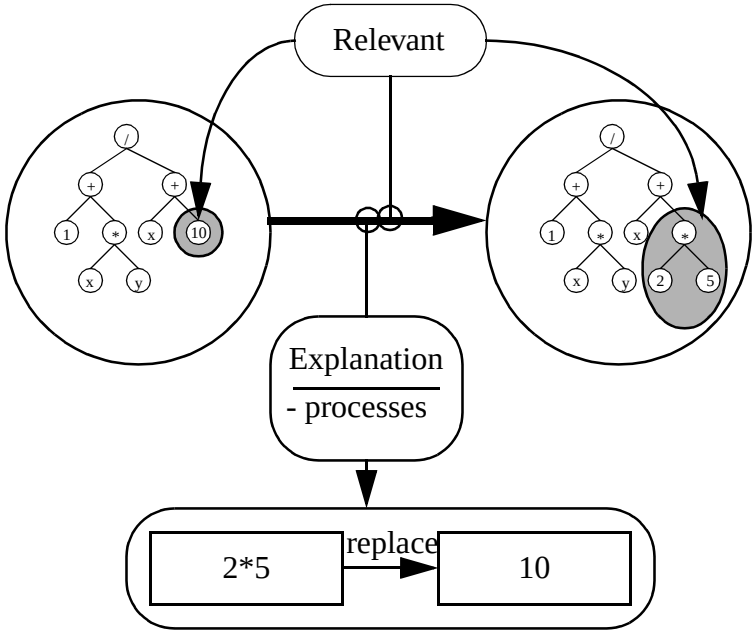


FIGURE 8. Representing the explanation by difference of Simplification 1 by a one step process

model" or a "relaxed model" as described in [Knoblock, 1994]. The essence of these abstracting techniques is to remove some elements from the states and/or preconditions involved in the process and replacing the transitions between states with abstract transitions obtained by dropping references to the removed elements.

To illustrate this by a very simple example, let us consider Simplification 2, for which the explanation was specified in the form of a process. The simplest abstraction that can be

applied to the process is to replace the constant '3' by a variable, say 'a'. This will result in a more abstract process, that will work for any value of 'a'. The problem with this abstraction is that there is more than one way to apply it. On one hand we could simply replace all the occurrences of '3' by 'a' without considering the context in which this is performed. The consequence will be that the result of the abstracted process will not be a simplification. This happens because the decomposing and factoring steps will not work for this abstraction (Figure 9(a)). On the other hand if, the abstraction is guided by the fact that '2' is less than '3' by '1' and that is why the decomposition and factoring work, the abstraction of '3' to 'a' should be accompanied by replacing '2' by '*a*-1'. The resulting abstraction of the process will work for any value of 'a' (that is it abstracts from what exactly number is the coefficient of the 'x-term' and it will produce a simplification (see Figure 9(b).

The above example shows that even simple abstractions can be performed in more than one ways. Performing formal abstraction, based only on the syntax of the objects abstracted, may lead to abstractions that are not useful for the problem for which the abstraction is being used.

In conclusion, we propose that the simplification relations be organized into abstraction

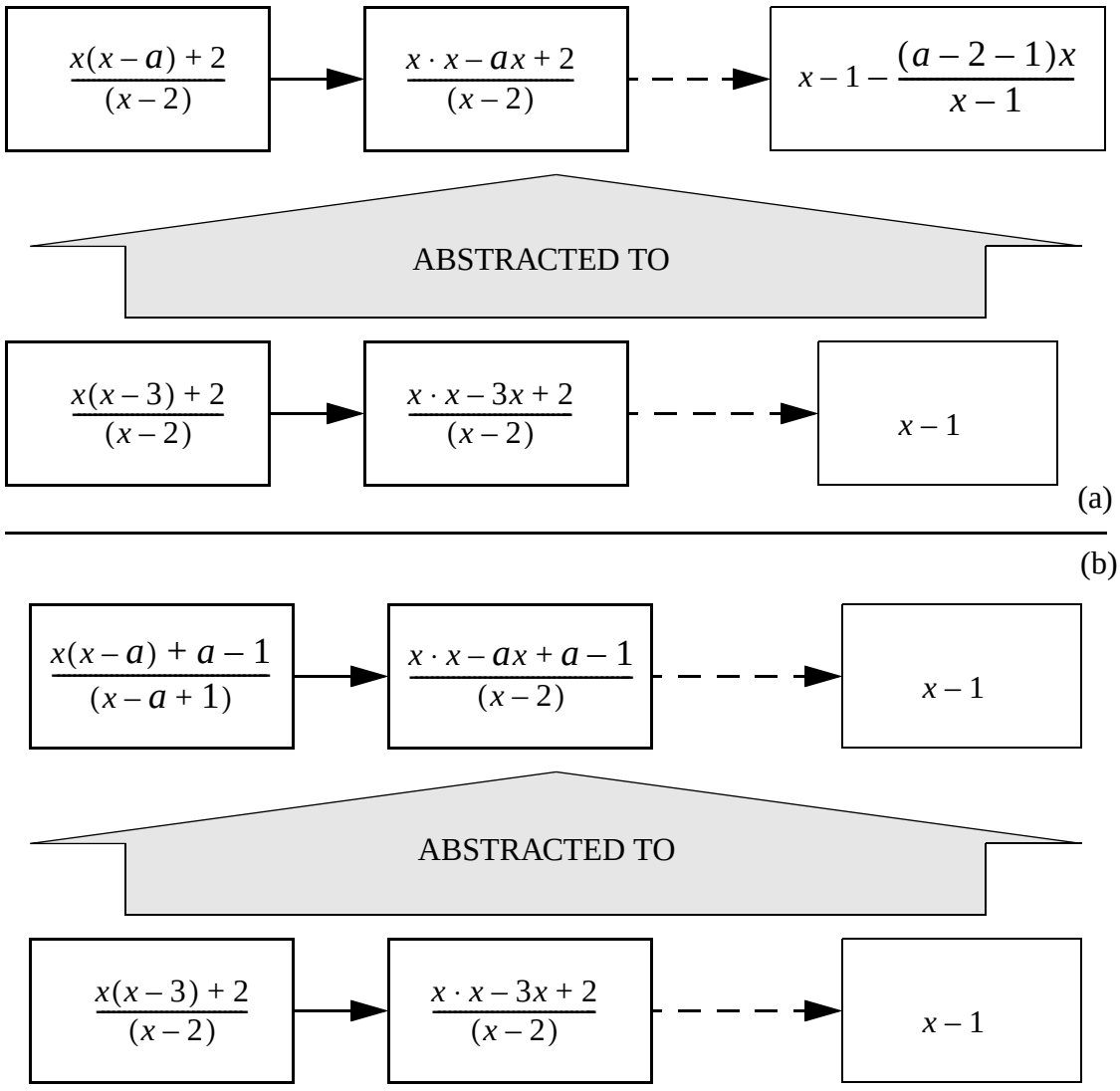


FIGURE 9. Two simple abstraction over the simplification process explaining Simplification 2.

hierarchies based on their explanations viewed as processes. These abstractions will allow the transfer of simplification knowledge from the source to the target. We will build the abstractions over the processes explaining the simplifications. In the mechanism of building abstraction we have to incorporate an evaluation mechanism which will check whether the abstraction produced will satisfy the problem goal and constraints. This verification will have to be done in the *abstracted domain*.

4.5 The Analogical Reasoning Process

In this subsection we describe the analogical reasoning process we are proposing for solving simplification problems. The process is illustrated in Figure 10. The rectangles represent phases of the process, while the rounded rectangles represent data.

4.5.1 Retrieving

Retrieving is the first phase of the analogical reasoning process. Its purpose is to find a simplification that corresponds to an object “similar” to the object that needs to be simplified.

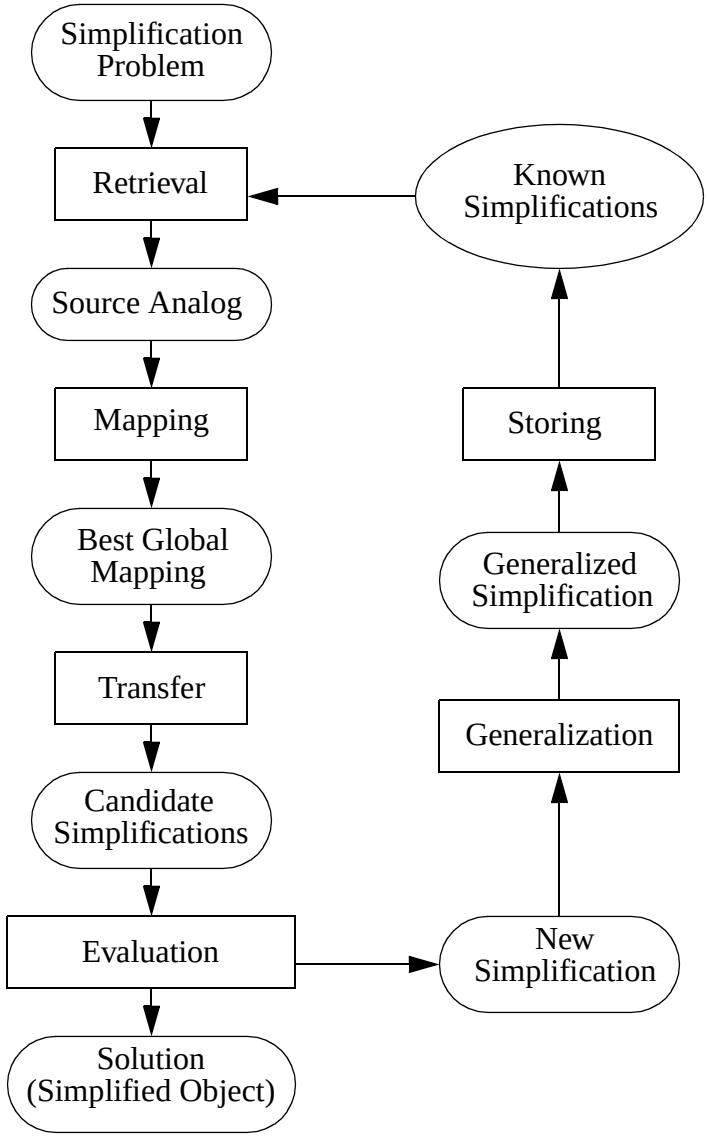


FIGURE 10. The analogical reasoning process used for solving simplification problems

As described earlier, we propose that, for the purpose of retrieval, simplifications are organized first into classes of simplifications corresponding to points of view and, second by an indexing scheme over the relevant elements of the objects involved in the simplification. Consequently retrieving similar simplifications will also work in two stages: a) *pruning*, that is restricting the search to only the class of simplification with the same point of view as the one specified in the simplification problem, and b) *indexing*, that is search using the indexing schemes.

The first stage is trivial and will be implemented by marking each object involved in some simplification with the corresponding point of view. Note, that if we decide to organize points of view into a hierarchy, a more efficient data structure should be used.

The second stage in retrieving a similar simplification is to *use the indexes* built over the relevant elements of the simplifications, in the class under consideration. Since there may be more than one such indexing hierarchy, corresponding to the different types of relevant elements (e.g., object, relations, and attributes) we need to decide which of those hierarchies need to be searched and in what order. One way to organize this is to search the indexes in decreasing order of their level of importance (see Subsection 4.3.2.). For instance, search the hierarchy of relations first, the hierarchy of objects next, and the hier-

archy of attributes last. Given that the objects specified in the simplification problem may be complex, searching each of these indexes may be expensive (at least for a retrieval).

Unfortunately, the two step retrieval process we have proposed so far may be inefficient for retrieving. This will happen if the problem contains many elements that can be associated with elements in the relevant elements of the simplifications under consideration.

To illustrate this, let us consider that our simplification problem is Problem 1 and that the collection of known simplifications that remained after the point of view based pruning, consists of Simplification 1 through 3. We illustrate this in Figure 11 (since the retrieval is only based on the more complex parts of the known simplification, we omit the rest of the representation for all the simplifications). In the figure we only show the possible associations between all the '*' operators in the target and the relevant element of Simplification 1. Although the relevant portion of Simplification 1 only contains three elements, of which only one is a '*' operator, there are five possible associations with the five occurrences of a '*' operator in the target. Note that there will be a total of 20 possible associations for the operator '*' only!

This problem calls for further improvement of the performance of the retrieval process.

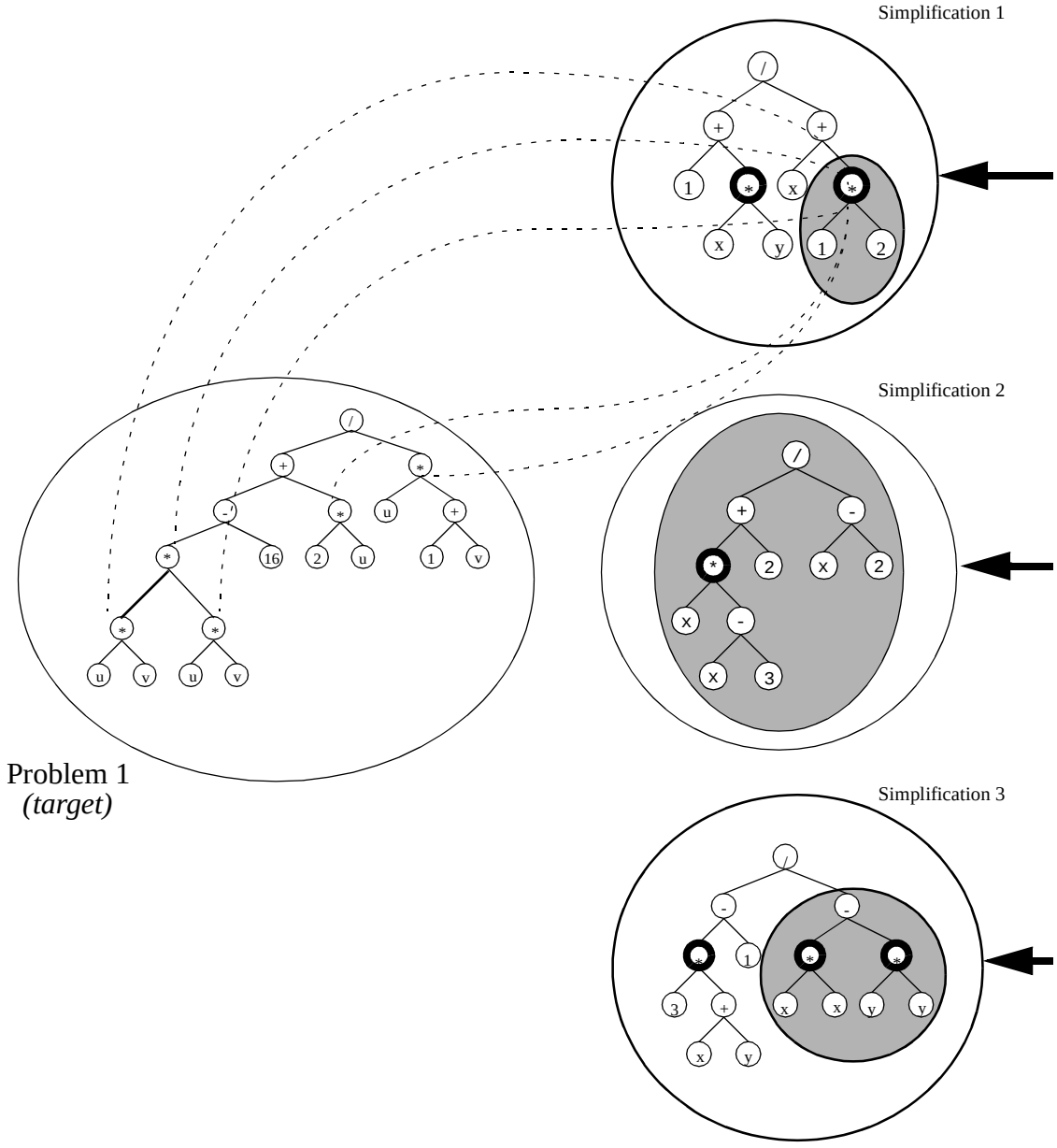


FIGURE 11. Possible associations of the '*' operators in the target with corresponding elements in one of the source simplifications

Such an improvement could be achieved with a pruning scheme similar to the *content-vector* based filtering proposed by Gentner & Forbus [1991]. This approach would filter the set of candidates for retrieval, based on the number of their features considered in the process. Forbus and Gentner propose building the content-vectors using the counts of the relations, functions and predicates in a structural description. These counts are then used to only select candidates for similarity which have the corresponding numbers of elements close to the one in the problem.

The problem with applying that idea directly to our problem is that it has a major limitation: it only works within a domain where there is a fixed number of predefined relations, functions and predicates. In such a domain a fixed sized vector of counters can be assigned to each candidate simplification. If, however, the analogical reasoning is used across domains the method is not applicable. One possible solution to overcome this problem would be to make a fixed association between relations, functions and predicates across the domain and build the content-vectors based on this association. This would however render the problem solving very inflexible.

In conclusion, for retrieving similar simplifications we propose a two-stage process. In the first stage the simplifications from other points of view than the one of the problem are

pruned from the search. In the second stage multiple indexing is used to retrieve candidate simplifications. In addition, for simplifications retrieved which are in the same domain as the problem, we will use feature-vector filtering to prune the search space.

The result of retrieving is a set of candidate analogs: that is, simplifications that are likely to be adaptable to the new simplification problem. Each candidate analog consists of a set of *match hypotheses* on which its selection as a candidate was based. A match hypothesis associates an element (e.g., relation, object or attribute) belonging to the relevant elements of the candidate analog, with an element of the object to be simplified. Each match hypothesis has associated with it a *score* computed during the retrieval process, which expresses its quality (e.g., match hypotheses between relations are considered as having higher quality than the ones between attributes of objects).

The candidate analog with the highest score is selected to be used in the next phase of the analogical reasoning process. In the rest of this section we will assume that a candidate analog has already been selected. As usual, we will refer to the selected candidate analog as the “source” and to the simplification problem as the “target”.

4.5.2 Mapping

Mapping is the second phase of the analogical reasoning process. It builds maximal sets of consistent correspondences (*matches*) between relevant elements of the source and elements in the target, called *global mappings* (or *gmap*s, as the are called in the Structure Mapping Engine (SME) literature). For mapping we propose to use a modified version of Falkenheiner's SME [Falkenheiner et al., 1986]. In the next section we give a short description of how SME works, emphasizing the modifications we suggest so that it suits our purposes. In this description we will follow the explanation given by Forbus & Bungler [1990].

To illustrate our discussion, let us consider that the problem to be solved is Problem 1 and that Simplification 4 is the source. Figure 12 represents the structures of these two expressions. The dotted lines represent some of the match hypotheses. Because the number of match hypotheses associated with this source analog is too high, for our explanation we will only use the ones marked. For easier reference we labeled the matches involved in the match hypothesis. We used numbers to label elements in the source and lower case letters to label the elements in the target.

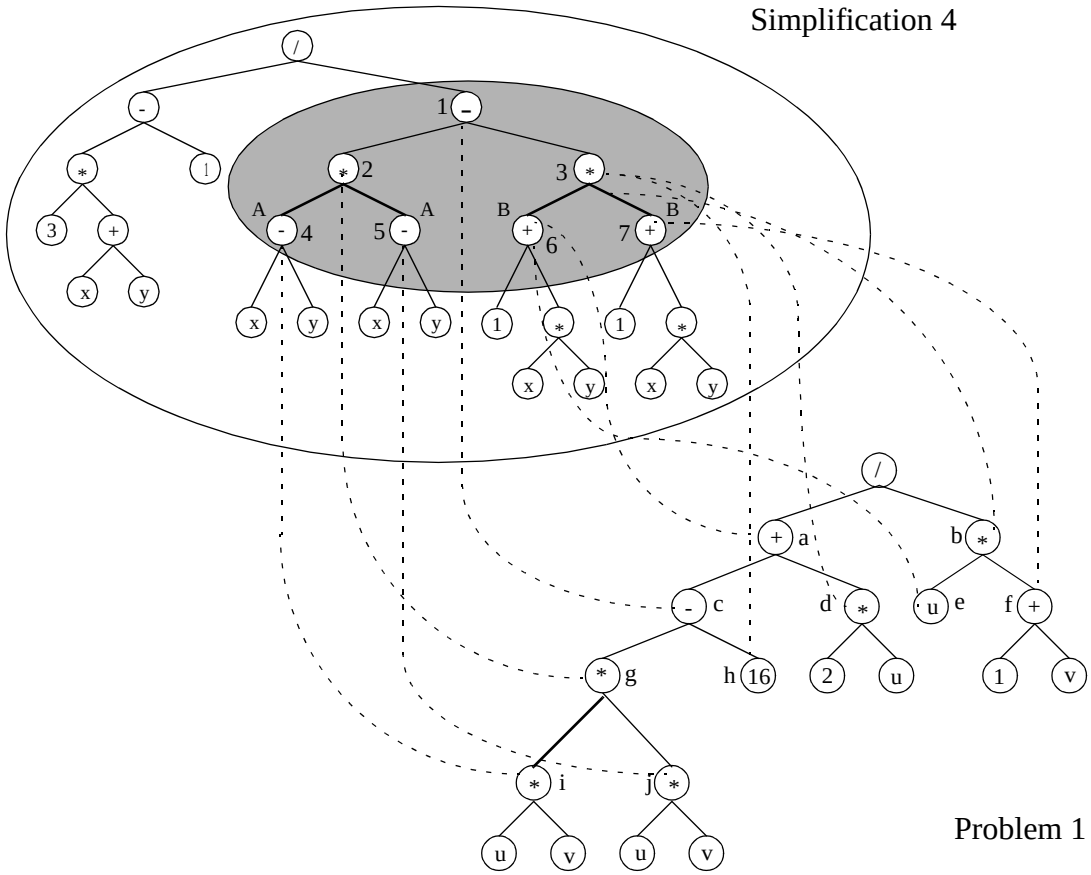


FIGURE 12. An example of a set of match hypotheses between one element of Problem 1 and similar relevant elements in Simplification 4, the selected candidate source analog.

4.5.2.1 Structure Mapping

SME takes as input two descriptions, one of the source and one of the target, and produces as output a set of gmaps of the source onto the target. Each gmap contains a maximal set

of matches. Here ‘maximal’ means that adding any match to it would violate the consistency of the gmap. SME also attaches to each gmap a *structural evaluation score* which provides an indication of the quality of the mapping.

The original version of SME, proposed by Falkenheiner et al. [1986], begins the mapping process by computing match hypotheses. Each match hypothesis represents a potential correspondence between relevant a element of the source and an element of the target. To construct these match hypotheses, SME relies on a set of rules which specify what kinds of elements should be placed in correspondence.

In our approach to solving the simplification problem by analogical reasoning, the retrieving process associates with each candidate source analog a set of correspondences between relevant elements of the source and elements in the target. Each of these correspondences has assigned to it a score that is an indication of the quality of that correspondence. Our implementation of SME uses these correspondences as initial match hypotheses

The next step in the operation of SME is to filter and combine the match hypotheses constructed. First, match hypotheses involving elements (functions and relations) whose arguments cannot be placed in correspondence are eliminated from further consideration.

Next, SME checks local consistency constraints between pairs of match hypotheses to detect violations of the one-to-one constraint. This means that pairs of match hypotheses which would map two different elements in the source to the same element in the target, or would map the same element in the source to different elements in the target are separated as being inconsistent.

The third step in the operation of SME is the construction of maximal sets of consistent match hypotheses, i.e., gmaps. This is performed by combining systems of match hypotheses generated in the previous step and testing them for consistency. A system of match hypotheses will be maximal if it cannot be consistently extended any further by combining it with some match hypothesis.

Figure 13 shows the match hierarchies obtained after the filtering step for the example illustrated in Figure 12. The two way arrows indicate violations of the one-to-one constraint. The darker ones connect leaf matches that *contradict each other directly,* that is matches which associate the same element in the source with two or more different elements in the target. The lighter arrows connect contradicting match hierarchies. The maximal sets of consistent matches obtained for our example are shown in Figure 14.

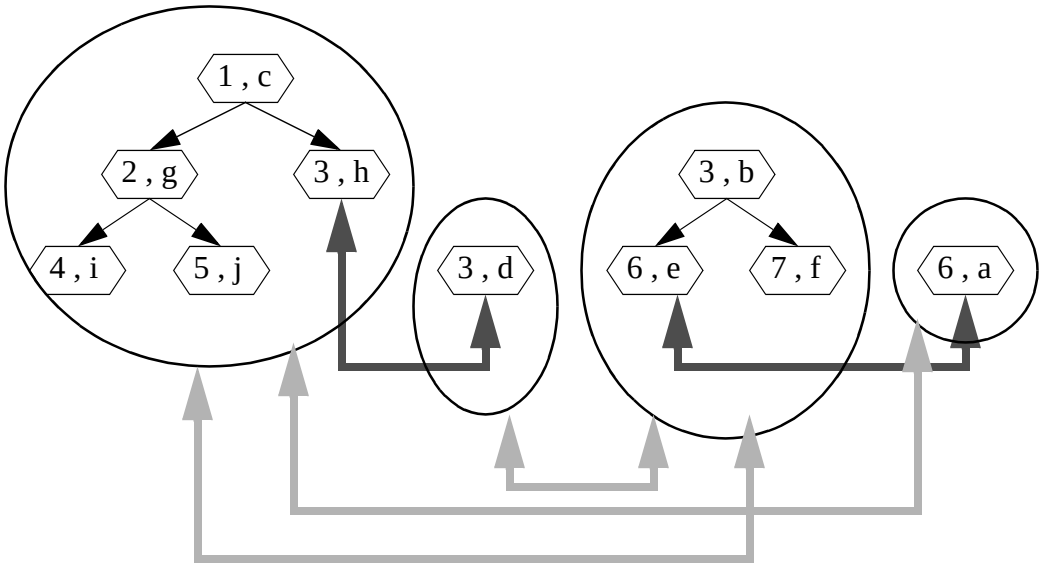


FIGURE 13. The match hierarchies for the example in Figure 12. Two way arrows show inconsistent pairs of matches (darker arrows) and hierarchies (lighter arrows).

Note, that the process of building gmaps uses only the relevant elements of the simplification corresponding to the source.

4.5.2.2 Structural Evaluation of Mappings

The mapping process produces a set of global mappings which can be the basis for different simplifications that are likely to be applicable to the simplification problem to be solved. Some of these global mappings may be better than others, in the sense that they

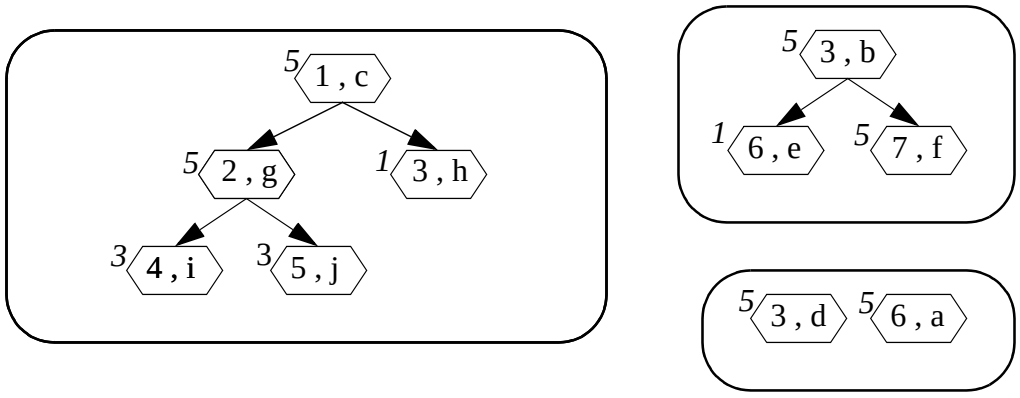


FIGURE 14. The gmaps built for the example in Figure 12. The labels represent scores assigned to the match hypotheses.

represent better matches between the corresponding source simplification and the target problem. Our purpose is to select the best of these global mappings to increase the chances of generating a simplification in the target. For this purpose we need to define a measure for estimating the quality of global mappings.

Measuring the quality of a global mapping should take into account two factors: the structure of the mapping and the quality of the correspondences (matches) involved in the mapping. This can be achieved by accumulating the measures of quality of the individual matches in the mapping over the structure of the global mapping. We call the result of applying such a measure to a global mapping the *structural evaluation score of* that mapping.

There are two general approaches used in accumulating the measures of quality for the matches involved, the *top-down approach* and the *bottom-up approach*.

The top-down approach for accumulating measures of match quality along the structure of a global mapping starts at the root and recursively propagates the quality of each match to all of its descendants. The rule of propagation has the following general form [Forbus & Gentner, 1989]: a match hypotheses adds its score to the match hypotheses of its descendants. The score of the global mapping is then computed by adding together all the scores accumulated in the leaf matches. The intuition behind this approach is that high scores will accumulate in the "leaf" matches expressing their role in supporting high level or complex systems of relations.

The bottom-up approach to measuring the quality of a global mapping computes the score of a global mapping by starting at the root match and adding its score with the recursively computed scores of its descendant matches. This means essentially an upward propagation of scores from the leaves to the root. The score of the global match will be the score accumulated in the root match (or the sum of the scores accumulated in the root matches if the global mapping consists of a set of tress of matches). The intuition behind bottom-up accumulating of scores is to increase the scores of matches that are supported by a larger

system of relations.

To explain how the scores for global mappings are computed we need to assign scores to the different kinds of match hypotheses. Let us assume that a match hypothesis connecting two identical operators has associated with it a score of 5, a match hypothesis connecting two constants or two variables has associated with it a score of 3, while a match hypothesis connection two different operators, or an operator and an operand has associated with it a score of 1. We labeled the match hypotheses in Figure 14 with their initial score, that is with the score associated to them at the time of retrieval.

The results of applying the two approaches to our example are presented in Figure 15. The numbers below the ‘leaf’ matches represent the scores accumulated by the top-down approach. The numbers above the ‘root’ matches represent the scores accumulated by the bottom-up approach. The pairs of numbers below the rounded boxes representing the global mappings represent the finals scores for the two approaches (top-down/bottom-up) for the corresponding gmap. Both of the types of measures clearly indicate global mapping A as being the best one.

In this dissertation we propose to use the top-down approach to evaluate the quality of glo-

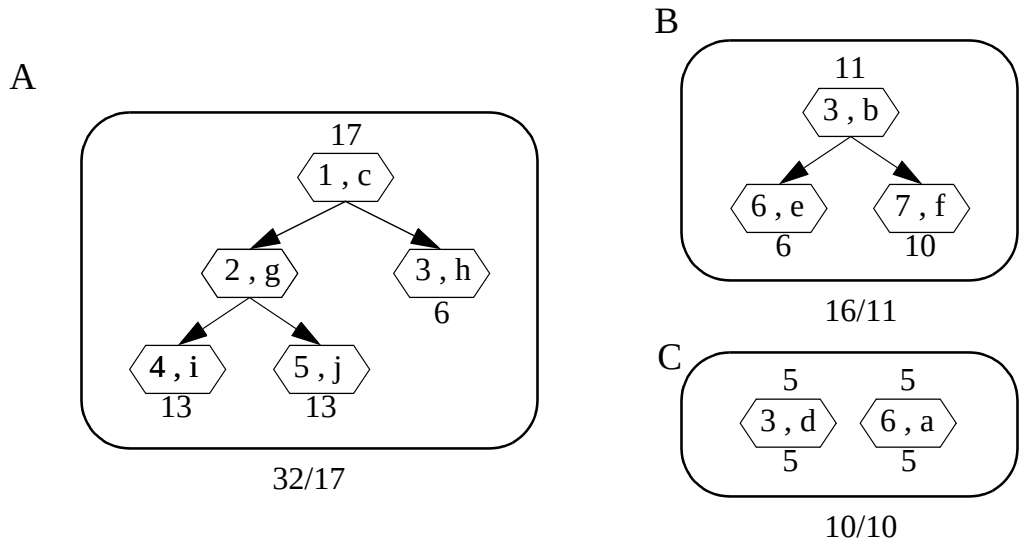


FIGURE 15. The results of evaluating the global mappings for the example in Figure 12. The pairs of numbers represent the structural evaluation score of the top-down approach versus the ones of the bottom-up approach.

bal mappings. The main reason for this decision is that the measure obtained by the top-down approach contains structural information as opposed to the bottom-up one, which is rather a weighted count of the scores of the matches involved in the mapping. The top-down approach will assign higher scores to gmaps with more relation matches that rely on more object matches. For global mappings consisting of the same number of matches, the measuring quality using the top-down approach to accumulate scores favors “deeper structures”.

At the end of the mapping phase the global mapping with the highest structural evaluation score will be selected for further consideration in the analogical reasoning process.

4.5.3 Transferring Simplification Knowledge

Once a global mapping has been selected as the best candidate for analogical transfer, it will be used to compute *candidate simplifications*. A candidate simplifications is a simplification in the source which can be hypothesized to be applicable in the target as a result of the correspondences of the global mapping.

A candidate simplification is computed by finding elements in the source which are consistent with the global mapping's correspondences, but are not in fact included in them. We will call these elements *unbound elements*. An unbound element is *consistent* with a given global mapping if there is no match in the global mapping which has that element as its member. Unbound elements are searched for in the set of relevant elements of the source since those are the only ones that play some role in that simplification.

Once the unbound elements are found the existence of corresponding elements in the target can be hypothesized. Building these hypotheses is performed by the simplification knowledge transfer process.

How exactly the simplification knowledge will be transferred depends on whether the explanation for the simplification is given by a difference or by a simplification process. The two different ways of transferring simplification knowledge will be presented in the next two subsections.

4.5.3.1 Transferring Differences

If the explanation of the source (simplification) is given by a difference (between the two objects involved), then the knowledge transfer will consist of applying the difference to the target and, if needed, adapting the resulting simplification.

To apply the difference to the target, the difference explaining the source must be first transformed according to the global mapping. This means to view the matches contained in the global mapping as substitutions. These substitutions are applied to the difference, i.e., in the representation of the difference, each occurrence of (the reference to) an element which is the first (source) member of a match will be substituted by the second (target) member of the match. Those elements in the difference which are not first member of any matches will be replaced by *variables*. During the process of adapting the simplification, these values will be assigned to variables. After the difference in the source has been transformed according to the global mapping (viewed as a substitution), it is applied to the

target. This will result in a new object representation which may be incomplete (contain variables, or unspecified portions).

Let us assume that for our example problem global mapping A in Figure 15 has been selected as the best one. Of the relevant elements in the source there are two which are not associated with any element by the global mapping A. Applying A as a substitution will result in a difference of the form: replace $(uv)^2 - T^2$ by $(uv + T)(uv - T)$, where T is a variable that corresponds to the subexpression B, which is not associated with any element in the target, according to the global mapping A. Figure 16 illustrates the result of applying the substitution to the example simplification problem (the '?' sign corresponds to the place where adaptation needs to be performed).

After transforming the difference in the source according to the global mapping and applying it to the target problem, the representation of a new object is obtained. This object may be incomplete, and as a consequence it may need to be *adapted*. Adapting an incomplete object is done by associating with the elements in the difference (of the source) which are not first member (i.e., the member from the source) of any matches, objects from the domain of the target. How these objects are selected depends on the target domain. The only requirement is that the new associations be consistent with the global

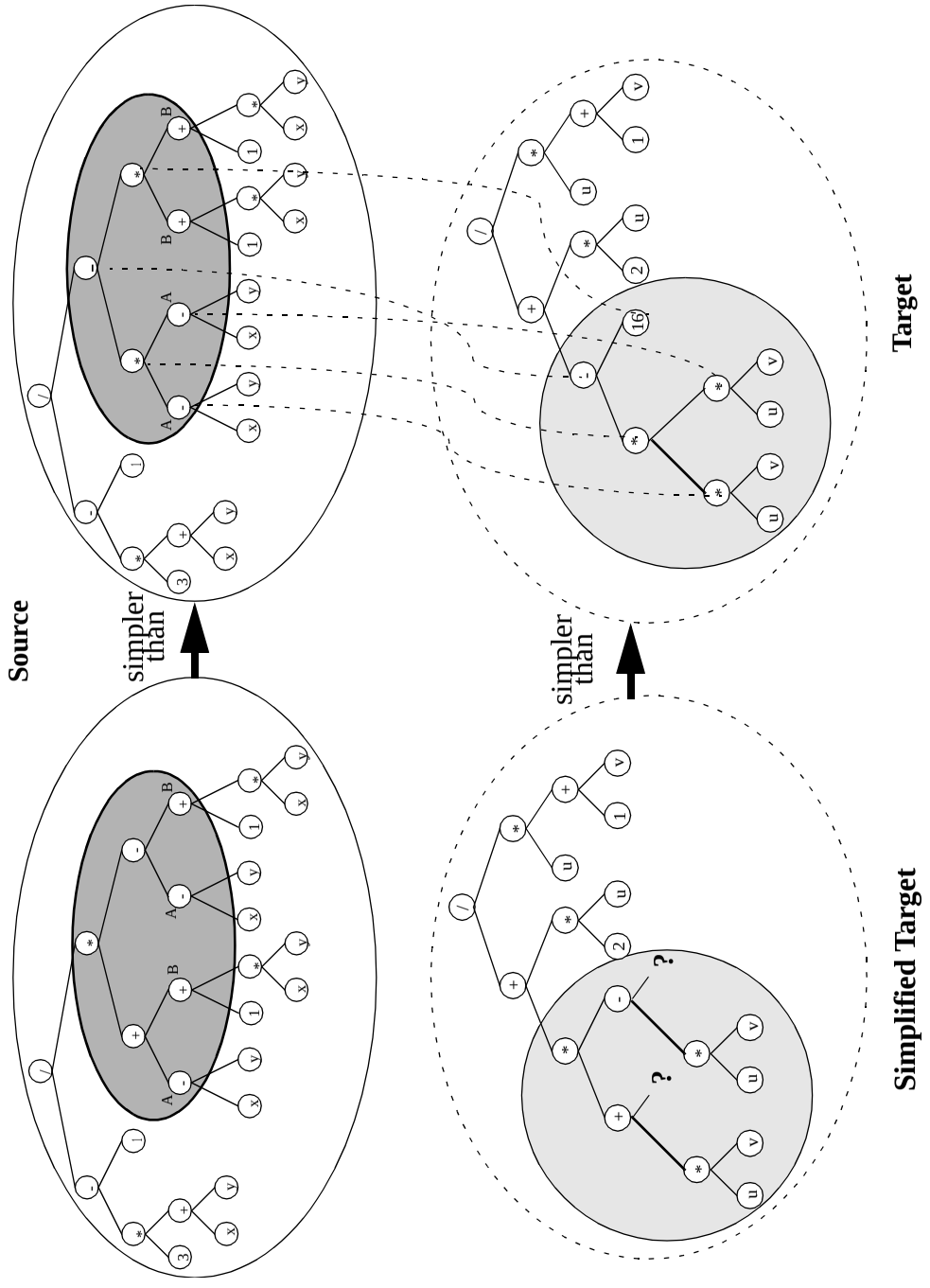


FIGURE 16. Applying global mapping A to Simplification 4.

mapping. It is possible that there is more than one way to select those objects in a manner consistent with the global mapping. These will correspond to different new objects. These objects need to be evaluated to check if they satisfy the requirements on the object, the constraints on the simplification, and if they are less complex (simpler) than the target. The objects which satisfy all of the above will be 'simplified objects' corresponding to the target. Each of these simplified objects corresponds to a new simplification. We call the set of simplifications generated *candidate simplifications.*

For our example the variable T could be replaced with any expression. Some domain specific knowledge needs to be used to actually select the right one, namely we need to know that "any positive number can be written as the square of a positive number'. Based on this we can write $16 = 4 \cdot 4$, which will allow us to associate B with 4 and come up with the simpler expression $(uv + 4)(uv - 4)$.

From all the candidate simplifications we have to select one. Ideally we would select the simplification corresponding to the least complex simplified target produced.

4.5.3.2 Transferring Simplification Processes

If the explanation of the source (simplification) is given by the description of the simplification process, then the knowledge transfer will consist of adapting the simplification process according to the global mapping and applying it to the target.

To adapt the simplification process according to the global mapping means viewing the global mapping as a substitution, and then applying it to the representation of the simplification. This is performed by substituting in each element (i.e., initial state, condition, transformation and final state) of each step of the simplification process, every occurrence of an element which is the first (source) member of a match by the second (target) member of that match. Those elements occurring in the representation of the simplification process which are not members of any matches of the global mapping will be replaced by *variables*. If the process description obtained this way contains variables, we say that it is *incomplete*.

To illustrate this, let us consider the following simplification problem:

Problem 2: Reduce the number of multiplications in the in the following expression:

$$3(3-u)+(u-1)$$

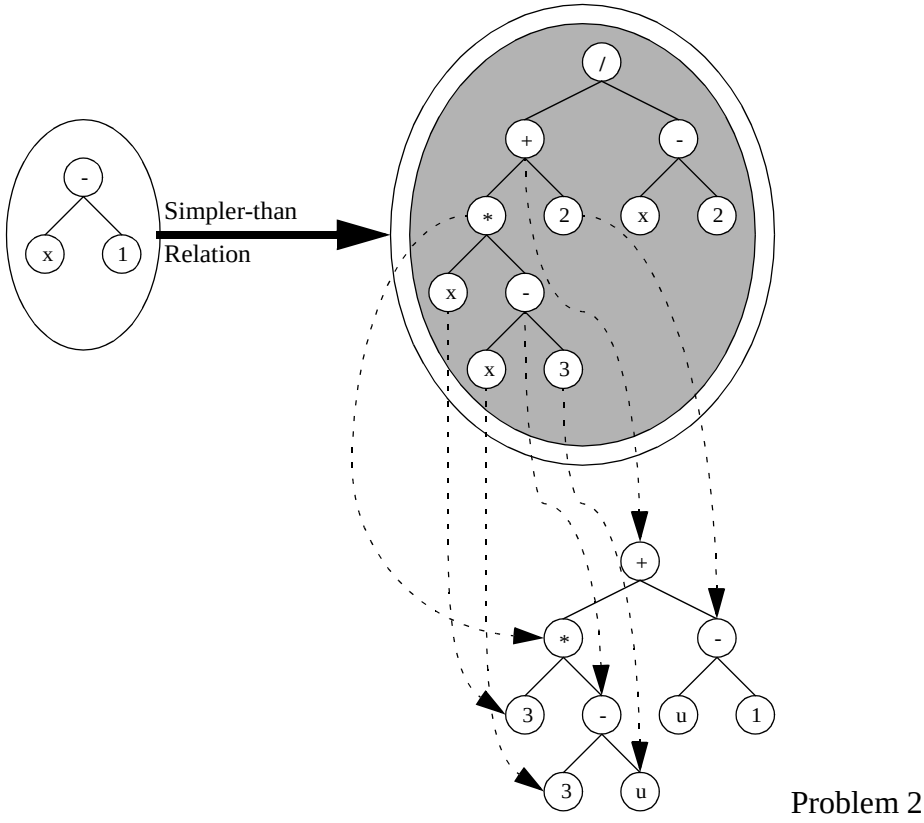


FIGURE 17. The best match retrieved for Problem 2. The dashed arrows represent the match hypotheses in the best gmap built.

Assume that after performing the retrieving phase of the analogical reasoning process, the best match source retrieved was Simplification 2 (Figure 17). Also assume that the dashed arrows connecting elements in the source to elements in the problem represent the match

hypotheses of which the best gmap built by the mapping phase is composed. This global mapping corresponds to a substitution (i.e. association between elements of the source and elements of the target). Remember that the explanation for Simplification 2 was given in the form of a simplification process (see page 114). To adapt that simplification process to Problem 2 according to the global mapping built, we need to apply the substitution given by the gmap to the simplification process specified in explanation of Simplification 2. The process description resulting from this is shown in Figure 18. Note, that the process description contains unspecified elements (two operators) and, as a consequence, is incomplete.

To apply an incomplete process description to the target two approaches are possible: a) bind the variables to (compatible) elements in the target and than apply the process obtained, or b) build an abstraction of the process and apply that abstract process to the target. For the first approach there may be several different way elements in the target can bound to the variables in the incomplete process description. Taking this approach would mean to consider all the possible ways this can be done, for each of them apply the process ad then perform the evaluations of the results. Since this may be quite expensive, we propose to use the second approach.

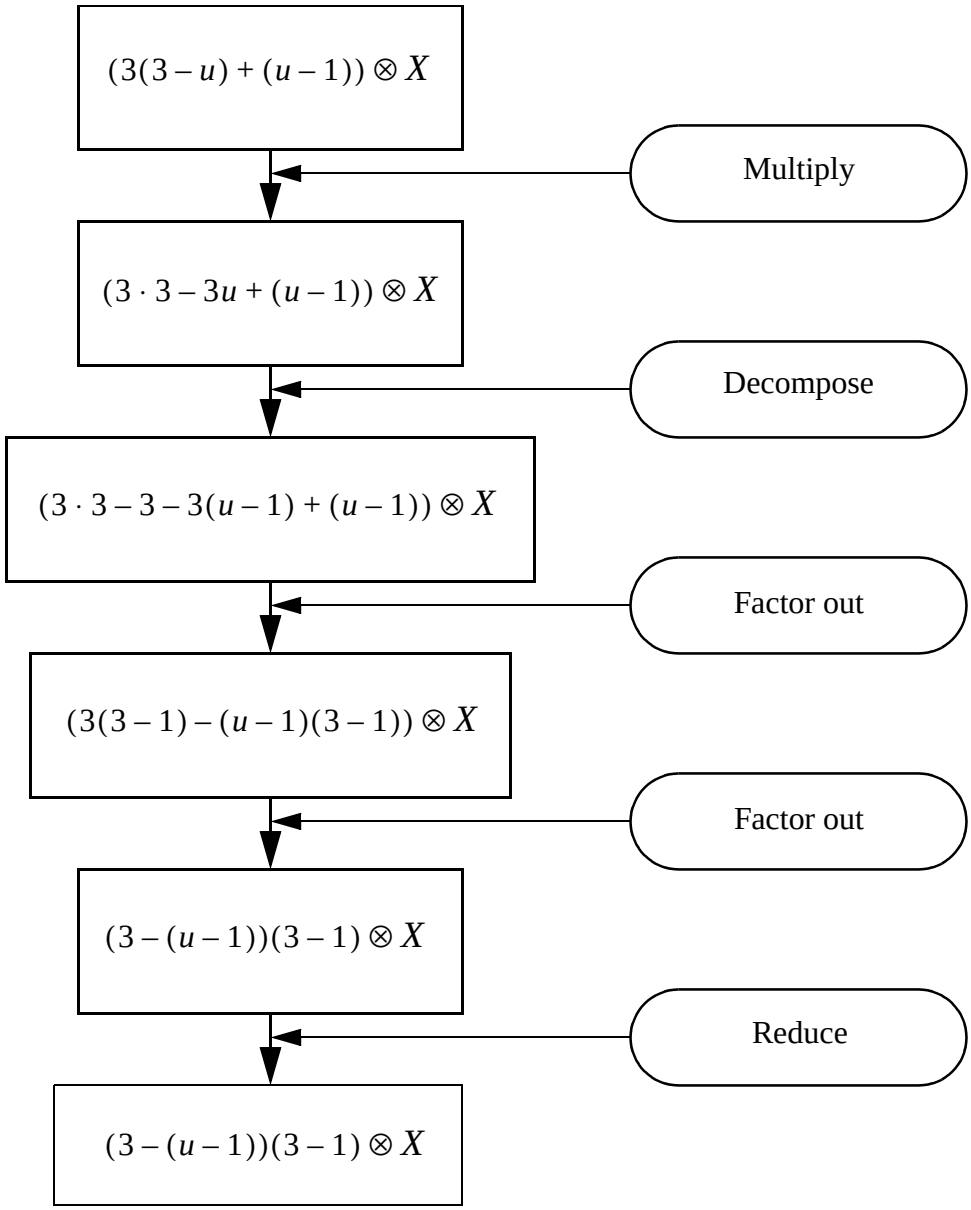


FIGURE 18. Adapted simplification process for Problem 2

Building an abstraction of an incomplete simplification process can be done by removing from the process description those elements which are not associated with elements in the target by the best global mapping found (i.e., the variables). The process description thus obtained will applicable to the target.

For our example the abstraction would consist of removing from each of the steps of the "variable operator" ☹ and the variable X. Note that, applying this abstracted process to the target will yield the expression $(3 - (u - 1))(3 - 1)$, which is not simpler then the original expression (i.e., Problem 2) from the point of view considered. This shows that evaluating the result of the transfer is absolutely necessary, and that transferring simplification knowledge from the source to the target doesn't necessarily result in a new simplification and. Also, while the result of applying the abstracted process to the target may not be a simpler object than the target itself, it may be possible that it will be easier to simplify within the domain of the target. For instance, using some basic arithmetic calculations, the result produced for our example can be transformed into the equivalent expression $8 - 2u$ (which, by the way, is not simpler from the point of view of the number of multiplications, however is simpler from the point of view of the overall number of operations).

4.5.4 Evaluating the Result of the Simplification

Each of the simplified targets resulting from transferring the simplification knowledge and adapting the result has to be evaluated for the: a) *requirements* on the object, b) *constraints* of the simplification and c) *complexity.*

The requirements on the objects (e.g., design requirements in the case of design simplification problems) have to be checked by domain specific methods. These may be either simulation, reasoning or evaluation. The constraints (which refer to properties of the target that the simplification process has to preserve) can also be checked by simulation, reasoning or evaluation. Finally, the verification complexity condition is done by applying the complexity measure corresponding to the simplification problem (i.e., to the point of view specified in the simplification problem) and compare the result with the complexity of the target.

For our example, we need to check the expressions resulting by assigning different expressions to the variable T, whether they are correct (well formed and legal), if they are equivalent to the target (i.e., they evaluate to the same value for every legal assignment of the variables involved) and if their complexity is lower than that of the target. For example, the object produced for this example satisfies the first two conditions and, since the com-

```
BestSimplifiedObject
 begin
    best := NULL;
    while new objects can be produced do
       o := ProduceNewObject;
       if SatisfiesRequirements(o) then
          if SatisfiesConstraints(o) then
             if Complexity(o) < best then
                best := o;
          end
       end
    end
    return best;
 end.
```

FIGURE 19. Algorithm for producing the best simplified object

plexity (as measured by the number of multiplications) is the same as the complexity of the target (5), it does not satisfy the last one.

Evaluation of a new object may be expensive. In addition, if all the possible new objects are generated first, and only than evaluation and the selection of the best (simplest) is performed, the simplification process will become very time consuming. To reduce the time of selecting the best simpler object we propose to perform evaluation interleaved with the generation of new objects. Thus, after a new (partial) object was produced as a result of transferring the simplification process producing new objects and selecting the best simplified object will be performed according to the algorithm presented in Figure 19. This

way of organizing generation and evaluation will immediately discard any object that doesn't satisfy the requirements and constraints and only evaluates the complexity of those objects that do.

After the best simpler object was selected the corresponding simplification can be generated. The target and the new object will be respectively the 'more complicated' and 'simpler' objects involved in the simplification. The explanation will be computed as the difference between the two objects, and the relevance calculation will be applied.

4.5.5 Generalization and Storing

The simplification corresponding to the target can be used to extend the simplification database. This can be done by either adding the simplification to the database, or by adding to the database a newly generated simplification which is a generalization over the source simplification and the simplification produced by the analogical reasoning process.

Simply adding the simplification to the simplification database is straightforward, as it only requires creating the appropriate links connecting it to the structure of the database. The question that is raised here is whether the simplification is "new" enough (i.e., different enough from simplifications already on the database) so that it is worth being stored.

To create a generalization over two simplifications (in our case the source simplification and the simplification produced by the analogical reasoning mechanism) we can use abstraction. This abstraction has to be applied both to the objects involved in the two simplifications and to the explanations of those simplifications. The abstraction process we are proposing for building a generalization over two simplifications consists of two phases:

- remove the elements that are not relevant from the objects involved, and
- build the generalizations from those elements that are shared by the simplifications.

Abstracting from the irrelevant elements of a simplification is simple because the relevant elements are explicitly known. Removing the elements that are not relevant in both of the simplifications result in two new simplifications in which all the elements represented are relevant. Abstracting from the elements not shared by the two simplification is performed by considering the global mapping produced by the structure mapping and the variable substitution used to produce the simplified object. These two represent a mapping of each element in the representation of the source simplification to an element of the newly generated simplification. The abstraction then will consist of finding for each of these mappings a common supertype of the types of the two elements involved. This common supertype can be found by following the predefined object type hierarchy.

CHAPTER 5

Application: Simplification of Designs

In this chapter we will apply the approach proposed in Chapter 4 for solving the simplifications of designs. In our explanations we will refer to simple examples from the domain of door lock design (Chapter 1 has already introduced an example from this domain). The first section of the chapter describes this domain. The next section defines the representation we are proposing to use for designs. It first defines the representations for the three aspects of designs we are considering in our research: structure, behavior and function. Then it discusses the connections and dependencies between the different aspects of a design and how those connections and dependencies are reflected in the representation. The third section discusses the problem of defining the complexity of designs. As in this research we limit ourselves to the above mentioned three aspects of designs, complexity is

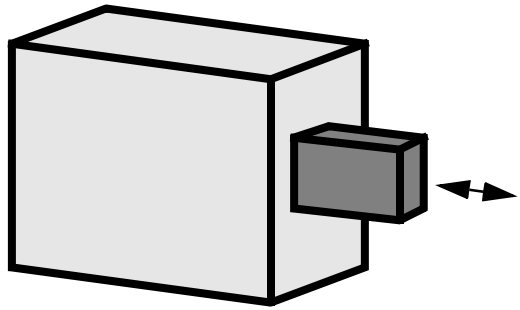


FIGURE 1. A schematic door lock

only defined with respect to structure, behavior and function. Section four describes what we mean by structural, behavioral and functional simplification in different contexts.

5.1 The Door Lock Domain

Simple door locks will be used throughout this chapter to explain and illustrate our ideas on design simplification by analogical reasoning.

We consider a door lock (Figure 1) to be a device that allows and prevents the opening of a closed door (or gate or window). It is composed of a *box* and a *bolt* which can be fully retracted into the box as a consequence of some input applied. When the bolt is com-

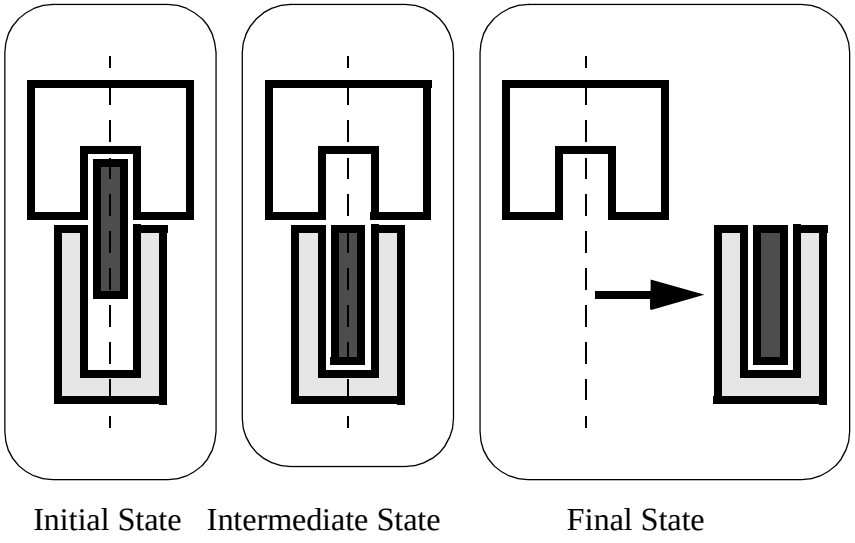


FIGURE 2. Behavior of a door lock

pletely retracted it allows the door (together with the whole lock) to move into the open position. When the input is no longer applied the bolt returns to its initial (unretracted position).

For our purposes, the opening of the door lock is a three state process (Figure 2). The initial state corresponds to the closed door and is characterized by an unretracted bolt and a shut door. The second (intermediate) state is characterized by a retracted bolt and a shut door. The door lock may get into this state from the initial state as a consequence of apply-

ing some input. The final state corresponds to the open door and is characterized by a retracted bolt and an open door. This state can be reached from the intermediate state by applying a second input to the door lock. The closing of the door lock can be described in a similar way.

In our examples we shall limit ourselves to door locks implemented using simple components with distinct functional roles (such as levers, cams, shafts, etc.) [Chakrabarti & Tang 1996]. We shall also assume that the inputs to a door lock will be forces characterized by their directions.

Figure 3 (a) illustrates a design for the door lock device. The door lock consists of a *handle*, a *cam*, a *bolt* and a *spring*. The Open function of the door lock is achieved by first applying input I1 (a vertical force with its direction pointing downwards) to the handle. When this force becomes greater than the restoring force of the spring, the handle will rotate around its end opposite to where the input was applied. Thus, the handle will transform a linear movement (corresponding to input I1) to a rotational movement of the cam. The cam, which is hooked to the bolt, transforms this rotational movement back to a linear one (but one with a horizontal direction), retracting the bolt into the box. When the bolt is

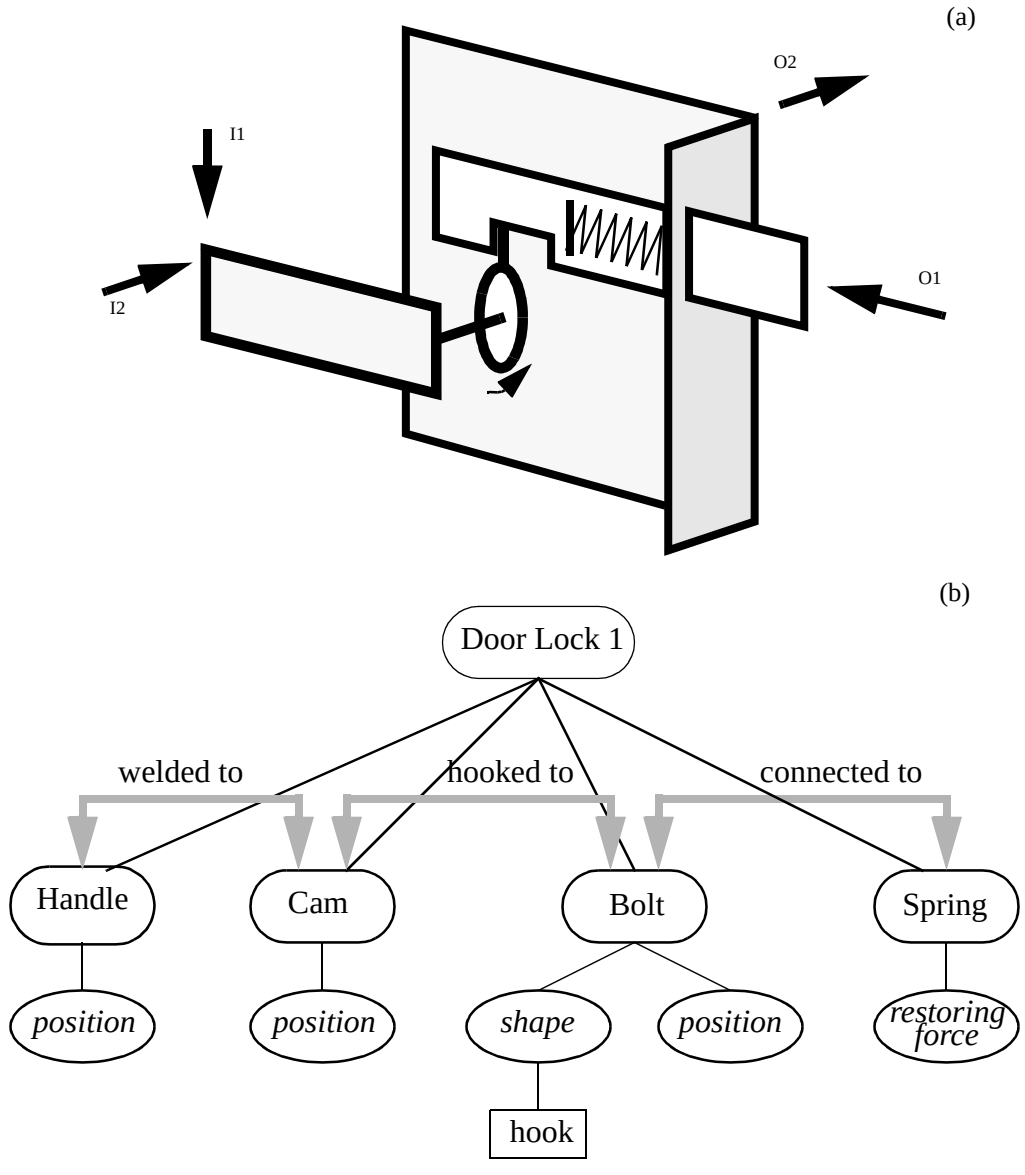


FIGURE 3. Door lock implementing the cam mechanism using a cam: (a) schematic and (b) structural representation

retracted into the box, the second input, I2 is applied to the handle. This is a horizontal input, perpendicular to the plane of the door onto which the door lock is mounted.

5.2 Representing Designs

We represent a design by representing its structure, behavior and function and the connections between these aspects. In the following subsections we describe the representations we are proposing for each of these aspects.

5.2.1 Representing Structure

For representing the structure of a design we use an *object, component, attribute and relation ontology.* A design is represented as an objects which may be composed of several other objects, called the object's *components*. Designs may have *attributes* attached to them. An attribute is a function that may be applied to an object to obtain a characteristic of the object. For instance the attribute "color" if applied to an object will give the color of that object. Attributes are not object specific, or object class specific, in the sense that they may be applied to many different objects of different object classes. For instance, a door lock may have a color, or a clothespin may have a color, and so on. However, there may be

objects for which a given attribute doesn't make sense (is undefined). For example a computer program doesn't have a smell.

A design may be in relations with its environment, that is, with objects which are not its components, or its component's components, and so on. We call such relations *external relations*. For instance a door lock is *mounted* onto a door. This connection between the door lock and the door it is mounted onto is an external relation of the door lock. For a given design, there may be relations between its components. We call such relations local (internal) relations. For instance the components cam and bolt of our door lock example in Figure 3 (a), are hooked to each other. This connection by hooking is a local relation of the door lock.

Figure 3(b) gives a structural representation of the door lock illustrated in Figure 3(a). The rounded boxes represent objects, while the ovals represent attributes. A line connecting two rounded boxes means that the object corresponding to the box in a lower position is a components of the one corresponding to the box in a higher position. A line connecting rounded box and an oval means that the attribute corresponding to the oval is an attribute of the object corresponding to the box. Note, that in our representation, if there is no oval representing a given attribute connected to an object, it means that either the attribute

doesn't make sense for that object, or that the representation abstracts from it (e.g., because it is irrelevant to the problem for which the representation is used). We treat both cases identically, considering that the object "doesn't have that attribute. For example, the object "cam" in Figure 3(b) has no attribute "shape" connected to it. Although a cam clearly has a shape, we omitted it because we didn't find it relevant for our purposes. The thick gray arrows in Figure 3(b) represent relations local to the door lock.

5.2.2 Representing Behavior

We view the behavior of a device as a process described by a sequence of state transitions. A state transition is specified by two (partial) state descriptions, the *initial state and the final stat*e, a *condition* and a specification of *how the state transitions is achieved*. A state transition may be achieved by a function, another behavior, or by a physical law.

We represent behaviors by a sequence of state transitions. Each state transition is represented by four elements: the representation of the initial state, the representation of the final state, the representation of the condition and the representation of the means by which the transition is achieved. We represent a state by using a *partial state representation*. A partial state representation consists of a set consisting of attribute/attribute-value

pairs, and relations. Here “partial state” refers to the fact that only those attributes and relations of the design are represented which are affected (i.e., used or changed) by the transition. The condition of a state transition is represented as a logical proposition that may evaluate to ‘true’ or ‘false’. Finally, the transformation is a reference to some other

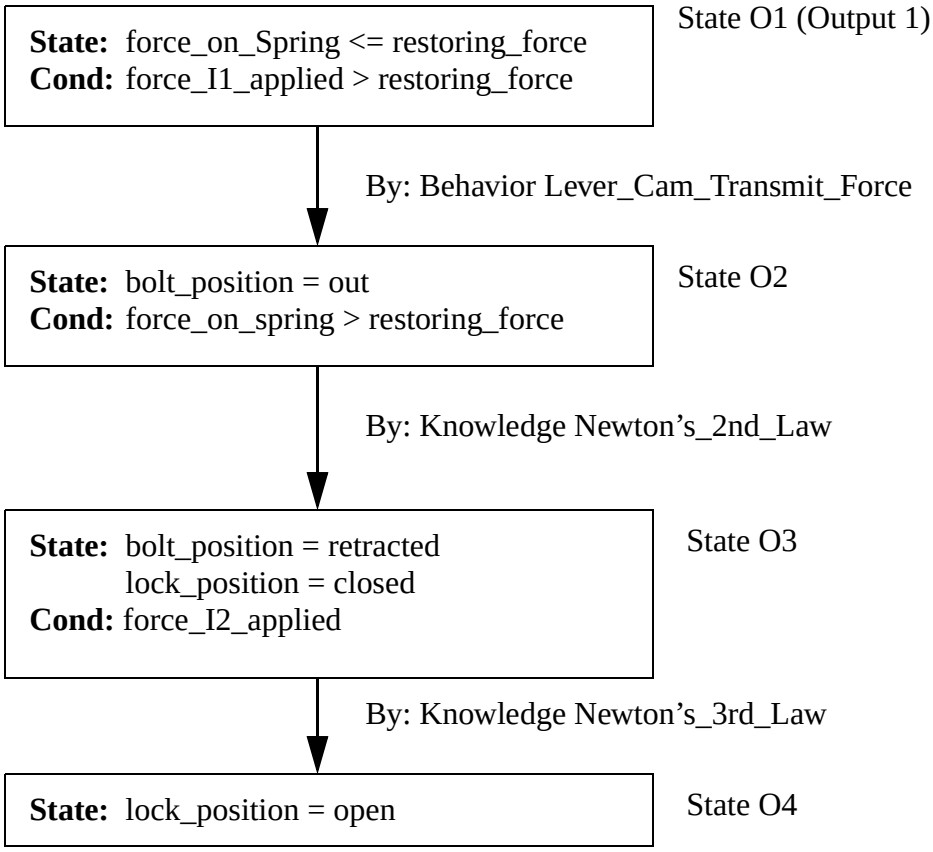


FIGURE 4. State transition graph for the top level Open behavior of the door lock in Figure 3.

behavior of the design, or to a function of its components, or to a physical law. Figure 4 uses a *transition* graph to illustrate the representation of a behavior of the door lock shown in Figure 3. Note that two of the transformations refer to physical laws, while the third one prefers to another behavior, Lever_Cam_Transmit_Force. This behavior (represented in

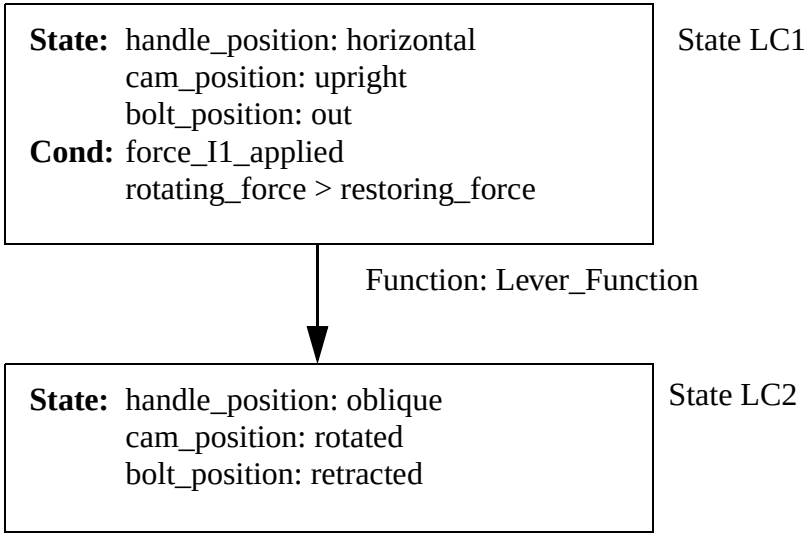


FIGURE 5. State transition graph for the 'Lever_Cam_Transmit_Force' behavior referred to by the 'Open' behavior represented in Figure 4.

Figure 5) describes how the lever-cam combination transmits force and transforms the direction of the movement. Note that this time the transition refers to a function, namely the lever function played by the handle.

5.2.3 Representing Function

A function of a design is defined in terms of its interaction with a given environment. To represent a function of a design we will represent the *environment* in which the design has

to be place into in order to achieve its function, the *interaction* of the design with the environment required to achieve the function, and the way the function is *deployed*.

We represent the *environment* of a design by of a set of objects, which are not components of the design, but are either in some relation with the design, or get into some relation with the design while it achieves its function.

The *interaction* of a design with its environment is a sequence of *inputs* applied to the design and *outputs* produced by the design. We view both the application of an input to a design, and the generation of an output by the design, as the instantiating of a relation between the design and its environment. The difference between the two is usually made based on the “direction” of the relation that is instantiated.

For example, we know that any force acting on an object will cause a reacting force to occur. This reacting force could be viewed as an output of the object. We however view this kind of interaction as “initiated” by the external force, and as a consequence having a feel of “sequence” or direction (action to reaction). This is why we view the application of a force to an object as an input.

```
Function: Open
 Environment:
    - input I1 applied to the handle
    - input I2 applied to the handle
 Interaction:
    - (force_I1_applied > restoring_force) → retract_bolt
    - bolt_retracted → apply_I2
 By (deployment):
    - Open_Behavior
```

FIGURE 6. The Open function of the door lock in Figure 3.

The mode of deployment of a function is represented by representing those properties and relations of the design, and those relations between the design and the environment that determine the causal interactions between the design and the environment.

If the *mode of deployment* assumes a sequence of state transformations of the design we say that the device achieves its function by a behavior. In this dissertation we are only concerned with devices that achieve their function through some behavior. As a consequence we will represent the mode of deployment of a function by a reference to the behavior by which the function is achieved.

Figure 6 shows the representation of the Open function of the door lock in Figure 3. The environment consists of two forces (I1 and I2) that can be applied to the handle of the door

lock as inputs. The interaction is described as a sequence of inputs applied to the door lock and outputs generated by the door lock. The first force applied is I1. If this force is greater than the restoring force of the spring, than the output produces is the retracted bolt. Next, if the bolt is retracted (that is the first output was generated), the second input, that is force I2, is applied. The output generated will be placing the door lock in ‘open’ status. The deployment of this function is represented by a reference to the behavior ‘Open_Behavior’.

5.2.4 Connections and Dependencies between the Different Aspects

The structure, behavior and function of a design are interdependent. Behavior describes a process of transformation of some structural element (component, attribute or relation) of the design. As such, behavior is *strongly dependent* on structure. This dependence is governed by physical laws. In our representation this dependence is expressed by the references to structural elements used in the representation of behavior. For example, all the partial state descriptions in the behavior represented in Figure 4 are expressed in terms of attributes and relations of the door lock’s components. A design may have several different behaviors, corresponding to different sequences of transformations. Our representation maintains a list of all the behaviors of a design which are referred to by at least one func-

tion of the design. Thus, the representation we are proposing explicitly connects the structure of a design to its behaviors through explicit references, and connects every behavior to the structure to which it corresponds to through the references used in the description of transformations.

The kinds of functions we are considering (i.e., functions achieved by behavior), depend on the behavior they are *implemented* (deployed) by. We represent this dependency by an explicit reference to the behavior (or function, or physical law). A behavior may implement more than one functions. Our representation does not explicitly maintain a connection from a behavior to each of the functions it implements. It maintains however, a list of all the intended functions of a design, linked to the representation of the design. Figure 7 illustrates the way the interdependencies explained above are represented for our example door lock.

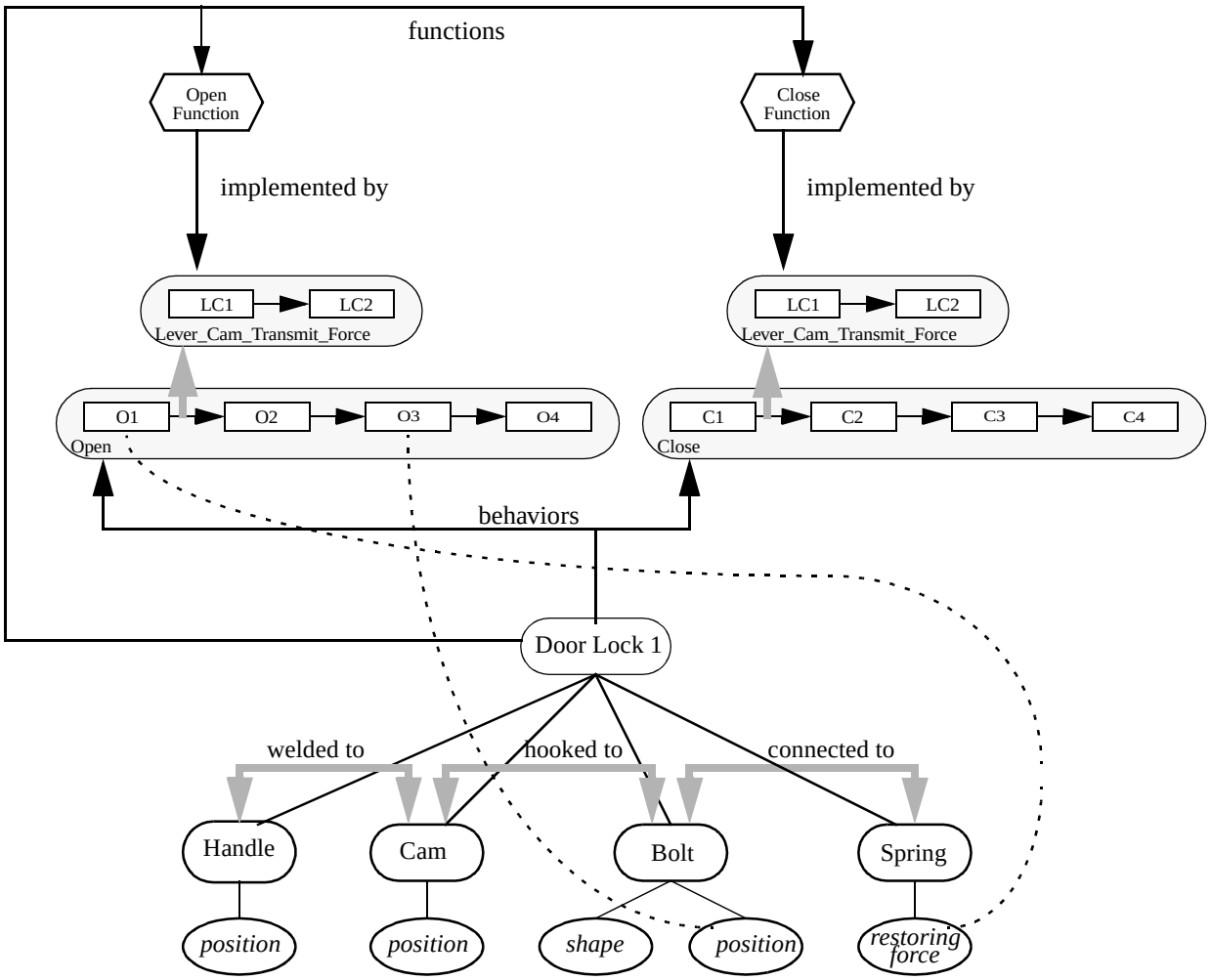


FIGURE 7. Dependencies between structure, behavior and function for the door lock in Figure 3.

5.3 Contexts, Aspects and Measures for Design Complexity

As described in Chapter 2, the complexity of an object can only be defined with respect to a point of view, that is, a combination of context, aspect and measure. In this subsection we will present some possible points of view for defining the complexity of a design.

5.3.1 Contexts for Measuring Design Complexity

As stated earlier, a context, in which complexity of an object will be measured, is a process that is performed on that object (or in which that object is involved). As a consequence, considering the processes that may be performed on designs, we can talk about complexity in the context of *designing*, *manufacturing*, *using*, *repairing* and so on. It should be clear that each of these contexts may require different (sometimes even conflicting) views on complexity. For instance, when using a design, its function matters, while during the process of manufacturing it does not (or, at least it is less obvious that it does, although during manufacturing it may be required that the function is *not* achieved).

5.3.2 Aspects for measuring Design Complexity

We relate the aspects for measuring design complexity to the levels of description of designs, that is structure, behavior and function. Thus, we will talk about *structural, behavioral* and *functional complexity* of a design.

Structural complexity of a design means that measuring complexity will refer to the structural aspect of the design, that is, attributes, components and local relations. *Behavioral complexity* means that measuring complexity will refer to the states and transitions the behaviors consist of. Finally, *functional complexity* means that measuring complexity will refer to the interaction of the design with its environment, or to its multiple functions.

5.3.3 Measures of Design Complexity

For a given context and aspect, a *measure of complexity* of designs will be a function, which applied to a design will result in a positive number, which will be interpreted as an estimation of the effort required to perform the process given by the context, in terms of the level given by the aspect. For example, a measure of complexity of a design for the context of manufacturing, in the aspect of structure could be the number of components of the design. The complexity of the door lock in Figure 3, by this measure, will be 4.

	Structure	**Behavior**	**Function**
Designing	- attribute nbr. - component nbr. - relation nbr. - attribute complexity	- attributes referred nbr. - components referred nbr. - relation referred nbr. - states nbr. - transitions nbr. - behaviors nbr.	- inputs nbr. - outputs nbr. - functions nbr. - input complexity - output complexity
Manufacturing	- attribute nbr. - component nbr. - relation nbr. - attribute complexity	NO	NO
Using	NO	- attributes referred nbr - components referred nbr. - relation referred nbr. - states nbr. - transitions nbr. - behaviors nbr.	- inputs nbr. - outputs nbr. - functions nbr. - input complexity - output complexity
Repairing	- attribute nbr. - component nbr. - relation nbr. - attribute complexity	NO	NO

TABLE 1. Possible elements for defining complexity measures for different context/aspect combinations.

Note that not all the combinations of context and aspect make sense for defining a measure of complexity for designs. Table 1 gives a list based on which measures of complexity can be defined for different combinations of context and aspect. The table cells containing ‘NO’ correspond to combinations of context and aspect which don’t make sense to be used for defining a measure of complexity. The elements in the cells can be combined to define

measures of complexity. Most of those elements refer to counts (denoted by “nbr.”). However there are some referring to “complexity”. As discussed in Chapter 2, this means that a measure of complexity can be defined recursively, along decompositions. Such a recursive definition requires a set of “base cases”, that is a set of designs for which the measure of complexity in question is postulated. For example, in the context of designing, for the aspect of structure, the complexity of a design may need to include an estimate of how complicated its shape is. Such a measure of complexity may characterize the effort required to describe the design (as the design itself is a description of a device). To define such a measure of complexity one must postulate a complexity measure for some elementary shapes, such as triangle, rectangle, circle (e.g., complexity of circle = 3, complexity of triangle = 6 and complexity of rectangle = 8, as given by how many numbers are required to represent each of them).

To illustrate the above discussion, let us define some example measures of complexity. We will apply the measures defined to the door lock in Figure 3.

5.3.3.1 Measures of Structural Complexity

A *measure of structural complexity* could be the number of *elementary components* of the design (elementary components are components that are not further decomposed in the

representation of the design). The complexity of the door lock as measured with this measure is 4. This measure would abstract away both the attributes of the design and its components as well as the local relations connecting those components.

A measure of complexity that would take into account both the number of components and the number of local relations could be defined as the *number of elementary components plus the number of relations*. The door lock as measured by its complexity measure will have complexity 4+3=7. While this measure of complexity includes the number of relations into its definition, it gives them the same "weight" in the computation as that of the elementary components.

Another way to combine the two counts (i.e., the count of elementary components, and the count of local relations) is to follow Boothroyd and Dewhurst's [1991] method to compute the *complexity factor* of a design. According to that, the complexity of the design would be computed by *multiplying the two counts and taking the square root of the result*. Note that, Boothroyd and Dewhurst, actually consider three counts (adding the number of 'component types' to the ones we are considering) for their formula and take the cube root of their product. Using this square root measure, the complexity of the door lock is $\sqrt{4 \cdot 3} = \sqrt{12} = 3.46$.

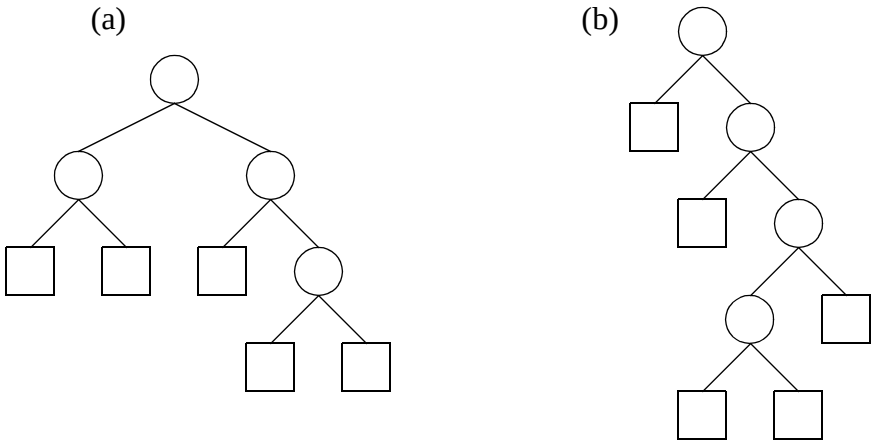


FIGURE 8. Two relation hierarchies which have the same complexity according to the measure proposed by Boothroyd and Dewhurst (circles represent relations, squares represent components)

Note that this measure is defined with the assumption that there are no higher level relations (i.e., relations between relations) in the structure. For this reason, while it does combine the number of components and the number of relations in the structure, it doesn't distinguish between designs with the same number of components and same number of relations, but using higher level relations. For example, if the two trees in Figure 8 repesented hierarchies of relations connecting components of two designs (e.g., a hierarchy of decompositions), then the measure of complexity defined above would yield for both of

them $\sqrt{4 \cdot 5} = \sqrt{20} = 4.47$. This is because both of them have the same number of relations and the same number of elementary components.

To be able to distinguish between the complexty of hierarchies of relations over a set of components, we propose that the complexity of such a hierarchy be measured by the *external path length* corresponding to the tree representing the hierarchy. By external path length of a tree we mean the sum of path lengths from the root of the tree to each of its leaves. Using this measure of complexity, the hierarchy (a) in Figure 8 has a complexity of 12, and the hierarchy (b) a complexity of 14. This difference in complexity is due to the fact that the measure defined assigns higher measure to hierarchies with higher systems of relations. If applied to the decompositional structure of a design, this measure of complexity could be interpreted as assigning higher values to "more critical" components, that is, components whose change would have a greater impact on the entire structure (e.g., the more levels of decompositions a component has "underneath", the more of its components, sub-components, and so all may be affected by its change).

Other measures of structural complexity for designing could be defined using the elements in Table 1. The actual elements that would go into defining such a measure and the way they would be combined depends on what is important for the purpose of the user.

5.3.3.2 Measures of Behavioral Complexity

A design may have more than one behavior. A *measure of behavioral complexity* of a design in the context of designing can be defined as a combination (e.g., sum, or maximum) of the complexity of its individual behaviors. Note here, that for our example, in the context of designing, the sum seems to be a better choice, because both of the behaviors have to be considered at design time. On the other hand, for use, taking the maximum complexity of the two behaviors (Open and Close) is a better choice, because at most one of those behaviors will be exhibited at any time.

Thus, we are faced with the problem of defining sample measures of complexity for a behavior. Let us recall that a behavior is represented as a sequence of state transitions. Each state transition consists of a partial representation of a state of the design, a condition and a transformation. Elements of these may be used in defining a measure of complexity for a behavior.

A simple measure of complexity for a behavior would be the number of state transitions it consists of. The behavioral complexity of the door lock as measured by this measure is 3. Note that, an equivalent way to measure behavioral complexity would be to count the number of states in a behavior, rather than the number of transitions.

Measuring complexity by counting the state transitions it consists of doesn't take into account "how complicated" the individual state transitions are. To take these into account the definition of complexity has to refer to either the initial and final states in each of the transitions, or the complexity of the transformation (or, possibly, both). We can thus define a complexity for each individual state transition and define the complexity of a behavior as the sum of the complexities of the transitions it consists of (note, that the complexity definition proposed in the previous paragraph would correspond to a complexity measure of one for considered for each transition).

Intuitively, the complexity of a transition depends on how much change the transition causes (e.g., how many state variables are modified) and on how complicated the transformation process is. Note that, the transformation in a state transition refers to another behavior, or to a function, or to some domain law. Thus, its complexity is defined by either the complexity of another behavior (recursively), or the complexity of a function (see subsection 5.3.3.3 below), or it can be defined (postulated) as a constant characterizing a physical principle).

Summing up the discussion, we propose that the behavioral complexity of a design be defined as the sum of the complexities $C(B_n)$ of all its top level behaviors B_n. With these notations, the complexity of a behavior is defined by:

$$C(B_n) = \sum_{t \in T_n} \mu(t) \cdot C(\theta(t)),$$

where $T_n = \langle t_1, t_2, t_3, \ldots, t_k \rangle$ the represents the sequence of state transitions in behavior B_n, $\mu(t)$ represents the number of changes (e.g., number of state variables affected), and $C(\theta(t))$ represented the complexity of the transformation $\theta(t)$. Note again that, $\theta(t)$ can be either a behavior, or a function, or a domain law.

Simpler behavioral measures, such as the state transition count defined above, can be derived by choosing appropriate values for $\mu(t)$ and $C(\theta(t))$ respectively.

Using the complete definition given above for $C(B_n)$, and assuming that the complexity of "Lever_Function" shown in Figure 5 is 1, and postulating that the complexity of transformations caused by a physical law is also 1, the complexity of the "Open_Behavior" represented in Figure 4 will be $1 \cdot (3 \cdot 1) + 1 \cdot 1 + 1 \cdot 1 = 5$. Note, that this measure

takes into account that the first state transition in the behavior is implemented by another behavior (“Lever_Cam_Transmit_Force”), which consists of a single state transition that affects three state variables and is implemented by the function “Lever_Function”.

5.3.3.3 Measures of Functional Complexity

A design may have more than one function. The functional complexity of a design in the context of designing may refer to either the number of functions the design has, or to how complicated its function(s) is (are). Thus, a general, *measure of functional complexity* of a design can be defined as a combination of the complexities of its (top level) functions.

Similar to the definition of behavioral complexity, this combination can be defined, for instance, as a sum of complexities of individual functions or as a maximum of complexities of individual functions. Again we can note that, in the context of designing, the sum seems to be a better choice, because all of the functions have to be considered at design time. On the other hand, for *use*, taking the maximum complexity of the functions is a better choice when at most one of the functions will be achieved at any time.

Thus, we need to give a sample definition for a the complexity of a function of a design. Remember that, we defined a function by an environment (a set of objects the design may

interact with), a set of interactions (a sequence of inputs-outputs pairs) and a mode of deployment. Thus a definition of the complexity measure for a function should take into account the complexity of the environment, the complexity of inputs and outputs as well as the complexity of the mode of deployment of the function.

The complexity of the environment can be defined as the number of the objects in the environment. This will essentially correspond to the number of objects the design will interact with while delivering the function considered. The complexity of the environment for the "Open" function represented in Figure 6 is 2, as the design interacts with two forces while delivering this function.

To define the complexity measure of an interaction we first need to define a complexity of each input-output pair of which the interaction consists, and then define a way to combine those complexities into a single measure. The complexity of an input-output pair can be defined as a combination of the complexity measures of the input and of the output. In our application domain (simple, schematic mechanisms), inputs can be forces which have a trajectory (e.g., linear, or circular) and outputs can be either forces or (object) states. For forces, a way to define the complexity measure may be by the complexity of (describing)

the trajectory associated. For states the complexity may be defined as the number of state variables needed to describe them.

Finally, the complexity of deployment could also be defined as the complexity of the behavior implementing the function.

In this stage of our research we define the complexity of a function as only depending on the complexity of the environment, that is the number of objects the design interacts with while delivering that function.

5.4 Structural, Behavioral and Functional Design Simplification

In this section we describe what we mean by structural, behavioral and functional simplification, respectively. Before starting our discussion however, let us note that in this dissertation we discuss behavioral and functional simplifications only at a conceptual level, by defining them and pointing out what specific issues they raise. The working system presented in the next chapter was built and tested on structural simplification problems only.

A *design simplification* is a binary relation connecting two designs, a simpler one and a more complicated one. Every design simplification has assigned to it a *point of view*, that is, a context, an aspect and a measure (of complexity), an *explanation* and a *set of relevant elements*. A design simplification has to satisfy the "simplification condition", that is, if the measure (corresponding to the point of view of the simplification) is applied to the two designs involved, for the corresponding context and aspect, the value obtained for the "simpler design" will be (strictly) less than the one obtained for the more complicated one.

The set of relevant elements associated with a design simplification consists of elements in the representation of the aspect corresponding to the point of view of simplification, which were used or affected by the simplification. For example, for a design simplification, in the context of designing, for the aspect of structure and with the measure defined by counting the components of the design, the set of relevant elements may consist of objects, relations and attributes of the designs involved. These elements may be referred to in conditions that needed to be satisfied for the simplification to be "realizable", or in the operations which were applied to transform the more complicated design into the simpler one.

The explanation of a design simplification is a description of the process that has been applied to the more complicated object to obtain the simpler one. Such a process is a

sequence of transformations. Each transformation consists of a partial description of two designs (one before the transformation has been applied, and one after the transformation has been applied), a predicate describing the conditions that had to be satisfied in order for the transformation to be applicable, and the operation that describes the transformation.

In the rest of this section we will give examples of design simplifications for each of the three aspects considered. The examples will again be drawn from the domain of door lock designs.

5.4.1 A Structural Simplification

A structural simplification of a design refers to either physical attributes of the design, or to its structural composition. For instance, an object with the shape attribute “circle” may be considered simpler than an object with the shape attribute “oval”. With respect to structural composition, a design can be simpler than another design if it has fewer components, fewer relations between components, or simpler relations between components. Here by “simpler relations” we mean relations with fewer arguments.

Let us consider the door lock illustrated in Figure 9. Comparing it with the door lock in Figure 3, it appears to have a simpler structure, because (while it has the same number of components, the same number of relations) the shapes of the roller and the plain bolt are simpler that those of the cam and the bolt with a hooked shape.

To represent this relation between the two designs as a design simplification we first need to consider a measure of complexity that is able to capture the difference described above. The measure of structural complexity based on the external path length of the decompositional structure, defined in section 5.3.3.1, would yield for both of the designs a complexity of 4. To capture the complexity introduced by attributes and attribute values, we must extend this measure.

The structural simplification connecting “Door Lock 1” and “Door Lock 2” is represented in Figure 10.

5.4.2 A Behavioral Simplification

A behavioral simplification refers to either the complexity of the partial state description, or the number of states, or the number of transitions in a behavior. By the complexity of a

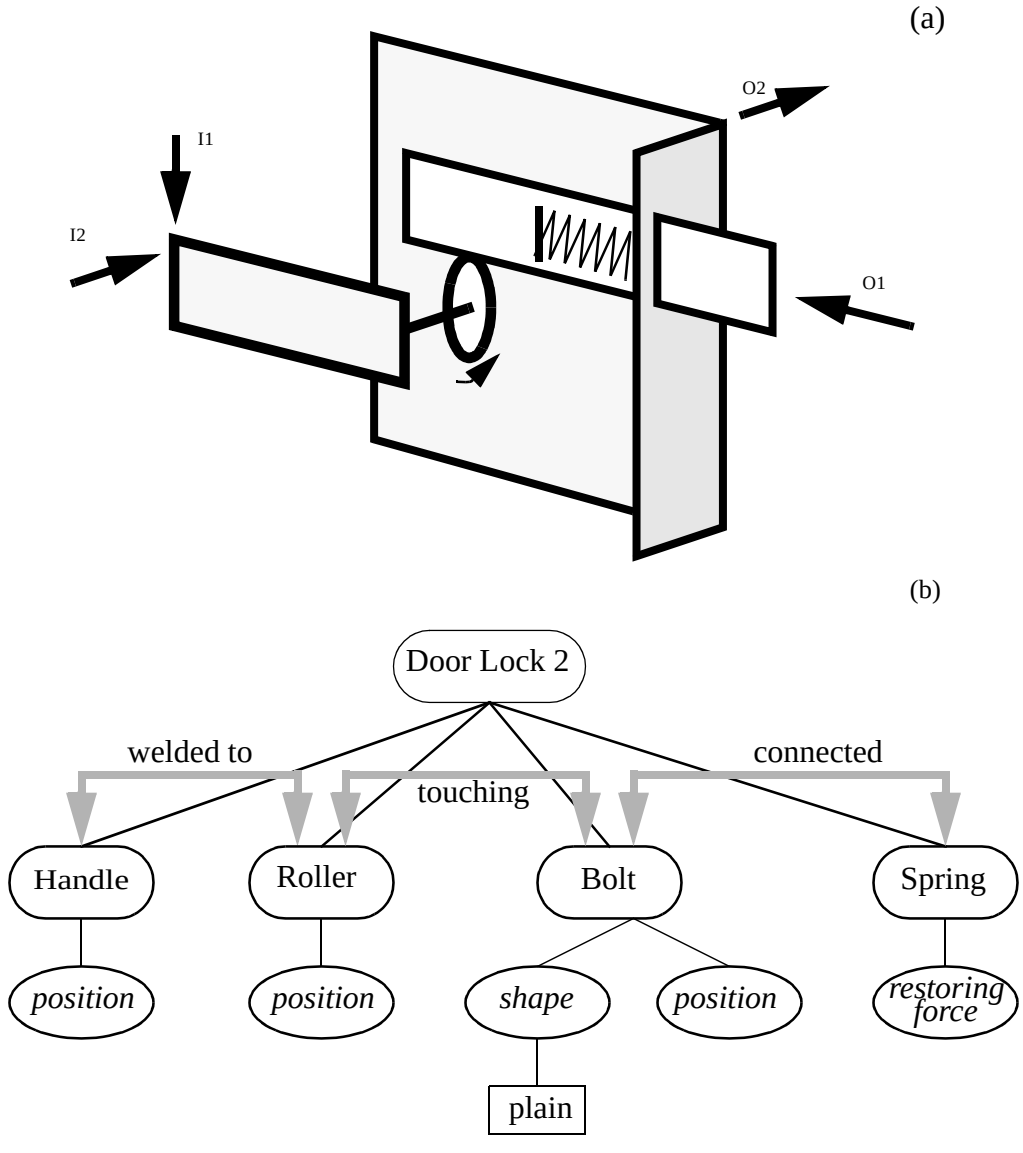


FIGURE 9. Door lock implementing the cam mechanism using a roller: (a) schematic and (b) structural representation

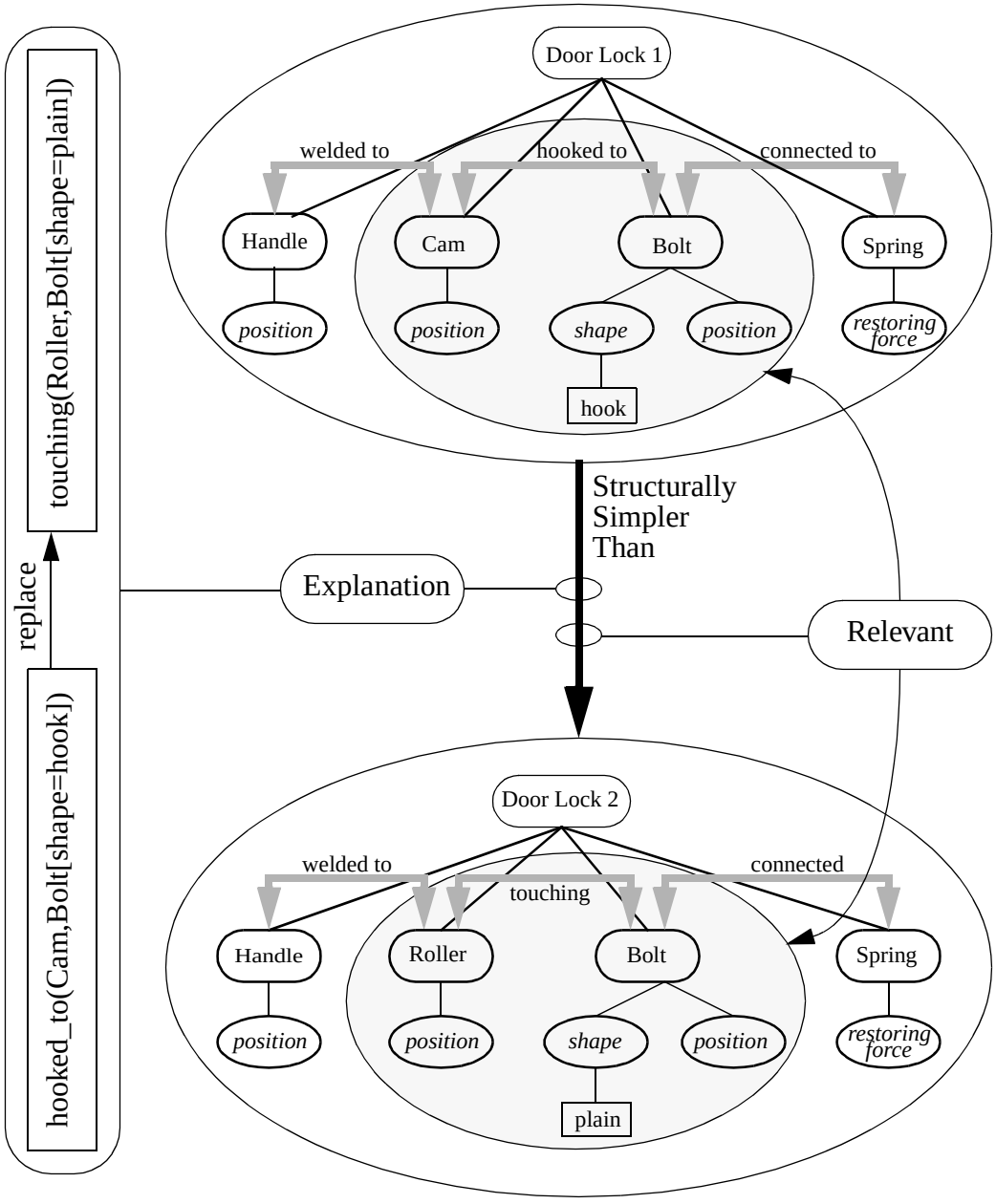


FIGURE 10. Simplification "Door Lock 1" to "Door Lock 2"

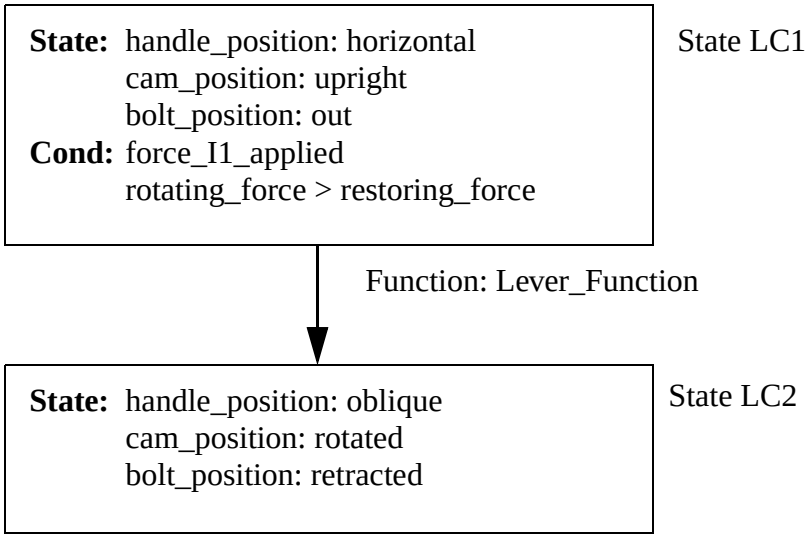


FIGURE 13. State transition graph for the 'Lever_Lever_Wedge_Transmit_Force' behavior referred to by the 'Open' behavior represented in Figure 4.

partial state description we mean a measure depending on the number of elements (attributes, objects and relations) in the partial state description.

To illustrate this with an example, let us consider the door lock design illustrated in Figure 11. This door lock uses two levers, a wedge and an L-shaped bolt to implement the door lock functions. Figures 12 and 13 represent the behavior of this door lock.

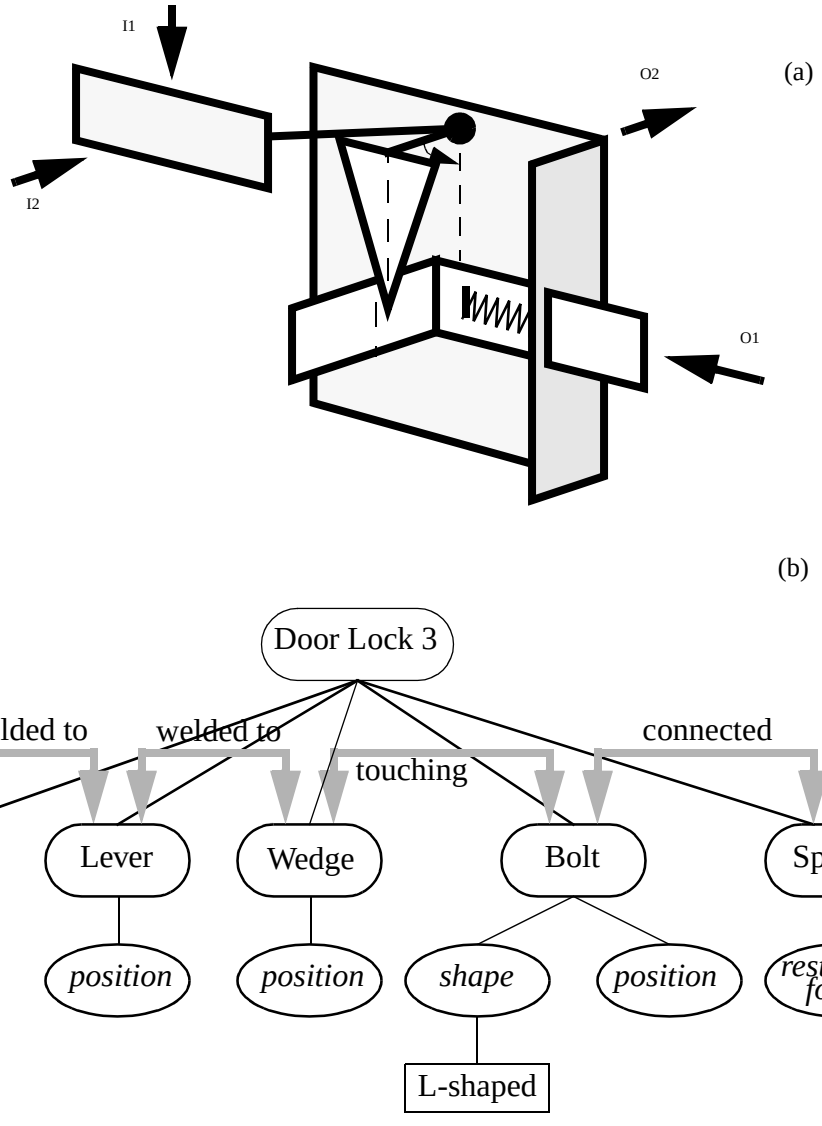


FIGURE 11. Door lock using a combination of two levers, a wedge and an L-shaped bolt: (a) schematic and (b) structural representation

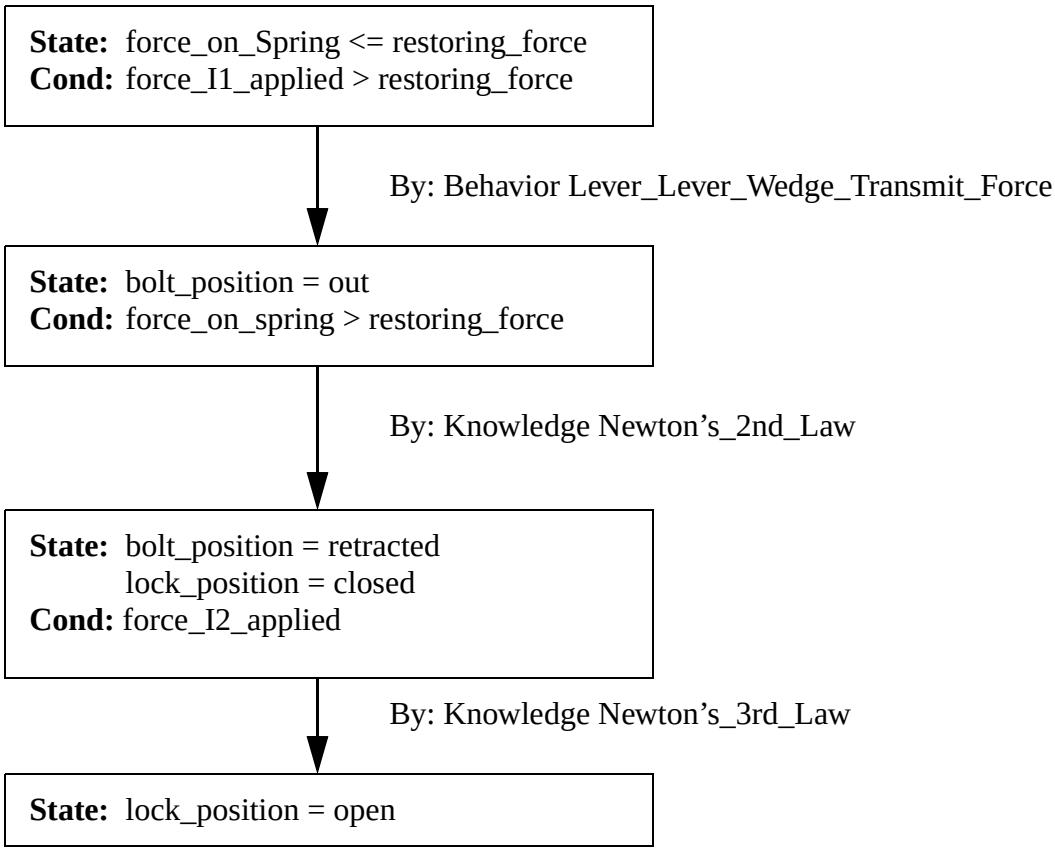


FIGURE 12. State transition graph for the top level behavior of the Door Lock 2.

5.4.3 A Functional Simplification

A functional simplification may refer to either the "complexity" of using a designed object, or to its possible multiple functions. An object may be "functionally simpler" than another object if it is easier to use. For example, if it requires fewer inputs or simpler inputs (e.g., force applied in a linear rather than curved motion). On the other hand, an object can also be "functionally simpler" than another object if it has fewer functions (i.e., it can be used for fewer purposes).

As an example, consider the door lock in Figure 14. This design uses a wedge and an L-shaped bolt to implement the door lock functions. The function of this door lock is represented in Figure 15. If we measure functional complexity by the number of inputs (i.e., the complexity of the environment), we can say that the door lock in Figure 14 is functionally simpler than the door lock in Figure 11.

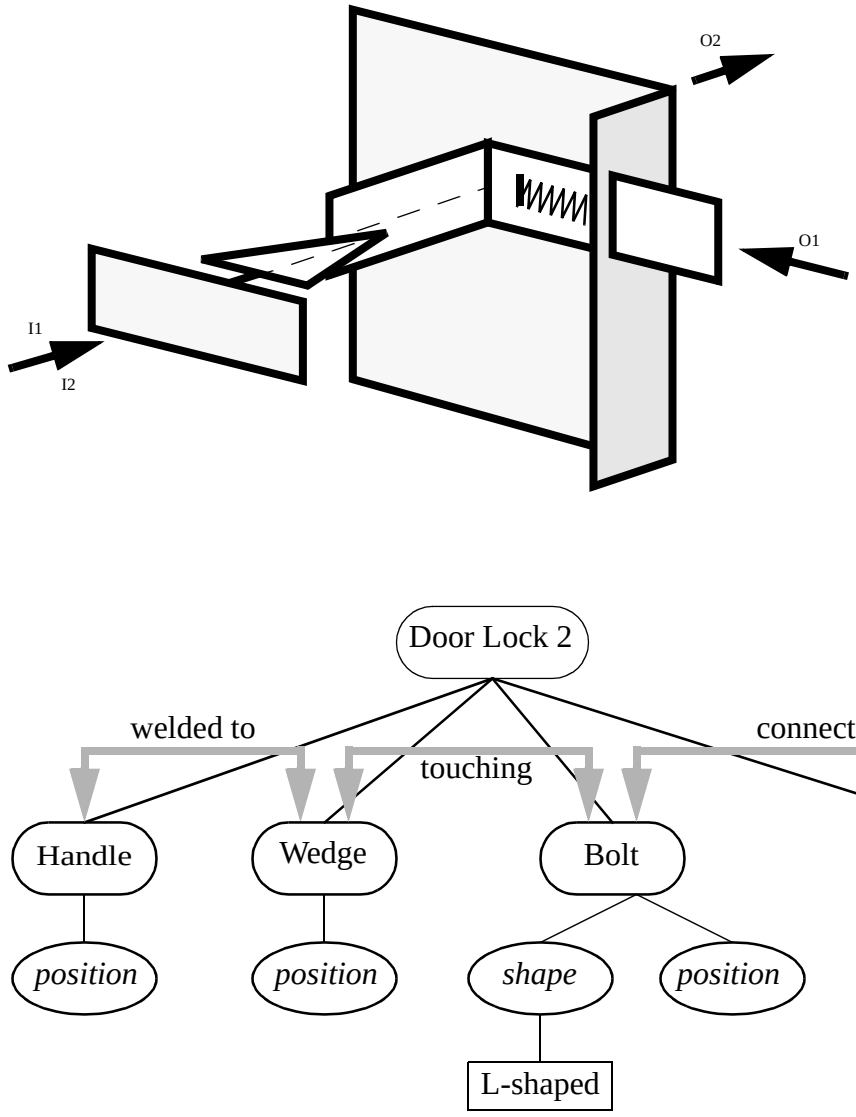


FIGURE 14. Door lock using a combination of a wedge and an L-shaped bolt: (a) schematic and (b) structural representation

```
Function: Open
 Environment:
    - input I1 applied to the handle
       Interaction:
    - (force_I1_applied > restoring_force)
              => retract_bolt and open_door
 By (deployment):
    - Open_Behavior
```

FIGURE 15. The Open function of the door lock in Figure 14.

CHAPTER 6

Implementation

In this chapter we present the implementation of the computer system that was used to demonstrate our approach to solving simplification problems. The first section gives a general description of the system architecture and provides an explanation of why CLIPS was the language of choice for our implementation. The second section presents the representation used by the system. Section three describes the abstraction mechanism used, while the last section describes the implementation of the analogical reasoning mechanism for simplification.

6.1 The System

The system was implemented in the CLIPS language [CLIPS 1993]. We chose CLIPS for the following reasons:

- it supports rule-based programming, using a powerful pattern matching algorithm, called Rete (also used in the implementation of the OPS-5 language);
- it supports object-oriented programming;
- it supports procedural programming;
- it implements a set of powerful query operations;
- it allows easy interfacing with other programming languages (e.g., C);
- a great variety of additional tools are available (e.g., a GUI builder).

These features of the language allowed an object-oriented design of the system and quick prototyping.

To us the major disadvantage of the language was the lack of a Lisp-like list data type. Due to this, the manipulation of nested lists and symbolic processing, such as evaluation of lists as function calls, cannot be directly implemented in CLIPS. To overcome this we implemented a ‘List’ class which provides the entire range of list manipulation methods available in most of the Lisp implementations, including evaluation of lists. This class was implemented in CLIPS.

The current implementation uses a text interface which is based on an interface library we implemented. This library contains functions for various kinds of input and output operations as well as for menu definitions. The library was implemented completely in CLIPS and can be used with any CLIPS program. We must note here that there is a CLIPS implementation, called wxCLIPS, which allows the development of graphical user interfaces (GUI's) for CLIPS programs. We plan to extend our system in the future with a GUI implemented in wxCLIPS.

The architecture of the system is shown in Figure 1. The system consists of a *database of known simplifications*, an *interface* module, a *data management* module, a *simplifier* and a *simplification abstraction* module.

6.1.1 The Database of Known Simplifications

The database of known simplifications stores a collection of CLIPS instances representing simplifications and the objects that are connected by those simplifications. The entire database is stored on disk, possibly in several files. To solve simplification problems all or part of the data base has to be loaded into the memory.

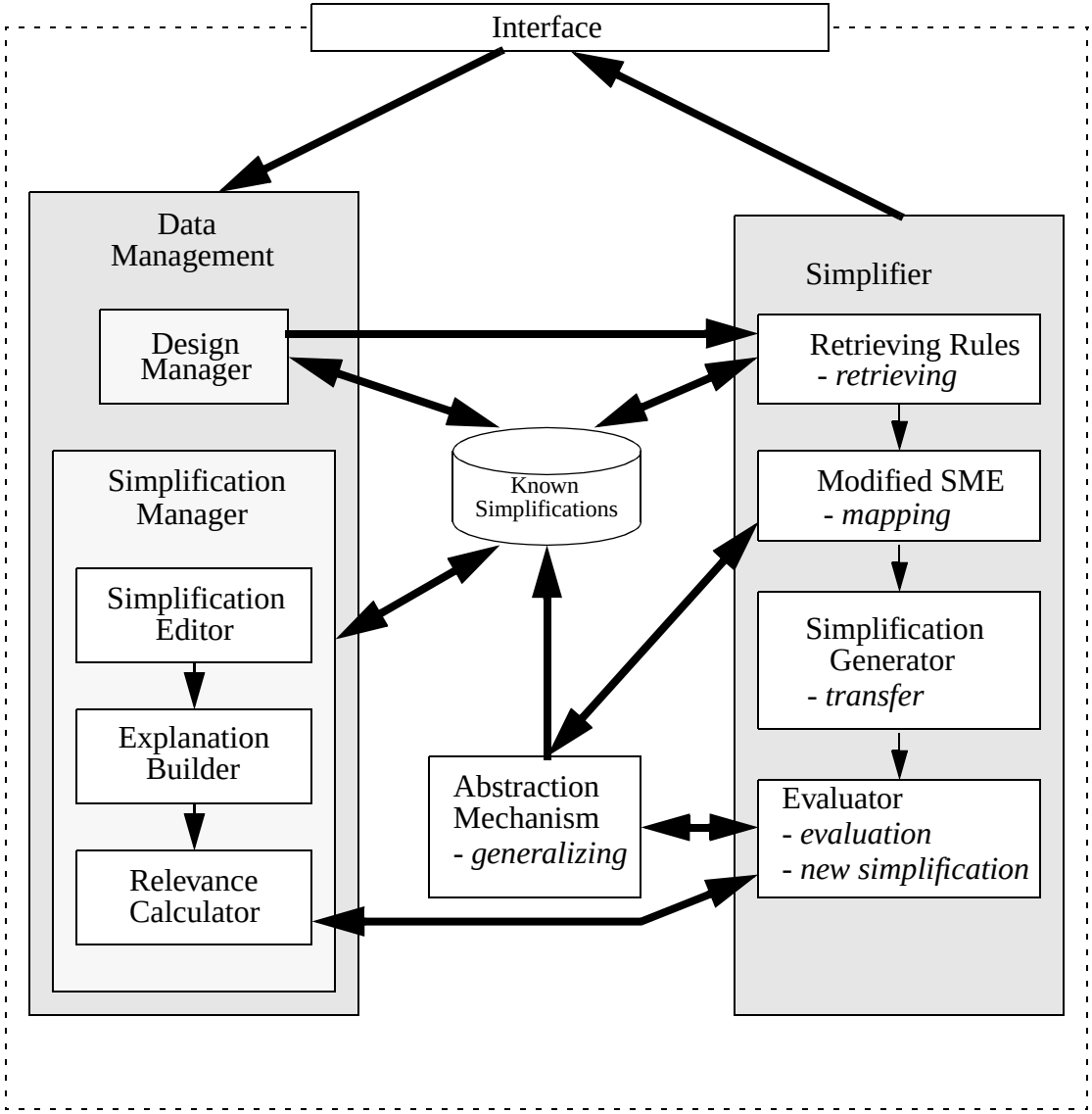


FIGURE 1. Architecture of the simplification system

We currently partition the data base based on the different application domains our system accepts. Those application domains are mathematics, programs and mechanical design. Each partition is stored in a different file. We made this design decision for our implementation to be able to perform experiments with “within domain” and “across domain” analogical reasoning. Any number of partitions corresponding to different application domains can be loaded into memory at the same time.

6.1.2 The Interface Module

The interface module allows the user to interact with the system. It uses a combined *command-line/menu interface*. This means that the user may type in commands at the system prompter, but entering the empty command (just carriage return) or an erroneous command will cause a menu of the available commands to be displayed. The sample runs shown in Appendix B illustrate the use of the interface.

6.1.3 The Data Management Module

The data management module allows the creation, editing, saving and loading of objects and simplifications. It consists of two submodules: the *design manager* and the *simplification manager.*

6.1.3.1 The Design Manager

The design manager submodule allows the creation and editing of objects represented in *external format* as well as saving and loading them *internal format*. The external representation of an object has the structure shown in Figure 2. Appendix A presents the external representation of several designs. The internal representation is in the form of CLIPS instances. While CLIPS instance files are text files (just like the files containing external representations), they use a much more compact representation of objects than our external representation and are saved and loaded efficiently by the corresponding CLIPS functions.

In the current implementation, creating and editing of objects is done using a standard text editor (currently Emacs running on UNIX) which is called by the system. When a new object is created a template file is loaded into the text editor. This template file contains the syntactical structures of all the elements which may be needed for externally representing an objects.

Note that for some domains, the general external representation defined by us may not be natural. For instance, representing mathematical expressions as objects (with attributes,

```
(RootDesign <root-design-name>
   (Attributes
        (Attribute <attribute-name> <default-value>)
        ... more attributes
   )
   (Components
      (Design <design-name> ... design description )
      ... more component designs
   )
   (Relations
      (Relation <relation-name> <list-of-components> )
      ...
   )
   (Behaviors
      (Process <process-name>
        (ProcStep
           (ObjectState <state-description> )
           (Condition   <condition-description> )
           (Apply       <action-description> )
           (ObjectState <state-description> )
        ) ... more process steps
      ... more behaviors (process descriptions)
   )
   (Uses
      (Process <process-description> )
      ... more process description
   )
)
```

FIGURE 2. The structure of the external representation of an object

components and relations, plus behavior and function) is very counterintuitive. For this reason, if a domain which is to be included into the system, has a well established and widely used system of representation, a module for interpreting that representation and

building the appropriate internal representations needs to be added to the system. Our system currently has a module which is able to read in arithmetical expressions in prefixed "Polish notation" and build the appropriate internal representations.

Besides creating and editing objects represented in external format, the simplification manager module implements routines for saving and loading objects in internal (CLIPS instance) format. Saving can be done selectively, that is, the user can select to save all or just some of the objects currently represented in the memory.

6.1.3.2 The Simplification Manager

The simplification manager allows for creating new simplifications, saving simplifications to a simplification database and loading simplifications from a simplification database.

Creating a new simplification consists of *editing the simplification*, *explaining the simplification* and *performing the relevance calculation*.

Editing a simplification can be done either by calling an external editor (currently Emacs running under UNIX), or interactively. Using the editor for simplifications is similar to the way of editing objects, that is, initially the editor loads a template describing the syntactical structures of the elements needed to represent simplifications (objects and explana-

```
(Simpler
   <simpler-design-description>
   <less-simple-design-description>
   (Explain
      (Difference (replaced <component-1> <component-2>))
      ... more difference descriptions
   )
)
```

Or, if the explanation is by simplification process:

```
(Simpler
   <simpler-design-description>
   <less-simple-design-description>
   (Explain
      (Process <process-description> )
   )
)
```

FIGURE 3. The structure of the external representation of a simplification

tion). This template can then be edited (Figure 3 shows the structure of the template file loaded into the editor when a new simplification is to be built).

During the interactive creation of a simplification the system first prompts the user to select the objects which will be involved in the simplification to be created. Next, the system asks the user to select the *explanation type* (by difference or by simplification process)

and the corresponding *explanation specification* (differences or sequences of transformation).

When a new simplification is created, the complexity of the two objects involved are compared, according to the complexity measure currently in use. If the complexities of the two objects are not in the correct relation (i.e., the complexity of the “simpler” object is greater or equal than the complexity of the “more complicated” one) the simplification is not generated.

After a new simplification is created the relevance calculation (see Chapter 4) is performed automatically.

The simplification manager submodule also implements routines for loading and saving simplifications represented in internal format, that is, in form of CLIPS instances.

6.1.4 The Simplifier Module

The simplifier module is the part of the system that solves simplification problems. It consists of four submodules: the set of *Retrieving Rules*, the *Modified SME*, the *Simplification Generator* and the *Evaluator.*

6.1.4.1 The Retrieving Rules Submodule

The Retrieving Rules submodule consists of a set of CLIPS rules implementing the retrieval phase of the analogical reasoning process. These rules are divided into two groups: *rules for selecting objects involved in simplifications* (source objects) with the same point of view (context, aspect and measure) as the simplification problem, and *rules for building match hypotheses*.

The rules for building match hypotheses look for relations in the source objects that: are *identical* (same name) to some relation in the object to be simplified, or have a *common super-relation* with some relation in the object to be simplified, or have the *same signature* (same number and type of arguments) with some relation in the object to be simplified. Each rule, when fired will create a match hypothesis object, that will be used as an input by the SME. Figure 4 shows the rules currently used by the system to build match hypotheses.

6.1.4.2 The Modified Structure Mapping Engine

The Modified SME submodule is a CLIPS implementation of the SME with two significant modifications: a) the match hypotheses are input to the SME as they are generated by

```
; Building a match for relations with the same name

 (defrule NameMatch
    (retrieved match ?design ?oldDesign ?simRel)
    ?locRelNew <= (object (is-a Relation)
                          (Name?n)
                          (Root?design)
                  )
    ?locRelOld <=  (object (is-a Relation)
                          (Name?n)
                          (Root?oldDesign)
                  )
    (test (neq ?locRelNew ?locRelOld))
    (test (IsRelevantp ?locRelOld ?simRel))
 =>
    (MatchHypothesis ?locRelNew ?locRelOld ?simRel)
 )

; Building a match for relations with a common ancestor

 (defrule CommonAncestorMatch
    (retrieve match ?design ?oldDesign ?simRel)
    ?locRelNew <= (object (is-a Relation)
                          (Root   ?design)
                          (Definition ?relDefNew)
                  )
    ?locRelOld <= (object (is-a Relation)
                          (Root   ?oldDesign)
                          (Definition ?relDefOld)
          )

    (test (neq ?locRelNew ?locRelOld))
    (test (IsRelevantp ?locRelOld ?simRel))
    (test (neq (CommonSupRel ?relDefNew ?relDefOld) nil))
=>
    (MatchHypothesis ?locRelNew ?locRelOld ?simRel)
)
```

FIGURE 4. CLIPS rules for building match hypotheses

```
; Building a match for relations of the same arity

 (defrule ArityMatch
    (retrieve match ?design ?oldDesign ?simRel)
    ?locRelNew <= (object (is-a Relation)
                          (Arity?n)
                          (Root?design)
                  )
    ?locRelOld <= (object (is-a Relation)
                          (Arity?n)
                          (Root?oldDesign)
                  )
    (test (neq ?locRelNew ?locRelOld))
    (test (IsRelevantp ?locRelOld ?simRel))
=>
 (MatchHypothesis ?locRelNew ?locRelOld ?simRel)
)
```

FIGURE 4 continued. CLIPS rules for building match hypotheses

the retrieving rules, and b) building the global mappings (maximal consistent systems of matches) is restricted to the relevant elements of the source object. Our implementation of the SME consists of a set of CLIPS routines based on the description in [Gentner 1983]. A high level description of the mapping phase as implemented my our modified SME is given in Figure 5.

The Modified SME will produce a best global mapping which then will be used by the Simplification Generator submodule to build candidate simplifications.

```
Mapping Algorithm
 Input:  M - a set of match hypotheses
 Output: G - a set of global mappings
begin
 - Build a partition L of M consisting of locally
   consistent subsets of M[1].
 - Propagate local inconsistencies up the arguments
   structure of relevant match hypotheses (i.e., match
   hypotheses which involve a relevant element) to rule
   out match hypotheses that are inconsistent due to the
   inconsistency of their arguments
 - Combine consistent sets of match hypotheses to obtain
   the set of global mappings
end
```

1. A set of match hypotheses is locally consistent if it represents an 1:1 association between elements.

FIGURE 5. A description of the mapping phase as implemented by the modified SME

6.1.4.3 The Simplification Generator

The Simplification Generator submodule takes as input the best global mapping between the source and the target produced by the modified SME and will produce candidate simplifications. This global mapping takes the form of a list of matches. Each match is a pair of elements, the first one being from the source and the second one from the target.

The operation of the simplification generator is described in Figure 6.

```
Simplification Knowledge Transfer
 Input:  g - global mapping
 Output: O - an object resulting from transferring the
             simplification knowledge
begin
 if source explained by difference then
    - interpret g as a substitution and
      apply it to the target to generate a new object O,
    - complete O by assigning values to the elements
      not bound by g[1]
 else (source explained by process)
    - interpret g as a substitution
    - create O as a copy of the target
    - for each step in the explanation do
        - assign values to the still unbound elements
        - apply the step obtained to modify O
      end (do)
 end (if)
end
```

1. Note that this assignment may be done in more than one way

FIGURE 6. Generation of a candidate simplifications from a global mapping

6.1.4.4 The Evaluator

The Evaluator submodule evaluates the candidate simplifications from two points of view. For a candidate simplification generated, it first checks if it is indeed a simplification. It does this by computing the complexities of the two objects involved according to the com-

plexity measure currently in use. If the two complexities are not in the correct relation, then the candidate simplification is discarded.

The second evaluation refers to comparing the "quality" of the candidate simplification to that of the currently best simplification obtained. This is achieved by running the Simplification Generator and the Evaluator interleaved. At any time during the generation of candidate simplification a "best candidate simplification" is stored. The quality of a candidate simplification is measured by the complexity of the "more complicated object" involved in it. Whenever a new candidate simplification is generated, if it passes the first phase of evaluation, its quality is immediately compared to that of the currently best one and if it is better, it will become the new currently best simplification. Note, that this second phase of evaluation does not require any extra computation, except for the comparison of two numbers, since the complexities have already been computed in the first phase.

6.1.5 The Simplification Abstraction Module

The Simplification Abstraction Module implements a mechanism for building an abstraction over two given simplifications. It is called when a new simplification is generated and it is sufficiently significant from the source simplification used for generating it. Currently our system only performs a *relevance-based abstraction*. This means that it will create a

```
Abstraction
  Input:  S(A,B) - simplification (A simpler than B)
  Output: AS - simplification
begin
  - Find the minimal part B that contains all the
    relevant portions (by propagation up along the
    structural representation)
  - Build an object b which is a copy of this minimal part
  - Build a from A by removing all the elements that
    are not part of b
  - Generate a simplification AS(a,b) with the an
    explanation identical to that of S(A,B)
end
```

FIGURE 7. Performing relevance-based abstraction

new simplification in which the elements which are not relevant have been removed from the objects involved. The operation of the simplification abstraction as implemented in our system is described in Figure 7. While this is a very simple way of building abstractions it is very useful for two reasons: a) it will create simplifications that involve simpler objects, which will be easier to match, and b) it allows the extraction of some simplification rules.

6.2 Representation

To represent designs and simplifications we took an object-oriented approach, defining a set of CLIPS classes. Besides the obvious advantages of object-oriented design and programming, this allowed us to use features offered by CLOOS, the CLIPS object oriented system [CLIPS, 1993], such as pattern matching on objects and queries on sets of objects.

Figure 8 presents the class hierarchy defined in our implementation. The thick gray arrows represent the “subclass of” (or “is-a”) relations between classes, while the thin black arrows represent the “has part” (i.e., the inverse of “part-of”) relation. In Appendix C we give a complete printout of the class definitions in this hierarchy. The class hierarchy should be extended every time a new application domain is added to the system.

To add a new application domain one needs to add a new class defining the objects of the domain, by deriving it from the “`Object`” class. In order for such a class to be properly integrated into the system one needs to overload a set of message handlers and functions for performing domain-specific input and output. We must note here that, for adding a new

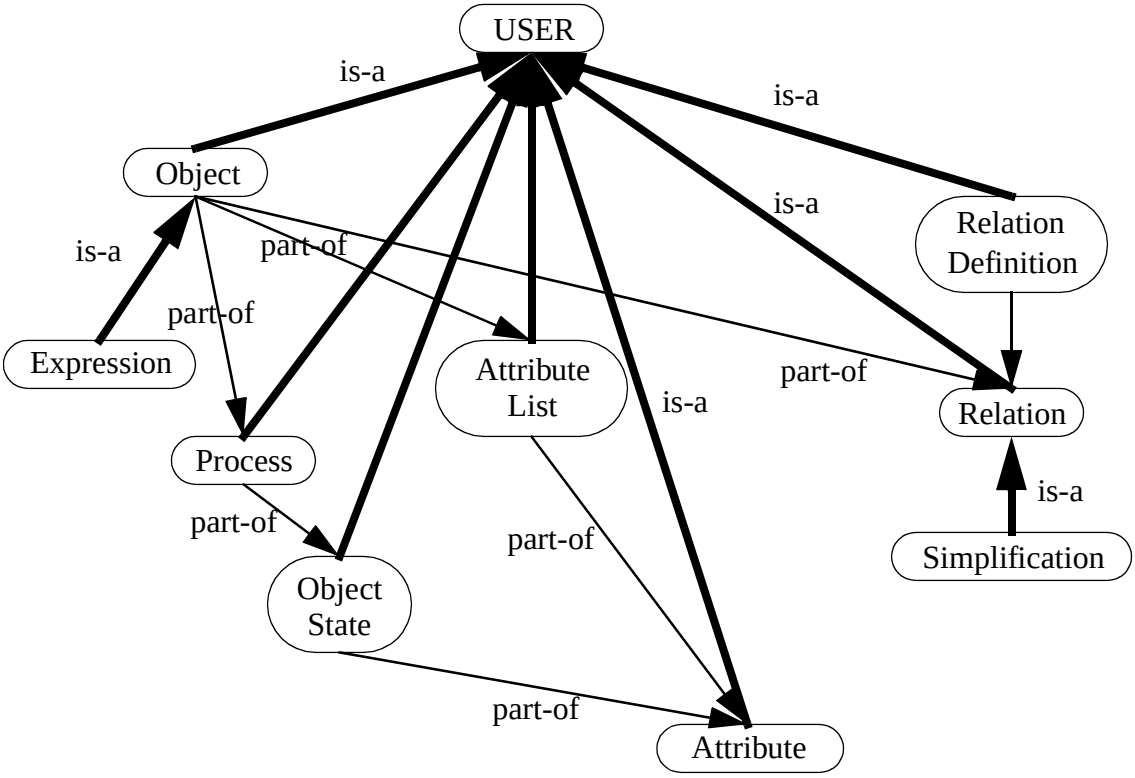


FIGURE 8. The hierarchy of classes defined in the design simplification system

application domain, the language of the system has to be extended by the definitions of relations and attributes specific to that domain. This can be done by either manually editing the default language file (called “default.lng”), or by creating a new language file and loading it into the system.

6.3 Implementation of the Abstraction Mechanism

The abstraction mechanism is used to produce a new object (or relation) representation by removing (abstracting from) some of the details in the representation of a given object (or relation). Our current implementation allows abstracting in objects by removing components, relations, attributes or attribute values, or in simplification relations, by applying abstraction to the objects involved, or by removing elements in the explanation (e.g., components, explanation process steps). Note here that when an abstraction over a given simplification is performed by removing elements in the explanation, the set of relevant elements needs to be recomputed for the abstract simplification. In our implementation this is automatically performed every time a new simplification is created.

In our system abstraction can be applied to an object, to make the analogical reasoning process more efficient by only considering the relevant elements of known simplifications, or to a set of simplifications to produce an abstract simplification (corresponding to generic simplification rule or to a simplification principle).

When applying abstraction to an object the abstraction process is guided by the problem solving goal, that is, simplification. This is done by only considering elements that are rel-

evant to some simplification (i.e., which are in the set of relevant elements attached to some simplification). We implemented this by including into the left hand side (i.e., the “if” side) of the CLIPS rules used for building the match hypotheses, conditions for testing relevance of the elements for which a match is trying to be hypothesized. The testing of this condition is efficient because the relevance computation is always done at the time of creating a new simplification. The advantage of applying this abstraction is that it results in pruning the from the database all the objects that are not relevant to a simplification of the type (i.e., point of view) searched for.

The only way our system can currently apply abstraction to simplifications is to remove irrelevant elements for a given simplification. This is done by applying abstraction to the two objects involved in the simplification. Note that there is no need to generate a new explanation because every element referred to in the old explanation in the set of relevant elements and thus will not be removed by the abstraction process. This process is performed by performing a CLIPS query on all the relevant elements of the given simplification. This process is efficient because it only uses value matching on two slots of CLIPS instances representing objects.

6.4 Implementation of the Analogical Reasoning Mechanism

Our analogical reasoning is essentially an implementation of Falkenheiner's Structure Mapping Engine (SME) [Falkenheiner et al. 1993]. in this implementation we used an object-oriented approach (as opposed to the purely procedural approach of the original implementation). For this we defined CLIPS classes for representing match hypotheses and mappings. The implementation of SME is by a set of CLIPS rules which use match conditions formulated in terms of objects of class match hypothesis and mapping. By this the operation of SME is described more clearly.

CHAPTER 7

System Demonstration

In this chapter we present a set of sample problems we have solved using the system described in the previous chapter. The goal of presenting these problems is on one hand, to illustrate the operation of the system, and on the other hand to demonstrate the breadth of the system. Each of the sections in this chapter presents one sample problem

7.1 Simplification of an Arithmetic Expression

Simplification of arithmetic expressions is one of the application domains on which our system was tested. Among others, our decision to use arithmetic expressions to demonstrate the system was based on the following considerations:

1. Since there are a significant number of transformation rules for arithmetic expressions, it is easy to build large simplification databases that allow the testing of different aspects of the system's operation;
2. The rules for forming arithmetic expressions are flexible enough to allow the building of interesting simplification problems;
3. The experience gained by performing simplifications on well-formed structures can be used in other domains, such as software or hardware.

In this section we will describe the simplification of an arithmetic expression as performed by our system.

7.1.1 The Sample Problem and Issued Raised

We presented the system with the following simplification problem:

Reduce the number of elements used to represent the following arithmetic expression:

$$\frac{x}{xy + x(z + uv)}$$

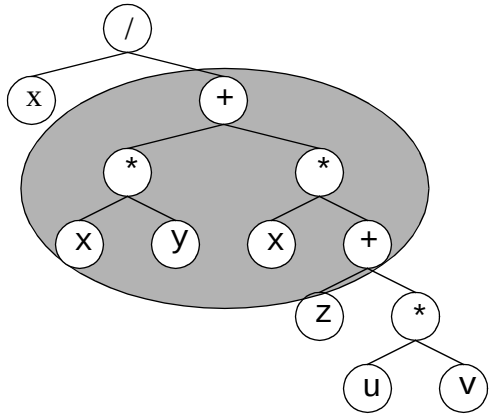


FIGURE 1. Structure of the arithmetic expression to be simplified. The shaded portion specifies where factoring can be applied.

The structure of this expression is shown in Figure 1.

For an expert in manipulating arithmetic expressions, it should be clear that this expression is equivalent with the following simpler expression:

$$\frac{1}{y + (z + uv)}$$

By showing how our system solves this problem we will demonstrate how the following issues are addressed:

- how to retrieve the best matching source simplification in the simplification database,

- how to apply the retrieved simplification to the current problem (target),
- how to decide whether the result produced is a valid simplification, and
- how to decide whether further simplifications could be applied.

In the following subsection we will describe how our system solves the simplification problem proposed.

7.1.2 Operation of the System

To Solve the simplification problem proposed above the system was loaded with a simplification database containing a set of sample simplifications. This set did *not* contain a simplification corresponding to the general distributivity property of multiplication with respect to addition (i.e., for all x,y,z $\in \Re$, $xy + xz = x(y + z)$). It contained, however, examples of factoring out a term from a sum of subexpressions in a more complicated expression. One such simplification example was the following:

$$3(1-2a) + \{(a-2b)[c(d+e)+z]\}$$

SIMPLER THAN

$$3(1-2a) + [(a-2b)\cdot c(d+e) + (a-2b)\cdot z]$$

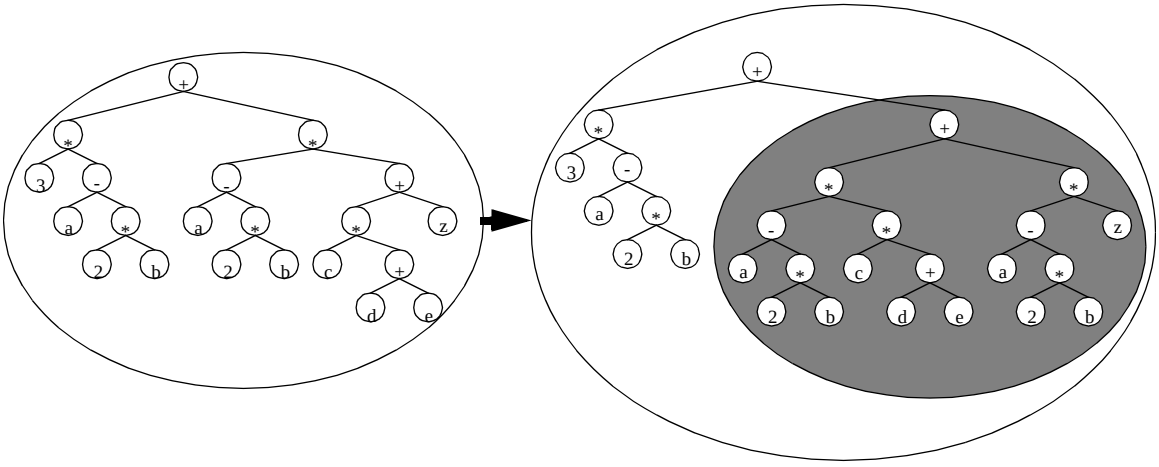


FIGURE 2. Representation of a simplification of an arithmetic expression

Figure 2 illustrates the structure of this simplification. When this simplification was added to the database, it was explained by the difference between the two expressions, that is by the replacement of the subexpression $(a - 2b)c(d + e) + (a - 2b)z$ by the subexpression $(a - 2b)[c(d + e) + z]$. Due to this, the database did not contain any explicit information about the conditions under which the simplification was performed, or about what was relevant in performing this simplification. As a consequence, as described in Chapter 4, the relevance calculation will result in designating the shaded portion shown in Figure 2

as relevant to the simplification. This relevant portion will be used in the retrieving and mapping phases of the analogical problem solving process.

The first difficulty raised by the simplification problem presented to the system is to retrieve the simplification shown in Figure 2 as the best source analog, or at least to find it sufficiently similar to the problem to be used for building a global mapping. The difficulty of this is a consequence of the fact that an arithmetic expression may have a very complicated structure which is built using only four operators, corresponding to the four arithmetic operations.

The first step in retrieving a source analog is to generate match hypotheses associating relations (corresponding to operators) in the target with relations in the relevant portions of known simplifications (i.e., simplifications stored in the database). For arithmetic expressions, it is the case that practically every relation in the target can be matched with any relevant relation in a simplification. This is true because any two relations (operators) have either have the same name or are descendants of the same super-relation (e.g., both '-' and '+' are 'additive operations'), and definitely have the same arity (number of operands).

Generating matches could be restricted to only considering relations with the same names (e.g., only match a '+' to another '+'). In this case, however, the system would not be able to discover any analogies based solely on structural similarities, or cross-domain analogies. The problem of generating matches is further complicated by the fact that it is possible that only a (small) part of the target can be simplified by analogy with some known simplification. For example, in our example only the shaded portion shown in Figure 1 can be simplified by analogy with the simplification shown in Figure 2.

After generating the match hypotheses, the system builds a set of global mappings of the target onto the source and then selects the best global mapping(s) to transfer the simplification knowledge. Once the set of matches was generated the building of global mappings is not difficult (see [Falkenheiner et al., 1993]). The difficulty arises in the evaluation of the quality of the global mappings generated. This is due to the fact that, a large (i.e., consisting of many match hypotheses) but weak (i.e., consisting of low score match hypotheses) global mapping may score higher than a small but strong one.

To overcome these difficulties, our system uses a score evaluation scheme that combines structural complexity measurement (external path length) with type-dependent match

weighting (i.e., assigning different weights to different types of matches). Due to this scheme our system correctly selects the best global matching for our problem.

From the global mapping the system builds a substitution and then it transfers the simplification knowledge to the target using this substitution. This results in the simpler expression presented above.

After the simpler expression was generated, a corresponding simplification is built and added to the simplification data base for future use.

Note, that currently, our system does not implement any mechanism for evaluating if the new simplification is worth to be stored. Thus, when a solution to a simplification problem is found, the user is prompted to decide whether the new simplification will be added to the data base or not.

7.2 Simplification of the Personal Fax Design

The goal of the example presented in this section is to demonstrate how the system handles issues raised by cross-domain analogy. The target domain of the example is mechanical design, while the source domain is the domain of arithmetic expressions. In addition to

demonstrating how the simplification method works, the goal of this example is to show how a domain with many known simplification rules and principles can be a source of inspiration for another domain that lacks them.

7.2.1 The Sample Problem and Issued Raised

The example presented was taken from [Petroski 1996], and it represents a two-step structural simplification for the “Old Fax” presented in Figure 1. The first step of the simplification is to remove the two pairs of rollers in the reader part. This is possible under the condition that the platen roller could take over the role of the rollers. This is obviously achievable because a similar mechanism is implemented in the printer part of the fax. The second step of the simplification is to replace the two stepping motors used in the two parts of the fax by a single stepping motor used by both parts.

We were expecting the system to produce a known structural simplification, similar to the “New Fax” presented in the same figure.

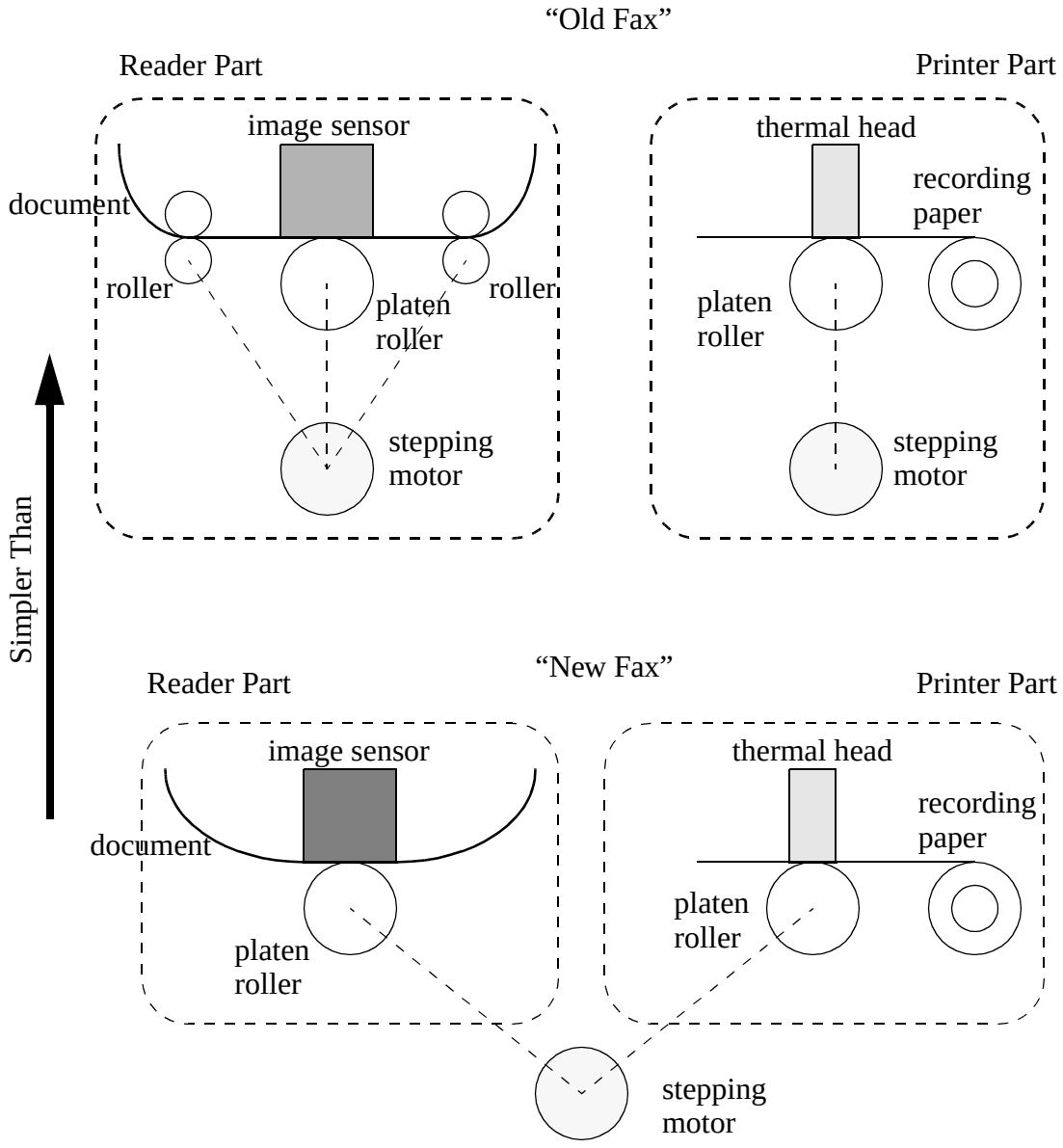


FIGURE 3. Simplification example of a Personal Fax (adapted from [Petroski 1996])

7.2.2 Operation of the System

When our system, loaded with a simplification database of arithmetic expressions, is presented with this example, it first retrieves sources containing chains of the same operation connecting objects of the same type (similar to “roller_1 moves document and platen_roller_1 moves document and roller_2 moves document”). Ideally it would retrieve a source of the form X simpler than $0 + X + 0$ (where X is an arbitrary arithmetic expression), which would obviously suggest the removing of the two rollers.

Note however that, even if such a simplification is not present in the database, a matching chain of any length will be retrieved (if one exists). The mapping built between the retrieved source and the target will suggest that two elements of the three be removed from the chain consisting of the two rollers and the platen roller. Which of the two can actually be removed has to be decided within the domain (of the fax machine). According to Petroski [1996] the only function of the two rollers was to feed the document to the image sensor. The platen roller however played a more central role in the process of reading, and could not be removed. As a consequence, the two rollers could be removed with appropriate propagation of this change (our system does not perform the propagation at this point).

The design generated by removing the two rollers is then proposed as a new simplification problem. The system will retrieve simplifications that are instances of the "factoring out" operation, that is sources of the form $X \cdot (Y + Z)$ simpler than $XY + XZ$ (where *X*, *Y* and *Z* are arbitrary arithmetic expressions). Such a source maps well onto the relation "stepping_motor_1 drives platen_roller_1 and stepping_motor_2 drives platen_roller_2". Such a mapping suggest that the structure be replaced by "stepping_motor drives platen_roller_1 and platen_roller_2".

The question that has to be addressed is whether this replacement is legal or not. In the explanation of the simplification mapped, the condition is that the two instances of *X* are identical (not the same!). The same condition can be checked for the two stepping motors, this time however in the domain of mechanical design. As this test succeeds, the transferred simplification can be applied.

CHAPTER 8

Experiments

In the previous section we described our implementation of a *design simplification system*. We built the system to demonstrate our ideas presented in the dissertation. In this chapter we describe the experiments we conducted with our design simplification system.

The general goal we pursued with the experiments described in this chapter was to demonstrate that the method we proposed in the dissertation for solving simplification problems works. For this we had to demonstrate that the system is capable of reproducing some known simplifications, both by using within-domain analogies and by using cross-domain analogies, as well as being capable of producing new simplifications. In addition, we used our experiments to demonstrate the benefits of the solutions we proposed for solving specific problems raised by the analogical reasoning approach. We can summarize the goals of our experiments as follows:

- to demonstrate that the system is effective, that is, capable of producing simplifications using known simplifications from either within the application domain, or across domains;
- to measure how using relevance influences the resources required to solve simplification problems;
- to study how using different ways of measuring complexity from the same point of view (e.g., structural complexity) effects the results produced;

To achieve each of these goals we performed several experiments. For our experiments we used two application domains: mathematical expressions and mechanical designs implemented by simple mechanisms. For each of these domains we used a separated database of simplifications in order to be able to use them both individually or in combination.

The results described in this chapter refer to a set of experiments consisting of 10 examples, of which 7 were solved using within-domain analogy and the other three using cross-domain analogy. The test examples used for within-domain analogy are listed in the first part of Appendix A.

The next sections describe the experiments we performed and discuss the results obtained. Each of the sections is organized around a specific goal.

8.1 Demonstrating that the Simplification System is Effective

To demonstrate that the system is effective we presented it with different problems, selected to test various situations.

The two major categories of experiments we performed were meant to demonstrate that:

a) the system is capable of producing simplifications by using within-domain analogies, and

b) the system is capable of producing simplifications by using cross-domain analogies.

To test the capability of the system to produce simplifications using within-domain analogies we used a simplification database of arithmetic expressions. We chose this domain for the following reasons:

- in mathematics there are many rules for equivalent transformations (some of which can actually be interpreted as simplifications), which allowed us to build a significant database of simplification examples;

- the fact that arithmetic expressions are well-formed representations (of sets of numbers) allowed us to concentrate in this phase on testing the analogical reasoning mechanism without having to be concerned with representational issues;
- since arithmetic expressions are built from relations (represented here by the operators connecting subexpressions) which are syntactically very similar to each other (in that they have the same signatures), they are good examples for testing the capability of the system to discover of similarities between *deep structures* (see note below).

  Note, that in most of the applications of model-based analogical reasoning implemented using the SME, the similarities found and exploited are between *shallow* (typically three level) *structures*. We call those structures "shallow" because the only consist of a set of "objects" connected by relations, with those relations connected by "higher level relations" (such as causality). Such systems rely more on the identity of the relation names in the structures matched, than on the structure itself. We believe that this is a significant limitation, especially if the application domain contains structured objects with possibly deep structures (such as arithmetic expressions, or designs).

To test the capability of the system to produce simplifications using cross-domain analo-

gies we used the same database of simplifications of arithmetical expressions and presented it with simplification problems drawn from the domain of simple mechanical designs. This choice is motivated by the following:

- design is our main application domain and the domain towards which we intend to further extend our research on simplification by analogical reasoning;
- most of the design domains lack good simplification rules or principles;
- mathematics, which (as stated earlier) has a significant number of simplification rules and examples can be a good source of inspiration for simplifications in other domains.

In this set of experiments we also pursued a set of specific subgoals. These subgoals were:

1. to demonstrate that the system is capable of transferring simplification knowledge whether the explanation of the source analog is given as a difference, or as a simplification process description;
2. to analyze the effect of the size of the simplification database on the effectiveness and efficiency of the system;
3. to determine what kinds of analogies the system will not be capable of building (if any).

4. to analyze how the different rules of match hypothesis generation influence the operation of the system.

In the following subsections we describe the setup of the experiments we performed to demonstrate the effectiveness of our system, and present and discuss the results obtained.

8.1.1 Setting up the Experiments

For using within-domain analogies the system was loaded with a simplification database consisting of simplifications of arithmetic expressions. The database was generated in a previous session, using the system's simplification management capability (see Chapter 7) and contained simplifications explained by difference, as well as simplifications explained by a simplification process description. We then performed the following experiments:

1. We presented the system with a set of expressions (one at a time) for which a simplification was known (but not stored in the database used) and recorded for each of those expressions whether a simplification was produced, whether it was correct from the point of view of the domain and whether it was 'the obvious simplification' an expert in mathematical expressions would perform.

2. We successively increased the size of the database by adding new simplifications and for each of the new versions we performed the same experiments as in the pre-

vious step. This time we measured and recorded the number of "specific operations" performed by the different phases of the analogical reasoning process.

The specific operations we counted for the different phases were:

- *creating of a match hypotheses* for the retrieving phase,
- *generation of gmaps* for the mapping phase,
- *generation of candidate simplifications* for the knowledge transfer phase.

The measures were then summarized and analyzed.

3. Analyzing the results of the experiments described at points 1 and 2 we hypothesized characterizations of the situations for which the system did not produce a simplification, as well as those for which it produced simplifications that were not the "obvious ones". For these hypotheses we built further examples to verify them.

For cross-domain analogies the system was loaded with the simplification databases that had been used for the within-domain analogies case. This time, however, the system was presented with examples from the domain of simple mechanical designs. To test whether the system is capable of reproducing a know simplification we used the simplification example presented in Section 7.2. The system was presented with the representation of the "Old Fax", as given in Appendix A. The experiment was repeated for the same sequence of databases as used for within-domain analogy. Again, we measured and registered the

number of specific operations performed for each of the phases of the analogical reasoning process.

Finally, we performed the complete set of experiments presented earlier turning off different combinations of match hypothesis generating rules. We performed exactly the same measurements and compared the results to analyze the influence of the different match hypothesis generation rules on the operation of our system.

8.1.2 Results and Discussion

The results of the experiments performed to demonstrate the effectiveness of the system are summarized in Table 1.

Database Size	Simplifications Produced	Correct Simplifications Produced	Match Hypotheses Generated	Gmaps Generated	Candidate Simplifications Generated
18	1	1	18	7	1
40	1	1	37	13	1
50	1	1	49	17	1
75	1	1	65	24	1
85	1	1	67	27	1

TABLE 1. Summary of Experiments Results for Within-Domain Analogies

The first column of the table characterizes the size of the database used in terms of the number of simplifications scaled with (i.e., multiplied by) the average complexity of the

"more complex" portion of those simplifications. We used this measure because the number of simplifications by itself is not informative enough: a database containing a single simplification of a very high complexity may be harder to search for matches than one containing many simplifications of very low complexity. By multiplying the number of simplifications and their average complexity together captures both of the "dimensions" of the database.

Since we expected that the system would solve all the problems we presented it with in this set of experiments, columns 2, 3 and 6 all contain 1's meaning that for each problem, the system produced exactly one candidate simplification, from which it produced the "correct" simplification (i.e., the one we expected). The other two columns suggest that both the number of match hypotheses generated and the number of global mappings (gmaps) built grows proportionally with the size of the database. Obviously this only allows the formation of a hypothesis which still need to be proven. We must note here, that a theoretical characterization of this growth is extremely difficult, if possible at all. This is true, mainly because the performance of the system will depend on both of the dimensions of the database, that is, number of simplifications and their complexity.

The conclusion we can draw from the results presented in this table is that the system we built is able to produce within-domain simplifications as expected by the user (of course,

provided it has the necessary example simplifications to rely on).

The results of experiments with different combinations of match hypothesis generating

Database Size	NAR	N	A	R	NA	NR	AR
18	18	4	7	7	11	11	14
40	37	8	14	14	23	23	28
50	49	9	19	19	30	30	38
75	65	11	26	26	39	39	52
85	67	13	30	30	43	43	60

TABLE 2.

rules are presented in Table 2. The letters in the heading row stand for the names of the match hypothesis generating rules (see Appendix A): **N**ame, common-**A**ncestor and a**R**ity, respectively. Each of those columns contain the numbers of match hypotheses generated for each of the rule combinations. Note that, for the domain of arithmetic expressions, which only use binary operations, all the match hypotheses generated by the "common ancestor" rule are also generated by the "same arity" rule, and the reverse is also true.

The results presented in the table indicate that using the different matching rules together allows for generating more match hypotheses. This information is not very useful on its own. It becomes more interesting together with the fact that even when the "same name" matching was turned off (AR), the system was able to produce the right solution. This happened due to the fact that relations with the same name were matched up as having a com-

mon ancestor or the same arity. This clearly shows that our system is able to operate correctly based only on the structure of the objects to be matched (without the need for restricting matching to objects with the same name).

A note we need to make here is that when only relying on the “common ancestor” and/or the “same arity” rules, the system’s performance, as measured by the number of match hypotheses generated, degrades compared to the case when only the “same name” rule is used (as it is done in other implementations of the structure mapping theory).

8.2 Measuring the Effect of Using Relevance

Applying the relevance of object elements (e.g., subexpressions of arithmetic expressions, or components of designs) to simplifications throughout the analogical reasoning process is one of the important contributions of this dissertation. It is the way by which the model of analogical reasoning proposed takes into account the problem solving goal (i.e., simplification), producing goal-directed analogical reasoning.

To demonstrate that using relevance to guide the analogical reasoning process improves the performance of our system, we had to measure its effect. For this we needed:

1. to hypothesize which phases of the analogical reasoning process could be affected by the use of relevance,
2. to define a measure of performance for each of those phases,
3. to design and perform a set of experiments to collect statistics about the performance measures of the different phases.

As we pointed out in Chapter 2, choosing physical time as measure of the time performance (complexity) of a process is not adequate for several reason. As it is done in the literature on algorithm complexity (see [Brassard & Bratley, 1996]), we have proposed there that the time performance of a process should be measured by the number of *specific operations* performed. In our experiments, performed to measure the effect of using relevance on the performance of our system, we adopted the same approach. Unfortunately there is no unique specific operation that can be counted to measure the system's overall performance. For this reason we chose to identify specific operations for each of the phases of the analogical reasoning process that are affected by the use of relevance.

In addition to simply measuring the effects of using relevance on the performance of the system, in this set of experiments we also pursued the following subgoals:

1. to study the effect of different kinds of relevance propagation methods (e.g., downward propagation, limited propagation, no propagation) on the operation of the system (Note here, that in the current implementation of our system we do not support upwards propagation, which is the reason why this propagation method was not tested);

2. to study whether relevance propagation has the same effect or not for the two different ways simplification explanations are specified (i.e., by difference, or by simplification process description.

The following subsections describe the setup for the experiments performed to measure the effect of using relevance to guide the analogical reasoning process, and present and discuss the results obtained.

8.2.1 Setting up the Experiments

To measure the effect of using relevance on the operation of the system, we first hypothesized that the phases affected will be: a) the retrieval of source analogs, and b) the mapping of the source analog retrieved onto the target. For both of these phases we defined a measure of performance in terms of a specific operation. The measures used were:

- the *number of a match hypotheses created* for the retrieving phase,
- the *number of gmaps generated* for the mapping phase.

We used the same simplification databases and examples as described in the previous subsection. We performed the following experiments:

1. We turned off the relevance checking in the system. Then, for each of the versions of the simplification database used in demonstrating the effectiveness of our system, we presented the system with all those examples used in the first set of experiments which produced correct simplifications. We repeated the same set of experiments with the relevance checking turned on. For both of the cases we collected the measures defined above and compared the results.
2. We also wanted to study whether considering relevant elements to be only the ones explicitly referred to in the explanation of simplifications would make a difference in the performance of the system. For this purpose we regenerated the databases used for our experiments such that the "relevance propagation" is limited to only one level (i.e., only to the elements explicitly referred to). We reran our experiments described in point 1. above with the newly generated simplification databases and performed the same measurements and comparisons.

3. We partitioned the set of examples for which we performed the experiments into two sets. One set contained examples that had been solved using analogy with simplifications explained by differences, while the other one contained examples that had been solved using analogy with simplifications explained by specifications of simplification processes. With these two sets of examples we performed the same experiments as the ones described in the first point, above. We used the results of our measurements to analyze whether the use of relevance has different effects for the two kinds of simplifications.

8.2.2 Results and Discussion

The results of the experiments for measuring the effect of using relevance are given in

	Relevance OFF		Relevance ON		One-Level Relevance	
Database Size	**Match Hypotheses Generated**	**Gmaps Generated**	**Match Hypotheses Generated**	**Gmaps Generated**	**Match Hypotheses Generated**	**Gmaps Generated**
18	18	7	11	4	14	6
40	37	13	27	9	32	11
50	49	17	36	12	41	17
75	65	24	41	15	52	21
85	67	27	56	21	61	33

TABLE 3. Summary of Experimental Results for Measuring the Effect of Using Relevance

Table 3. The three main columns in the table correspond to performing the experiments

without taking into account the relevance, with the relevance fully propagated down (see Chapter 4), and with the relevance only propagated down one level in the structure of the design matched, respectively. In all the cases the system produced the expected simplification.

The results show that when taking relevance into account the system generates much fewer match hypotheses and, as a consequence of this, much fewer global mappings. This leads to an improvement of the performance of the system. Just by looking at the results in Table 3, we could claim that applying (fully propagated) relevance produced an improvement of about 50, in terms of both the number of match hypotheses generated and number of global mappings generated. This is however a result that cannot be claimed to hold in every situation because the actual results depend on the size of the database, the complexity of designs involved in the simplifications and the number of elements in each simplification that are relevant to it (e.g., for a given simplification it is possible that only one element is relevant, but it is also possible that all the elements of the "simpler" design involved are relevant).

Our experiments with the one-level propagation of relevance show that, in this case, the system generates slightly more match hypotheses and gmaps than when using full propagation, but less than when not using propagation at all. The reason for this is that one-level

propagation restricts the elements that can be matched, but not to the extent by which full propagation does. It appears that using full propagation would always be the best choice. However this may not always be the case. For simplifications for which the explanation is given by difference a full propagation may be needed because there is no way of knowing under what conditions the difference is applicable. On the other hand, for simplifications for which the explanation is given by a simplification process description, no propagation should needed, because, ideally, all the relevant elements should be referenced in the process description (in a condition, a transformation, or a state description). As a consequence, the issue of propagation needs further studying.

The conclusion we draw from this part of our experiments is that using relevance to guide the analogical reasoning process can significantly improve the performance of the system.

8.3 Measuring the Effect of Using Different Complexity Measures

There are two phases of the analogical reasoning process where structural complexity measures are used: a) *during retrieving*, when searching for the best candidate source analog, and b) *during mapping*, when searching for the best global mapping (gmap). In both of these cases the system computes the complexity of structured objects. Thus, we can use

either of the three ways of measuring structural complexity presented in Chapter 2 (i.e., element count, combined element and relation count and external path length). For the two phases mentioned above we conducted two different sets of experiments.

For the effect of the complexity measure used on the retrieving phase of the analogical reasoning process our goal was to measure how well each of the three measures characterized the similarity between two structures.

To determine the effect of the complexity measure used on the mapping phase our goal was to measure how well each of those measures characterized the quality of a mapping. In both of the cases we presented the system with specially built examples which were designed to reveal the advantages and disadvantages of the three measures.

The following subsections describe the way we have set up the experiments for measuring the effect of using different complexity measures, and present and discuss the results obtained.

8.3.1 Setting up the Experiments

To measure the effect of different complexity measures on the operation of the system we preformed a set of experiments that concentrated on the two phases of interest as

described above. This time we used the last (largest) version of our simplification database of arithmetic expressions, and varied the examples presented to the system. The examples we used in these experiences were specially built such that they represent different combinations of structural characteristics (e.g., shallow but broad structures versus deep structures) and of match types (e.g., exact name matches of elements versus common ancestor matches, versus arity matches). We used three groups of examples corresponding to these three types of match.

For each of these groups we performed experiments with all the three structural complexity measures mentioned in subsection 8.1.3. During the experiments we collected data about the number of match hypotheses that were generated during the retrieval phase and how the known match was ranked according to the corresponding measure, as well as the number of gmaps generated during the mapping phase and whether the known gmap (i.e., the global mapping that we would use to build the analogy) was generated.

8.3.2 Results and Discussion

The first result of our measurements in this set of experiments was that the number of match hypotheses generated was actually independent of the complexity measure used. This result was expected because the creation of match hypotheses only relies on the types

of objects.

The experiments showed that using different complexity measures for evaluating matches and global mappings influence *the way matches are ranked* according to their scores, following the propagation of matches, the *number of global mappings* generated and whether a global mapping leading to a *correct simplification* was generated or not. Characterizing the exact dependency between the complexity measure used and their effect is hard because those effects depend both on the simplification problem being solved and the simplification database used for solving the problem.

Based on our observations, the following characteristics of the problem and of the database of simplification may influence the three aspects of the matching and mapping processes mentioned above (i.e., ranking of the matches, number of global mappings generated and producing of a correct result):

- the number of components, relations and attributes of the designs involved in the processes (i.e., the target design and the source designs in the simplification database)
- the depth and breadth of the structure of the designs involved in the processes, and
- the size of the simplification database used.

The most ‘elusive’ of these characteristics seems to be the depth and breadth of the structure of the designs involved in the processes. Even minor differences in either of these characteristics may result in a very different result of the matching and, consequently of the mapping. Characterizing this kind of effect would probably require the use of better complexity measures. This, however, would only be possible at an increased computational cost, which may not be acceptable because evaluation of matches and mappings is performed a great number of times (depending on the contents and size of the database used).

8.4 Conclusions

As a general conclusion to our experiments we can say that the system presented in Chapter 6 has proven to be effective in solving simplification problems. We demonstrated that using relevance to a simplification to guide the analogical reasoning process improves the performance of the system. Finally, while it is clear that using different complexity measures for evaluating matches and global mappings influences both the effectiveness and the efficiency of the system, the exact dependency is hard to analyze both theoretically and experimentally.

CHAPTER 9

Conclusion

In this dissertation we described our research concerning *simplification*. In it we defined the simplification problem, proposed a way to solve simplification problems and analyzed the results of a series of experiments we had conducted to demonstrate our ideas. In this last chapter we will draw some final conclusions about our research as well as present our plans for continuing and extending this research.

9.1 Contributions

The research presented in this dissertation defined had two major objectives: a) to define the simplification problem, and b) to propose a way to solve simplification problems. In pursuing these objectives this research has produced the following contributions:

1. It gave an *operational definition of simplification and of a simplification problem*.

The study of simplification and of solving simplification problems is, to our knowledge, a new area of research. Simplification was defined as reduction of complexity. This required a definition of the complexity of objects. The definition of complexity with respect to a point of view (i.e., combination of context, aspect and measure) gives a fresh view on complexity.

2. It proposed a model for *solving simplification problems by using analogical reasoning*.

   As simplification is a novel area of research, solving simplification problems by analogical reasoning is of course new as well. More importantly, however, it constitutes a new application of analogical reasoning in general and of model-based analogical reasoning in particular.

3. It proposed an *improvement to the general mode-based analogical reasoning model as applied to simplification problems*, by using the problem solving goal to guide the process.

   Using the problem solving goal to guide the analogical reasoning process has been suggested by several authors [Gentner 1993] [Holyoak & Thagard 1989] [Forbus & Oblinger 1990], but, to our knowledge, our work is unique in the way it uses it in all of the phases. We proposed that the knowledge that is to be used to guide the analogical reasoning process should be the relevance to simplification of different elements of the

objects processed. This way, the processing required by the analogical reasoning can be significantly reduced.

4. It *defined the notion of simplification for different aspects of designs*.

   Simplification of designs is the most important application area projected for our research. To address this problem, our research proposed definitions for the complexity of the three most frequently considered aspects of designs: structure, behavior and function. We also pointed out that since structure, behavior and function are interdependent, this will be reflected in the dependency between their complexities. As a consequence we formulated another very important research problem, namely the study of the propagation of simplifications.

5. It proposed a *model-based analogical reasoning approach to design simplification* problems.

   This represents a new application of model based analogical reasoning. To build the model we defined the representation of objects and of simplifications, and the organization of the database of known simplifications. These were designed such that they support the processing required by our model of goal-directed model-based analogical reasoning.

6. It *produced a working system* that implements the model proposed for solving simplification problems.

The working simplification system was implemented using the CLIPS language. The implementation used all the elements proposed by our research and was extensively tested on examples.

Here we must note again the system was demonstrated for structural simplification problems only. However, as described in Chapter 1, the approach is applicable to behavioral and functional simplification problems as well. In order for the system to be able to perform behavioral and functional simplification the following two changes are needed:

a) The representation of processes (e.g., behaviors, interactions) must be changed to a decomposition-relation form. This means that a process will be represented as a set of steps, connected by a sequencing (followed-by) relation. The current implementation of the system contains all the class definitions needed to make this modification.

b) New matching rules for generating match hypotheses for objects specific to representing behaviors and functions (e.g., state descriptions, inputs and outputs) need to be added to the set of Retrieving Rules (see Chapter 6). This can be done easily by taking the existing match hypothesis generating rules as models.

7. Through a series of experiments, this research *demonstrated that the system is operational.*

   The experiments demonstrated that the system is capable of producing simplifications both using known simplifications from the same domain as the problem, and from a different domain.

8. Using the same experiments the research *analyzed effects of the improvements proposed* on the performance of the system.

   Measurements performed in those experiments also revealed that using the simplification goal (in our case relevance to some simplification) can significantly improve the performance of the system.

In addition to these general contributions, the research presented in this dissertation also produced the following technical results:

1. The use of relevance in generating match hypotheses, retrieving source analogs and building mappings improves the performance of the analogical reasoning mechanism.
2. Explicitly representing the explanation of known simplifications allows a) the automatic calculation of the relevance for a given simplification, and b) guidance of the transfer of simplification knowledge, either by transformation[1] (in the case of explanations speci-

fied by differences), or by derivation (in the case of explanations specified by simplification process).

3. The use of external path length as measure for the complexity of tree structures improved the effectiveness of retrieving good source analogs for deep structures.

As a general conclusion the research presented in this dissertation proposed new research directions, presented original definitions, proposed new applications and approaches to existing problems, and implemented and experimentally studied a new system.

9.2 Future Work

In addition to the results proposed, the research presented in this dissertation opened up new research directions and raised a series of theoretical and practical questions that need to be studied. This gives us several opportunities to extend our research on simplification, in general, and on design simplification, in particular. In the following we briefly describe our plans for future work, starting with the closest goals and ending with farther ones.

1. Here we use the terms ‘transformation’ and ‘derivation’ corresponding to the knowledge transfer phases in transformational and derivational analogical reasoning, respectively.

9.2.1 Performing Further Experiments with the System

We will perform further experiments with the system. In doing so we will pursue goals such as:

- testing the system for various examples, especially with real life problems, drawn from the area of design,
- testing the system with several databases, corresponding to different application domains, loaded simultaneously, in order to experiment with the capability of the system of finding the best analog from several domains,
- testing the scalability of the system.

These experiments can be done with the current implementation of the system, but they ned to be performed first to lay a firm foundation for our further experimental research.

9.2.2 Improve the Usability of the System by Building a GUI

To make the use of the system easier as well as to make it available to other users (researchers or designers) we will develop a graphical user interface (GUI) for the system. This can be achieved by using existing GUI development tools, such as wxCLIPS.

9.2.3 Extending the System to other Types of Simplification

Currently the system has been tested for structural simplifications, but in our future research we will perform experiments with both behavioral and functional simplification. This will allow us to fully demonstrate our ideas about design simplification.

9.2.4 Adding New Application Domains

Adding new application domains to the ones currently accepted by the system will extend the area of possible applications as well as increase the capability of the system to use cross-domain similarities to produce interesting, and hopefully novel simplifications.

9.2.5 Studying the Simplification Propagation Problem

The simplification propagation problem which we formulated in Chapter 2 is by itself a very interesting and rich area of research. It raises problems such as: What representation can adequately support the propagation of simplifications? How can the propagation of simplification be performed? What are the possible consequences of simplification propagation and how can those consequences be evaluated and anticipated?

In our future research we plan to address the problem of simplification propagation. Our ultimate goal in this direction is to incorporate a simplification propagation mechanism

into our system.

9.2.6 Studying the Possibility of Generating Creative Simplifications

Finally, as stated earlier we expect that due to the use of analogical reasoning, our approach to solving simplification problems may come up with creative simplifications. For example, “importing” a simplification idea from a different domain may suggest a completely novel way of simplifying.

We are interested in studying under what conditions our goal-directed analogical reasoning simplification process will be able to produce creative results.

As an overall conclusion we can say that in this dissertation we proposed a *new direction of research*, that of design simplification, defined the *design simplification problem* which is the general problem of this research direction, and proposed an approach to solving problems of this type, called *design simplification by analogical reasoning*. We also described a *computer system* that implements the approach proposed, and presented and discussed a set of *experiments* we had performed to demonstrate our system.

Based on these, we believe that the results presented in this dissertation has all the ingredients of a complete research.

Bibliography

M.E. Balazs & D.C. Brown, "The Use of Function, Structure and Behavior in Design", Preprints of *Workshop on Representing Function in Design*, AID-94, AI in Design Conference, 1994.

M.E. Balazs, D.C. Brown, P. Bastien & C. Wills, "How to Present Designs", in *Knowledge Intensive CAD*, vol. 2, (Eds.) Mantyla, Finger and Tomiyama, Chapman & Hall, 1997.

M.E. Balazs & D.C. Brown, "Function in Design Presentations", Preprints of *Workshop on Functional Reasoning*, AID'96, Stanford, CA, 1996.

M.E. Balazs & D.C. Brown, "A Preliminary Investigation of Design Simplification by Analogy, in Proceedings of *Artificial Intelligence in Design '98*, (Eds.)Gero & Sudweeks, Kluwer, 1998.

M.E. Balazs & D.C. Brown, "Structural, Behavioral and Functional Simplification of Designs", Proceedings of the *Functional Modeling and Teleological Reasoning Workshop*, AAAI-98, Madison, Wisconsin, USA, 1998.

G. Brassard & P. Bratley, *Fundamentals of Algorithmics*, Prentice-Hall, 1996.

F. Bacchus & Q. Yang, "Downward refinement and the efficiency of hierarchical problem solving", *Artificial Intelligence*, 71, 1994, pp. 43-100.

H. A. Bashir & V. Thomson, "Models for Estimating Design Effort", *Design Studies* (submitted), 1999.

H. A. Bashir & V. Thomson, "A Quantitative Estimation Methodology for Design Projects", *Journal of Engineering Design* (submitted), 1999.

S.R. Bhatta, A.K. Goel & S. Prabhakar, "Innovation in analogical design: A model-based approach" In: *Artificial Intelligence in Design'94*, (Eds.) Gero & Sudweeks, Kluwer Academic Publishers, 1994, pp. 57-74.

S.R. Bhatta, A.K. Goel, "Discovery of Physical Principles from design Experience", *AI EDAM*, 2, May, 1994.

M. Boden, "What is Creativity", In: *Dimensions of Creativity*, (Ed.) Boden, MIT Press, Cambridge, MA, London, England, 1994, pp. 75-118.

G. Boothroyd & P. Dewhurst, "Product Design for Manufacture and Assembly", In: *Design For Manufacture*, (Eds.) Corbett, Dooner, Meleka & Pym, Addison-Wesley, 1991, pp. 165-173.

J.G. Carbonell, “Derivational analogy: A theory of reconstructive problems”, In: *Readings in Knowledge Acquisition and Learning*, (Eds.) Buchanan & Wilkins, Morgan Kaufmann Publishers, San Mateo, CA, 1993, pp. 727-738.

A. Chakrabarti & M.X. Tang, “Generating Conceptual Solutions on FUNCSION: Evolution of a Functional Synthesizer“, In: *Artificial Intelligence in Design ‘96*, (Eds.) Gero & Sudweeks, Kluwer Academic Publishers, 1996, pp. 603-622.

B. Chandrasekaran, “Functional Representation: A Brief Historical Perspective”, *Applied Artificial Intelligence*, Special Issue on Functional Reasoning, 1994.

B. Chandrasekaran & J. R. Josephson, “Representing Function as Effect: Assigning Functions to Objects in Context and Out”, *Working Notes of the AAAI-96 Workshop on Modeling and Reasoning with Function*, 1996.

L. Chittaro, “Functional Diagnosis and Prescription of Measurements using Effort and Flow Variables”, IEEE Control Theory and Applications, Vol. 142, No. 5, 1995.

L. Chittaro, G. Guida, C. Tasso, & E. Toppano, “Functional and Teleological Knowledge in Multimodelling Approach for Reasoning about Physical Systems: A case Study in Diagnosis”, *IEEE Transactions on Systems, Man and Cybernetics*, Vol, 23., No. 6, 1993.

CLIPS, *C Language Integrated Production System*, Version 6.0, Lyndon B. Johnson Space Center, Software Technology Branch, 1993.

S. Dasgupta *Creativity in Invention and Design*, Cambridge University Press, 1994.

S. Dasgupta, *Technology and Creativity*, Oxford University Press, 1996.

B. Falkenheimer, K. Forbus & D. Gentner, "The structure-mapping engine: Algorithm and examples", In: *Readings in Knowledge Acquisition and Learning*, (Eds.) Buchanan & Wilkins, Morgan Kaufmann Publishers, San Mateo, CA, 1993, pp. 695-726.

R. Finke, T.B. Ward & S.M. Smith, *Creative Cognition*, MIT Press, Cambridge, MA, London, England, 1992.

K. D. Forbus, D. Gentner, A. Markman & R. W. Ferguson, "Analogy just looks like high level perception: Why a domain-general approach to analogical mapping is right", *Journal of Experimental and Theoretical Artificial Intelligence*, 1997

K. Forbus, K. & D. Gentner, "Structural evaluation of analogies: what counts?", *Proceedings of the Cognitive Science Society.* 1989.

K. Forbus & D. Oblinger, Making SME greedy and pragmatic. *Proceedings of the Cognitive Science Society*, 1990.

A. Gelsey, "Automated physical modeling", In: *Proc. 11th Int. Jnt. Conf. on AI*, Detroit, MI, August, 1989, pp. 1225-1230.

D. Gentner, "Structure-Mapping: A Theoretical Framework for Analogy", *Cognitive Science*, 7, 1983, pp. 155-170.

D. Gentner, “Analogical Inference and Access”, In: *Analogica*, (Ed.) Prieditis, Lecture Notes in Artificial Intelligence, Morgan Kaufmann Publishers, Los Altos, CA, 1988, pp. 63-88.

D. Gentner and K. Forbus, “MAC/FAC: A model of similarity-based retrieval”, *Proceedings of the Cognitive Science Society*, 1991.

D. Gentner, “The mechanism of analogical learning”, In: *Readings in Knowledge Acquisition and Learning*, (Eds.) Buchanan & Wilkins, Morgan Kaufmann Publishers, San Mateo, CA, 1993, pp. 673-694.

F. Giunchiglia & T. Walsh, “A theory of abstraction“, *Artificial Intelligence*, 57, 1992, pp. 323-389.

A. Goel, “Representation of Design Functions in Experience-Based Design”, In: *Intelligent Computer Aided Design*, (Eds.) Brown, Waldron & Yoshikawa, North-Holland, Amsterdam, Netherlands, 1992, pp.283-308

A. Goel, “Design, Analogy and Creativity”, *IEEE Expert*, vol. 12, no. 3, May/June, 1997, pp.62-70.

R. P. Hall, “Computational Approaches to Analogical Reasoning: A Comparative Analysis”, *Artificial Intelligence*, 39, 1989, pp. 39-120.

D.H. Hofstadter, *Fluid concepts and creative analogies*, Basic Books, New York, 1995.

K.J. Holyoak & P. Thagard, “Analogical mapping by constraint satisfaction”, *Cognitive Science*, 7(2), 1989.

K.J. Holyoak & P. Thagard, *Mental Leaps: Analogy in Creative Thought*, MIT Press, Cambridge, MA, London, England, 1995.

S. Kedar-Cabelli, “Toward a Computational Model of Purpose-Directed Analogy”, In: *Analogica*, Research Notes in Artificial Intelligence, (Ed.) Prieditis, 1988, pp. 89-108.

A. Keuneke, “Device Representation: The Significance of Functional Knowledge”, *IEEE Expert*, vol. 6, no. 2, April, 1991, pp. 22-25.

C. A. Knoblock, “Automatically generating abstractions for planning”, *Artificial Intelligence*, 68,1994, pp. 243-302

A. Y. Levy, “Creating Abstractions Using Relevance Reasoning”, In: *Proceedings of the 12th National Conference on Artificial Intelligence, AAAI’94*, Vol. 1 1994: 588-594

D. Manfaat, *The SPIDA system*, personal communication, 1997.

D. Manfaat, A.H.B. Duffy and B.S. Lee, “Generalization of Spatial Layouts”, Workshop on Machine Learning in Design, 23-24 June 1996, *Fourth International Conference on Artificial Intelligence in Design’96*, Stanford University, USA, 1996.

J. Mostow, “Design by Derivational Analogy: Issues in the Automated Replay of Design Plans, *Artificial Intelligence*, 40, 1989, pp. 119-184.

P. P. Nayak & A. Y. Levy, “A Semantic Theory of Abstractions”, In: *Proceedings of the International Conference on Artificial Intelligence, IJCAI’95*, 1995, pp. 196-203.

D.N. Perkins, “Creativitie’s Camel: The Role of Analogy in Invention”, In: *Creative Thought - An Investigation of Conceptual Structures and Processes* (Eds.) T. B. Ward, S.M. Smith & J. Vaid, American Psychological Association, 1997

H. Petroski, *Invention by Design*, Harvard University Press, 1996.

S. Prabhakar & A.K. Goel, “Functional modeling for enabling adaptive design of devices for new environments”, *Artificial intelligence in Engineering*, 12, 1998, pp. 417-444.

L. Qian & J.S. Gero, “A design support systems using analogy”, In: *Artificial Intelligence in Design'92*, (Ed.) J.S. Gero, Kluwer Academic Publishers, 1992, pp.795-813.

V. Sembugamoorthy & B. Chandrasekaran, “Functional Representation of Devices and Compilation of Diagnostic Problem-Solving Knowledge”, In: *Experience, Memory and Learning*, (Eds.) Kolodner & Reisbeck, Lawrence Erlbaum Associates, 1986.

J. Sticklen, A. Goel, B. Chandrasekaran & W. E. Bond, “Functional Reasoning for Design and Diagnosis”, in: Proceedings of the Model-Based Diagnosis International Workshop@, 1989.

H. W. Stoll, “Design for manufacture: an overview”, In: *Design for Manufacture, Strategies, Principles and Techniques*, (Eds.) Corbett, Dooner, Meleka & Pym, Addison-Wesley Publishing Company, 1991.

E. Stroulia., A.K. Goel, “Representation of Design Structure, Behavior and Function for Blame Assignment”, *AI EDAM,* special issue on Functional Reasoning, 1995.

N.P. Suh, *The Principles of Design,* Oxford University Press, 1990.

N.P. Suh, “A Theory of Complexity, Periodicity and the Design Axioms”, *Research in Engineering Design,* Vol. 11, No.2, 1999, pp. 116-131.

S. Thadami & B. Chandrasekaran, “Structure to function reasoning”, *AAAI’93, Workshop on Representing and Reasoning With Device Function,* 1993.

Y. Umeda & T. Tomiyama, “Experimental Use and Extension of the FBS Modeler”, *AAAI’94 Workshop on Representing and Reasoning with device Function,* 1994.

M.M. Veloso, *Planning and Learning by Analogical Reasoning,* Lecture Notes in Artificial Intelligence (886), Springer Verlag, 1994.

M. Wolverton, *Retrieving Semantically Distant Analogies,* Ph. D. Dissertation, Stanford University, 1994.

Appendix A

This appendix illustrates the external representation of designs by presenting the representation of the “Old Fax” described in Chapter 8.

```
;=========================================================
;  PERSONAL FAX MACHINE
;=========================================================

(RootDesign Personal_Fax_Current
  (Attributes)
  (Components
  ;=======================================================
  ;  PRINTING PART
  ;=======================================================
  (bind ?printPart
     (Design Printing_Part
        (Attributes)
        (Components ; // of Printing Part
        ;=================================================
        ; STEPPING MOTOR (printer)
        ;=================================================
        (bind ?stepping-motor1
           (Design Stepping_Motor_1
              (Attributes
                 (bind ?attrib-dyn-sm1(Attribute dynamics))
                 (bind ?attrib-str-sm1(Attribute started))
              )
              FALSE ;// no components
              FALSE ;// no local relations
```

```
            (Behaviors
               (bind ?proc-strt-sm1
               (Process Start-Stepping Motor
                  (ProcStep
                     (bind ?stat-stop-sm1
                     (ObjectState ?attrib-str-sm1 FALSE
                                  ?attrib-dyn-sm1 static))
                     (Condition Accept Input ?attrib-str-sm1 TRUE))
                     (Apply and
                        (Apply send
                                ?attrib-str-sm1 put-Value TRUE
                        (Apply send
                                ?attrib-dyn-sm1 put-Value rotates))
                     (bind ?stat-rota-sm1
                     (ObjectState ?attrib-str-sm1 TRUE
                                  ?attrib-dyn-sm1 rotates))
                  )
               ))
               (bind ?proc-stop-sm1
               (Process Stop-Stepping-Motor
                  (ProcStep
                     ?stat-rota-sm1
                     (Condition Accept (Input ?attrib-str-sm1 FALSE))
                     (Apply and
                        (Apply send ?attrib-str-sm1 put-Value FALSE)
                        (Apply send ?attrib-dyn-sm1 put-Value static))
                     ?stat-stop-sm1
                  )
               ))
            )
         ))
      ;=================================================
      ; PLATTEN ROLLER (printer)
      ;=================================================
      (bind ?platen-roller1
         (Design Platen_Roller_1
            (Attributes
               (bind ?attrib-dyn-pr1(Attribute dynamics))
            )
            FALSE ;// no components
            FALSE ;// no local relations
            (Behaviors
               (bind ?proc-strt-pr1
               (Process Start-Platen-Roller
                  (ProcStep
                     (bind ?stat-stop-pr1
                     (ObjectState ?attrib-dyn-pr1 static))
                     (Condition TRUE)
                     (Apply send ?attrib-dyn-pr1 put-Value rotates)
                     (bind ?stat-rota-pr1
```

```
                  (ObjectState ?attrib-dyn-pr1 rotates))
               )
            ))
            (bind ?proc-stop-pr1
            (Process Stop-Platen-Roller
               (ProcStep
                  ?stat-rota-pr1
                  (Condition TRUE)
                  (Apply send ?attrib-dyn-pr1 put-Value static)
                  ?stat-stop-pr1
               )
            ))
         )
      ))

   ;=====================================================
   ; THERMAL HEAD
   ;=====================================================
   (bind ?thermal-head1
      (Design Thermal_Head_1
         (Attributes
            (bind ?attrib-str-th1 (Attribute started))
         )
         FALSE ;// no components
         FALSE ;// no local relations
         (Behaviors
            (bind ?proc-strt-th1
            (Process Start-Thermal-Head
               (ProcStep
                  (bind ?stat-nopr-th1
                  ObjectState ?attrib-str-th1 FALSE))
                  (Condition TRUE)
                  (Apply send ?attrib-str-th1 put-Value TRUE)
                  (bind ?stat-prnt-th1
                  (ObjectState ?attrib-str-th1 TRUE))
               )
            ))
            (bind ?proc-stop-th1
            (Process Stop-Thermal-Head
               (ProcStep
                  ?stat-prnt-th1
                  (Condition TRUE)
                  (Apply send ?attrib-str-th1 put-Value FALSE)
                  ?stat-nopr-th1
               )
            ))
         )
      ))
```

```
   ;=================================================
   ; RECORDING PAPER
   ;=================================================
   (bind ?recording-paper1
      (Design Recording_Paper_1
         (Attributes
            (bind ?attrib-dyn-rp1(Attribute dynamics))
         )
         FALSE ;// no components
         FALSE ;// no local relations
         (Behaviors
            (bind ?proc-strt-rp1
            (Process Start-Recording-Paper
               (ProcStep
                  (bind ?stat-stop-rp1
                  (ObjectState ?attrib-dyn-rp1 static))
                  (Condition TRUE)
                  (Apply send ?attrib-dyn-rp1 put-Value translates)
                  (bind ?stat-trns-rp1
                  (ObjectState ?attrib-dyn-rp1 translates))
               )
            ))
            (bind ?proc-stop-rp1
            (Process Stop-Recording-Paper
               (ProcStep
                  ?stat-trns-rp1
                  (Condition TRUE)
                  (Apply send ?attrib-dyn-rp1 put-Value static)
                  ?stat-stop-rp1
               )
            ))
         )
      ))
   ) ;// end Components

;=RELATIONS=============================================

(bind ?rels-print
(Relations ;// of Printing Part
   (bind ?touch-print
   (Relation Touching
            ?stepping-motor1 ?platen-roller1 ?recording-paper1))
   ;// this is a preprocessed form of
   ;// (and
   ;//    (Relation Touching ?stepping-motor1 ?platen-roller1)
   ;//    (Relation Touching ?platen-roller1 ?recording-paper1))

))
```

```
    ;=BEHAVIORS================================================

    (Behaviors ;// of Printing Part
       (bind ?start-print
       (Process Start-Printing
          (ProcStep
             (bind ?stat-stop-print
             (ObjectState ?stat-stop-sm1 ; initial state
                          ?stat-stop-pr1
                          ?stat-nopr-th1
                          ?stat-stop-rp1))
             ?touch-print                ; condition
             (Process Start-Components   ; printing process
                ?proc-strt-sm1
                ?proc-strt-pr1
                ?proc-strt-th1
                ?proc-strt-rp1
             )
             (bind ?stat-strt-print
             (ObjectState ?stat-rota-sm1 ; final state
                          ?stat-rota-pr1
                          ?stat-prnt-th1
                          ?stat-trns-rp1))
          )
       ))
       (bind ?stop-print
       (Process Stop-Printing
          (ProcStep
             ?stat-strt-print            ; initial state
             ?rels-print                 ; final state
             (Process Stop-Components    ; stopping process
                ?proc-stop-sm1
                ?proc-stop-pr1
                ?proc-stop-th1
                ?proc-stop-rp1
             )
             ?stat-stop-print            ; final state
          )
       ))
    )

    ;=FUNCTIONS================================================
    (Uses
       (bind ?startPrint (Process Start-Printing ))
       (bind ?stopPrint  (Process Stop-Printing ))
    )
 ) ;// end Printing Part
)
```

```
;=========================================================
;  IMAGE SENSING PART
;=========================================================
(bind ?scanPart
   (Design ImageSensingPart
      (Attributes)
      (Components ; // of Image Sensing Part
      ;=================================================
      ; STEPPING MOTOR (reader)
      ;=================================================
      (bind ?stepping-motor2
         (Design Stepping_Motor_2
            (Attributes
               (bind ?attrib-dyn-sm2(Attribute dynamics))
               (bind ?attrib-str-sm2(Attribute started))
            )
            FALSE ;// no components
            FALSE ;// no local relations
            (Behaviors
               (bind ?proc-strt-sm2
               (Process Start-Stepping-Motor
                  (ProcStep
                     (bind ?stat-stop-sm2
                     (ObjectState ?attrib-str-sm2 FALSE
                                  ?attrib-dyn-sm2 static))
                     (Condition Accept (Input ?attrib-str-sm2 TRUE))
                     (Apply and
                        (Apply send ?attrib-str-sm2 put-Value TRUE)
                        (Apply send
                                ?attrib-dyn-sm2 put-Value rotates))
                     (bind ?stat-rota-sm2
                     (ObjectState ?attrib-str-sm2 TRUE
                                  ?attrib-dyn-sm2 rotates))
                  )
               ))
               (bind ?proc-stop-sm2
               (Process Stop-Stepping-Motor
                  (ProcStep
                     ?stat-rota-sm2
                     (Condition Accept (Input ?attrib-str-sm2 FALSE))
                     (Apply and
                        (Apply send ?attrib-str-sm2 put-Value FALSE)
                        (Apply send ?attrib-dyn-sm2 put-Value static))
                     ?stat-stop-sm2
                  )
               ))
            )
   ))
```

```
;=================================================
; PLATTEN ROLLER (reader)
;=================================================
(bind ?platen-roller2
   (Design Platen_Roller_2
      (Attributes
         (bind ?attrib-dyn-pr2(Attribute dynamics))
      )
      FALSE ;// no components
      FALSE ;// no local relations
      (Behaviors
         (bind ?proc-strt-pr2
         (Process Start-Platen-Roller
            (ProcStep
               (bind ?stat-stop-pr2
               (ObjectState ?attrib-dyn-pr2 static))
               (Condition TRUE)
               (Apply send ?attrib-dyn-pr2 put-Value rotates)
               (bind ?stat-rota-pr2
               (ObjectState ?attrib-dyn-pr2 rotates))
            )
         ))
         (bind ?proc-stop-pr2
         (Process Stop-Platen-Roller
            (ProcStep
               ?stat-rota-pr2
               (Condition TRUE)
               (Apply send ?attrib-dyn-pr2 put-Value static)
               ?stat-stop-pr2
            )
         ))
      )
   ))

;=================================================
; CONTACT IMAGE SENSOR (reader)
;=================================================
(bind ?image-sensor2
   (Design Contact_Image_Sensor_2
      (Attributes
         (bind ?attrib-str-is2(Attribute started))
      )
   FALSE ;// no components
   FALSE ;// no local relations
   (Behaviors
      (bind ?proc-strt-is2
      (Process Start-Image-Sensor
         (ProcStep
            (bind ?stat-nord-is2
```

```
               (ObjectState ?attrib-str-is2 FALSE))
               (Condition TRUE)
               (Apply send ?attrib-str-is2 put-Value TRUE)
               (bind ?stat-read-is2
               (ObjectState ?attrib-str-is2 TRUE))
            )
         ))
         (bind ?proc-stop-is2
         (Process Stop-Image-Sensor
            (ProcStep
               ?stat-read-is2
               (Condition TRUE)
               (Apply send ?attrib-str-is2 put-Value FALSE)
               ?stat-nord-is2
            )
         ))
      )
   ))

   ;=================================================
   ; A ROLLER (reader)
   ;=================================================
   (bind ?roller1
      (Design Roller_1
         (Attributes
            (bind ?attrib-dyn-rl21
               (Attribute dynamics))
         )
         FALSE ;// no components
         FALSE ;// no local relations
         (Behaviors
            (bind ?proc-strt-rl21
            (Process Start-Roller
               (ProcStep
                  (bind ?stat-stop-rl21
                  (ObjectState ?attrib-dyn-rl21 static))
                  (Condition TRUE)
                  (Apply send ?attrib-dyn-rl21 put-Value rotates)
                  (bind ?stat-rota-rl21
                  (ObjectState ?attrib-dyn-rl21 rotates))
               )
            ))
            (bind ?proc-stop-rl21
            (Process Stop-Roller
               (ProcStep
                  ?stat-rota-rl21
                  (Condition TRUE)
                  (Apply send ?attrib-dyn-rl21 put-Value static)
                  ?stat-stop-rl21
               )
```

```
            ))
         )
      ))

   ;=================================================
   ; ANOTHER ROLLER (reader)
   ;=================================================
   (bind ?roller2
      (Design Roller_2
         (Attributes
            (bind ?attrib-dyn-rl22(Attribute dynamics))
         )
         FALSE ;// no components
         FALSE ;// no local relations
         (Behaviors
            (bind ?proc-strt-rl22
            (Process Start-Roller
               (ProcStep
                  (bind ?stat-stop-rl22
                  (ObjectState ?attrib-dyn-rl22 static))
                  (Condition TRUE)
                  (Apply send ?attrib-dyn-rl22 put-Value rotates)
                  (bind ?stat-rota-rl22
                  (ObjectState ?attrib-dyn-rl22 rotates))
               )
            ))
            (bind ?proc-stop-rl22
            (Process Stop-Roller
               (ProcStep
                  ?stat-rota-rl22
                  (Condition TRUE)
                  (Apply send ?attrib-dyn-rl22 put-Value static)
                  ?stat-stop-rl22
               )
            ))
         )
      ))

   ;=================================================
   ; DOCUMENT (reader)
   ;=================================================
   (bind ?document2
      (Design Document_2
         (Attributes
            (bind ?attrib-dyn-dc2(Attribute dynamics))
         )
         FALSE ;// no components
         FALSE ;// no local relations
         (Behaviors
            (bind ?proc-strt-dc2
```

```
            (Process Start-Document
               (ProcStep
                  (bind ?stat-stop-dc2
                  (ObjectState ?attrib-dyn-dc2 static))
                  (Condition TRUE)
                  (Apply send ?attrib-dyn-dc2 put-Value translates)
                  (bind ?stat-trns-dc2
                  (ObjectState ?attrib-dyn-dc2 translates))
               )
            ))
            (bind ?proc-stop-dc2
            (Process Stop-Document
               (ProcStep
                  ?stat-trns-dc2
                  (Condition TRUE)
                  (Apply send ?attrib-dyn-dc2 put-Value static)
                  ?stat-stop-dc2
               )
            ))
         )
      ))
) ;// end Components

;=RELATIONS=============================================

(bind ?rels-scan
   (Relations ;// of Scanning Part
      (bind ?touch-scan
      (Relation Touching ?stepping-motor2
               (Relation And
                  (Relation Touching ?roller1 ?document2)
                  (Relation Touching
                        ?platen-roller2 ?document2 ?image-sensor2)
                  (Relation Touching ?roller2 ?document2)
      )))
))

;=BEHAVIORS=============================================

(Behaviors ;// of Scanning Part
   (bind ?start-scan
   (Process Start-Scanning
      (ProcStep
         (bind ?stat-stop-scan
         (ObjectState ?stat-stop-sm2
                     ?stat-stop-pr2
                     ?stat-nord-is2
                     ?stat-stop-rl21
                     ?stat-stop-rl22
                     ?stat-stop-dc2))
```

```
            ?touch-scan
            (Process Start-Components
               ?proc-strt-sm2
               ?proc-strt-pr2
               ?proc-strt-is2
               ?proc-strt-rl21
               ?proc-strt-rl22
               ?proc-strt-dc2
            )
            (bind ?stat-strt-scan
            (ObjectState ?stat-rota-sm2
                         ?stat-rota-pr2
                         ?stat-read-is2
                         ?stat-rota-rl21
                         ?stat-rota-rl22
                         ?stat-trns-dc2))
            )
      ))
      (bind ?stop-scan
      (Process Stop-Scanning
         (ProcStep
            ?stat-strt-scan
            ?touch-scan
            (Process Stop-Components
               ?proc-stop-sm2
               ?proc-stop-pr2
               ?proc-stop-is2
               ?proc-stop-rl21
               ?proc-stop-rl22
               ?proc-stop-dc2
            )
            ?stat-stop-scan
         )
      ))
    )

   ;=FUNCTIONS=============================================

   (Uses
      (bind ?startScan (Process Start-Scanning ))
      (bind ?stopScan  (Process Stop-Scanning ))
   )

   ) ;// end Image Sensing Part
) ;// (end bind Image Sensing Part)
)

;================================================================
; end Components of Personal Fax
;================================================================
```

```
;=FAX RELATIONS=====================================================

(Relations
   (Relation And rels-print ?rels-scan)
)


;=FAX BEHAVIORS=====================================================

(Behaviors
   (bind ?send
      (Process Send-Behavior
         (ProcStep
            ?stat-stop-scan
            (Condition TRUE)
            ?start-scan
            ?stat-strt-scan
         )
         (ProcStep
            ?stat-strt-scan
            (Condition TRUE)
            ?stop-scan
            ?stat-stop-scan
         )
      ))
   (bind ?receive
      (Process Receive-Behavior
         (ProcStep
            ?stat-stop-print
            (Condition TRUE)
            ?start-print
            ?stat-strt-print
         )
         (ProcStep
            ?stat-strt-print
            (Condition TRUE)
            ?stop-print
            ?stat-stop-print
         )
      ))
)
```

```
  ;=FAX FUNCTIONS=====================================================

     (Uses
        (bind ?send (Process Send-Behavior ))
        (bind ?receive  (Process Receive-Behavior ))
     )

) ;// end Personall Fax Description
```

Appendix B

This appendix illustrates the use of the simplification system by presenting the printout of an interaction during a problem solving session.

```
SIMPLIFY ( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
RETRIEVE SIMILAR to: ( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )

MATCHES FOUND: 387 of which
+ same relation:   51
+ common ancestor: 168
+ same arity:168
PROPAGATE MATCHES
.....................................................
COMPUTE SCORES ..........................................
   SORT MATCHES

Match objects [( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) <=>
( + ( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) ) ( * ( - a ( * 2 b ) ) z )
)] (16)
```

```
Match objects [( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) ) <=>
( + ( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) ) ( * ( - a ( * 2 b ) ) z )
)] (16)

Match objects [( * x ( + z ( * u v ) ) ) <=>
( * 1 ( - a ( * 2 b ) ) )] (16)

Match objects [( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) ) <=>
( * 1 ( - a ( * 2 b ) ) )] (16)

Match objects [( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) <=>
( * 1 ( - a ( * 2 b ) ) )] (16)

Match objects [( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) <=>
( + 0 ( - a ( * 2 b ) ) )] (16)

Match objects [( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) ) <=>
( + 0 ( - a ( * 2 b ) ) )] (16)

Match objects [( * x ( + z ( * u v ) ) ) <=>
( + 0 ( - a ( * 2 b ) ) )] (16)

Match objects [( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) <=>
( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) )] (12)

Match objects [( + z ( * u v ) ) <=>
( + ( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) ) ( * ( - a ( * 2 b ) ) z )
)] (11)

Match objects [( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) ) <=>
```

```
( - a ( * 2 b ) )] (11)

Match objects [( * x ( + z ( * u v ) ) ) <=>
( + ( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) ) ( * ( - a ( * 2 b ) ) z )
)] (11)

Match objects [( + z ( * u v ) ) <=>
( * 1 ( - a ( * 2 b ) ) )] (11)

Match objects [( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) <=>
( * ( - a ( * 2 b ) ) 1 )] (11)

Match objects [( + z ( * u v ) ) <=>
( + 0 ( - a ( * 2 b ) ) )] (11)

Match objects [( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) <=>
( + ( - a ( * 2 b ) ) 0 )] (11)

Match objects [( * x ( + z ( * u v ) ) ) <=>
( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) )] (7)

Match objects [( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) ) <=>
( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) )] (7)

Match objects [( + z ( * u v ) ) <=>
( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) )] (7)

Match objects [( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) <=>
( + d e )] (6)
```

```
Match objects [( + z ( * u v ) ) <=>
( + d e )] (6)

Match objects [( * u v ) <=> ( * ( - a ( * 2 b ) )
( * 3 c ( + d e ) ) )] (6)

Match objects [( * x y ) <=>
( * ( - a ( * 2 b ) ) z )] (6)

Match objects [( * x y ) <=>
( * 2 b )] (6)

Match objects [( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) ) <=>
( * ( - a ( * 2 b ) ) z )] (6)

Match objects [( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) ) <=>
( * 2 b )] (6)

Match objects [( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) ) <=>
( + d e )] (6)

Match objects [( * x ( + z ( * u v ) ) ) <=>
( + d e )] (6)

Match objects [( * u v ) <=>
( + ( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) ) ( * ( - a ( * 2 b ) ) z )
)] (6)

Match objects [( * u v ) <=>
( + d e )] (6)
```

```
Match objects [( * x y ) <=>
( + ( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) ) ( * ( - a ( * 2 b ) ) z )
)] (6)

Match objects [( * x y ) <=>
( + d e )] (6)

Match objects [( * u v ) <=>
( * 1 ( - a ( * 2 b ) ) )] (6)

Match objects [( * x y ) <=>
( * 1 ( - a ( * 2 b ) ) )] (6)

Match objects [( * x ( + z ( * u v ) ) ) <=>
( * ( - a ( * 2 b ) ) 1 )] (6)

Match objects [( * u v ) <=>
( * ( - a ( * 2 b ) ) 1 )] (6)

Match objects [( * x y ) <=>
( * ( - a ( * 2 b ) ) 1 )] (6)

Match objects [( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) ) <=>
( * ( - a ( * 2 b ) ) 1 )] (6)

Match objects [( + z ( * u v ) ) <=>
( * ( - a ( * 2 b ) ) 1 )] (6)

Match objects [( * u v ) <=>
```

```
( + 0 ( - a ( * 2 b ) ) )] (6)

Match objects [( * x y ) <=>
( + 0 ( - a ( * 2 b ) ) )] (6)

Match objects [( + z ( * u v ) ) <=>
( + ( - a ( * 2 b ) ) 0 )] (6)

Match objects [( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) ) <=>
( + ( - a ( * 2 b ) ) 0 )] (6)

Match objects [( * x ( + z ( * u v ) ) ) <=>
( + ( - a ( * 2 b ) ) 0 )] (6)

Match objects [( * u v ) <=>
( + ( - a ( * 2 b ) ) 0 )] (6)

Match objects [( * x y ) <=>
( + ( - a ( * 2 b ) ) 0 )] (6)

BEST MATCHES
   APPLY simpler ( ( + ( * 3 ( - a ( * 2 b ) ) ) ( * ( - a 2 ) ( * ( - a
( * 2 b ) ) ( + ( * 3 c ( + d e ) ) z ) ) ) ) ( + ( * 3 ( - a ( * 2 b ) ) )
) ( * ( - a 2 ) ( + ( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) ) ( * ( - a
( * 2 b ) ) z ) ) ) ) )
   TO    Match objects [( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) <=> ( +
( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) ) ( * ( - a ( * 2 b ) ) z ) )]
(16)
   IN ( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) => ( * x ( + y ( + z ( * u v )
) ) )
```

```
   SIMPLIFIED OBJECT: ( / x ( * x ( + y ( + z ( * u v ) ) ) ) )
VALID SOLUTION


   APPLY simpler ( ( + ( * 3 ( - a ( * 2 b ) ) ) ( * ( - a 2 ) ( * ( - a
( * 2 b ) ) ( + ( * 3 c ( + d e ) ) z ) ) ) ) ( + ( * 3 ( - a ( * 2 b ) )
) ( * ( - a 2 ) ( + ( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) ) ( * ( - a
( * 2 b ) ) z ) ) ) ) )
   TO    Match objects [( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
<=> ( + ( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) ) ( * ( - a ( * 2 b ) )
z ) )] (16)
   IN ( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) ) => ( * ( * x y ) ( + (
* 3 gen7097 ( + gen7101 gen7105 ) ) ( * x ( + z ( * u v ) ) ) ) )
   SIMPLIFIED OBJECT: ( * ( * x y ) ( + ( * 3 gen7097 ( + gen7101 gen7105
) ) ( * x ( + z ( * u v ) ) ) ) )
NOT VALID


   APPLY simpler ( ( - a ( * 2 b ) ) ( * 1 ( - a ( * 2 b ) ) ) )
   TO    Match objects [( * x ( + z ( * u v ) ) ) <=> ( * 1 ( - a ( * 2 b
) ) )] (16)
   IN ( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
( * x ( + z ( * u v ) ) ) => ( + z ( * u v ) )
   SIMPLIFIED OBJECT: ( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
NOT VALID


   APPLY simpler ( ( - a ( * 2 b ) ) ( * 1 ( - a ( * 2 b ) ) ) )
   TO    Match objects [( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
<=> ( * 1 ( - a ( * 2 b ) ) )] (16)
   IN ( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) ) => ( + ( * x y ) ( * x
( + z ( * u v ) ) ) )
   SIMPLIFIED OBJECT: ( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
```

```
NOT VALID

   APPLY simpler ( ( - a ( * 2 b ) ) ( * 1 ( - a ( * 2 b ) ) ) )
   TO    Match objects [( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) <=> ( *
1 ( - a ( * 2 b ) ) )] (16)
   IN ( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) => ( * x ( + z ( * u v ) ) )
   SIMPLIFIED OBJECT: ( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
NOT VALID


   APPLY simpler ( ( - a ( * 2 b ) ) ( + 0 ( - a ( * 2 b ) ) ) )
   TO    Match objects [( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) <=> ( +
0 ( - a ( * 2 b ) ) )] (16)
   IN ( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) => ( * x ( + z ( * u v ) ) )
   SIMPLIFIED OBJECT: ( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
NOT VALID


   APPLY simpler ( ( - a ( * 2 b ) ) ( + 0 ( - a ( * 2 b ) ) ) )
   TO    Match objects [( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
<=> ( + 0 ( - a ( * 2 b ) ) )] (16)
   IN ( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) ) => ( + ( * x y ) ( * x
( + z ( * u v ) ) ) )
   SIMPLIFIED OBJECT: ( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
NOT VALID


   APPLY simpler ( ( - a ( * 2 b ) ) ( + 0 ( - a ( * 2 b ) ) ) )
```

```
   TO    Match objects [( * x ( + z ( * u v ) ) ) <=> ( + 0 ( - a ( * 2 b
) ) )] (16)
   IN ( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
( * x ( + z ( * u v ) ) ) => ( + z ( * u v ) )
   SIMPLIFIED OBJECT: ( / x ( + ( * x y ) ( * x ( + z ( * u v ) ) ) ) )
NOT VALID
```

Appendix C

This appendix contains the CLIPS class definitions for the internal representations of objects and relations.

```
;*****************************************************************
; class: ATTRIBUTE DEFINITIONS
;*****************************************************************

(defclass defAttribute (is-a USER)
                    (role concrete) (pattern-match reactive)

  (slot Name        (create-accessor read-write))
  (slot Type        (create-accessor read-write))
  (slot Constraints (create-accessor read-write) (default TRUE))
  (slot Default     (create-accessor read-write))
)

;*****************************************************************
; class: ATTRIBUTE
;*****************************************************************

(defclass Attribute (is-a USER)
                    (role concrete) (pattern-match reactive)

  (slot Name        (create-accessor read-write))
  (slot Value       (create-accessor read-write))
  (slot Definition  (create-accessor read-write))
  (slot Of          (create-accessor read-write) (default FALSE))
  (slot Object      (create-accessor read-write) (default FALSE))
)
```

```
;*****************************************************************
; class: SET OF ATTRIBUTE
;*****************************************************************

(defclass setOfAttributes (is-a USER)
                      (role concrete) (pattern-match reactive)

  (multislot Members   (create-accessor read-write) (default (create$)))
  (slot      Count     (create-accessor read-write) (default 0))
  (slot      Object    (create-accessor read-write) (default FALSE))
)

;*****************************************************************
; class: RELATION DEFINITION
;*****************************************************************

(defclass defRelation (is-a USER)
                      (role concrete) (pattern-match reactive)

  (slot Name           (create-accessor read-write))
  (slot Signature      (create-accessor read-write))
  (slot Ordered        (create-accessor read-write) (default FALSE))
  (slot SupRel         (create-accessor read-write) (default nil))
  (multislot SubRels   (create-accessor read-write) (default (List)))
  (multislot Members    (create-accessor read-write) (default (create$
))))
)

;*****************************************************************
; class RELATION INSTANCE
;*****************************************************************

(defclass Relation (is-a USER)
                    (role concrete) (pattern-match reactive)

  (slot Name         (create-accessor read-write))
  (slot Arity        (create-accessor read-write))
  (slot Members      (create-accessor read-write))
  (slot Attributes   (create-accessor read-write))
  (slot Definition   (create-accessor read-write))
  (slot Root         (create-accessor read-write) (default FALSE))
  (slot Object       (create-accessor read-write) (default FALSE))
  (slot InRelation   (create-accessor read-write) (default FALSE))
  (slot CntRelsIn    (create-accessor read-write) (default 0))
)


;*****************************************************************
; class OBJECT
```

```
;*****************************************************************

(defclass Object (is-a USER)
                   (role concrete) (pattern-match reactive)

  (slot Name         (create-accessor read-write))
  (slot Class        (create-accessor read-write) (default nil))
  (slot ExternalRels (create-accessor read-write) (default FALSE))
  (slot Attributes   (create-accessor read-write) (default FALSE))
  (slot Composed     (create-accessor read-write) (default FALSE))
  (slot LocalRels    (create-accessor read-write) (default FALSE))
  (slot Root         (create-accessor read-write) (default FALSE))
  (slot Object       (create-accessor read-write) (default FALSE))
  (slot InRelation   (create-accessor read-write) (default FALSE))

)

;*****************************************************************
; class OBJECT STATE
;*****************************************************************

(defclass ObjectState (is-a USER)
                        (role concrete) (pattern-match reactive)

  (multislot Attributes (create-accessor read-write))
  (multislot Values     (create-accessor read-write))
)
```

Appendix D

This appendix shows the contents of the simplification database used for running the demonstration examples presented in Chapter 7.

```
SIMPLIFICATIONS CURRENTLY LOADED
     simpler ( x ( + x 0 ) )
             - explanation:
                ( replace ( + x 0 ) x )
             - relevants:
                ( + x 0 ) / Object
                x / Object
                0 / Object
                + ( x 0 ) / Relation

          simpler ( x ( + 0 x ) )
             - explanation:
                ( replace ( + 0 x ) x )
             - relevants:
                ( + 0 x ) / Object
                0 / Object
                x / Object
                + ( 0 x ) / Relation

          simpler ( x ( + x 0 0 ) )
             - explanation:
                ( replace ( + x 0 0 ) x )
             - relevants:
                ( + x 0 0 ) / Object
                x / Object
                0 / Object
```

```
      0 / Object
      + ( x 0 0 ) / Relation

simpler ( x ( + 0 x 0 ) )
   - explanation:
      ( replace ( + 0 x 0 ) x )
   - relevants:
      ( + 0 x 0 ) / Object
      0 / Object
      x / Object
      0 / Object
      + ( 0 x 0 ) / Relation

simpler ( x ( + 0 0 x ) )
   - explanation:
      ( replace ( + 0 0 x ) x )
   - relevants:
      ( + 0 0 x ) / Object
      0 / Object
      0 / Object
      x / Object
      + ( 0 0 x ) / Relation

simpler ( ( * x y ) ( * x ( + 0 y 0 ) ) )
   - explanation:
      ( replace ( + 0 y 0 ) y )
   - relevants:
      ( + 0 y 0 ) / Object
      0 / Object
      y / Object
      0 / Object
      + ( 0 y 0 ) / Relation

simpler ( x ( - x 0 ) )
   - explanation:
      ( replace ( - x 0 ) x )
   - relevants:
      ( - x 0 ) / Object
      x / Object
      0 / Object
      - ( x 0 ) / Relation

simpler ( x ( - x 0 0 ) )
   - explanation:
      ( replace ( - x 0 0 ) x )
   - relevants:
      ( - x 0 0 ) / Object
      x / Object
      0 / Object
      0 / Object
```

```
      - ( x 0 0 ) / Relation

simpler ( ( * x y ) ( * x ( - y 0 ) ) )
   - explanation:
      ( replace ( - y 0 ) y )
   - relevants:
      ( - y 0 ) / Object
      y / Object
      0 / Object
      - ( y 0 ) / Relation

simpler ( x ( * x 1 ) )
   - explanation:
      ( replace ( * x 1 ) x )
   - relevants:
      ( * x 1 ) / Object
      x / Object
      1 / Object
      * ( x 1 ) / Relation

simpler ( x ( * 1 x ) )
   - explanation:
      ( replace ( * 1 x ) x )
   - relevants:
      ( * 1 x ) / Object
      1 / Object
      x / Object
      * ( 1 x ) / Relation

simpler ( x ( * x 1 1 ) )
   - explanation:
      ( replace ( * x 1 1 ) x )
   - relevants:
      ( * x 1 1 ) / Object
      x / Object
      1 / Object
      1 / Object
      * ( x 1 1 ) / Relation

simpler ( x ( * 1 x 1 ) )
   - explanation:
      ( replace ( * 1 x 1 ) x )
   - relevants:
      ( * 1 x 1 ) / Object
      1 / Object
      x / Object
      1 / Object
      * ( 1 x 1 ) / Relation

simpler ( x ( * 1 1 x ) )
```

```
   - explanation:
      ( replace ( * 1 1 x ) x )
   - relevants:
      ( * 1 1 x ) / Object
      1 / Object
      1 / Object
      x / Object
      * ( 1 1 x ) / Relation

simpler ( ( + x y ) ( + x ( * 1 y 1 ) ) )
   - explanation:
      ( replace ( * 1 y 1 ) y )
   - relevants:
      ( * 1 y 1 ) / Object
      1 / Object
      y / Object
      1 / Object
      * ( 1 y 1 ) / Relation

simpler ( x ( / x 1 ) )
   - explanation:
      ( replace ( / x 1 ) x )
   - relevants:
      ( / x 1 ) / Object
      x / Object
      1 / Object
      / ( x 1 ) / Relation

simpler ( x ( / x 1 1 ) )
   - explanation:
      ( replace ( / x 1 1 ) x )
   - relevants:
      ( / x 1 1 ) / Object
      x / Object
      1 / Object
      1 / Object
      / ( x 1 1 ) / Relation

simpler ( ( + x y ) ( + x ( / y 1 ) ) )
   - explanation:
      ( replace ( / y 1 ) y )
   - relevants:
      ( / y 1 ) / Object
      y / Object
      1 / Object
      / ( y 1 ) / Relation

simpler ( ( * 7 x y ) ( * 7 ( + 0 x 0 ) y ) )
   - explanation:
      ( replace ( + 0 x 0 ) x )
```

```
         - relevants:
            ( + 0 x 0 ) / Object
            0 / Object
            x / Object
            0 / Object
            + ( 0 x 0 ) / Relation

   simpler ( ( * ( - a 2 ) ( * x ( + y z ) ) )
         ( * ( - a 2 ) ( + ( * x y ) ( * x z ) ) ) )
         - explanation:
            ( replace ( + ( * x y ) ( * x z ) ) ( * x ( + y z ) ) )
         - relevants:
            ( + ( * x y ) ( * x z ) ) / Object
            ( * x y ) / Object
            x / Object
            y / Object
            * ( x y ) / Relation
            ( * x z ) / Object
            x / Object
            z / Object
            * ( x z ) / Relation
            + ( ( * x y ) ( * x z ) ) / Relation

      simpler ( ( - a ( * 2 b ) ) ( + ( - a ( * 2 b ) ) 0 ) )
         - explanation:
            ( replace ( + ( - a ( * 2 b ) ) 0 ) ( - a ( * 2 b ) ) )
         - relevants:
            ( + ( - a ( * 2 b ) ) 0 ) / Object
            ( - a ( * 2 b ) ) / Object
            a / Object
            ( * 2 b ) / Object
            2 / Object
            b / Object
            * ( 2 b ) / Relation
            - ( a ( * 2 b ) ) / Relation
            0 / Object
            + ( ( - a ( * 2 b ) ) 0 ) / Relation

      simpler ( ( - a ( * 2 b ) ) ( + 0 ( - a ( * 2 b ) ) ) )
         - explanation:
            ( replace ( + 0 ( - a ( * 2 b ) ) ) ( - a ( * 2 b ) ) )
         - relevants:
            ( + 0 ( - a ( * 2 b ) ) ) / Object
            0 / Object
            ( - a ( * 2 b ) ) / Object
            a / Object
            ( * 2 b ) / Object
            2 / Object
            b / Object
            * ( 2 b ) / Relation
```

```
        - ( a ( * 2 b ) ) / Relation
        + ( 0 ( - a ( * 2 b ) ) ) / Relation

simpler ( ( - a ( * 2 b ) ) ( + ( - a ( * 2 b ) ) 0 0 ) )
   - explanation:
      ( replace ( + ( - a ( * 2 b ) ) 0 0 ) ( - a ( * 2 b ) ) )
   - relevants:
      ( + ( - a ( * 2 b ) ) 0 0 ) / Object
      ( - a ( * 2 b ) ) / Object
      a / Object
      ( * 2 b ) / Object
      2 / Object
      b / Object
      * ( 2 b ) / Relation
      - ( a ( * 2 b ) ) / Relation
      0 / Object
      0 / Object
      + ( ( - a ( * 2 b ) ) 0 0 ) / Relation

simpler ( ( + 0 ( - a ( * 2 b ) ) ) ( + ( - a ( * 2 b ) ) 0 0 ) )
   - explanation:
      ( remove 0 )
   - relevants:
      0 / Object

simpler ( ( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) )
  ( * ( - a ( * 2 b ) ) ( + 0 ( * 3 c ( + d e ) ) 0 ) ) )
   - explanation:
     ( replace ( + 0 ( * 3 c ( + d e ) ) 0 ) ( * 3 c ( + d e ) ) )
   - relevants:
      ( + 0 ( * 3 c ( + d e ) ) 0 ) / Object
      0 / Object
      ( * 3 c ( + d e ) ) / Object
      3 / Object
      c / Object
      ( + d e ) / Object
      d / Object
      e / Object
      + ( d e ) / Relation
      * ( 3 c ( + d e ) ) / Relation
      0 / Object
      + ( 0 ( * 3 c ( + d e ) ) 0 ) / Relation

simpler ( ( - a ( * 2 b ) ) ( * ( - a ( * 2 b ) ) 1 ) )
   - explanation:
      ( replace ( * ( - a ( * 2 b ) ) 1 ) ( - a ( * 2 b ) ) )
   - relevants:
      ( * ( - a ( * 2 b ) ) 1 ) / Object
      ( - a ( * 2 b ) ) / Object
      a / Object
```

```
          ( * 2 b ) / Object
          2 / Object
          b / Object
          * ( 2 b ) / Relation
          - ( a ( * 2 b ) ) / Relation
          1 / Object
          * ( ( - a ( * 2 b ) ) 1 ) / Relation

       simpler ( ( - a ( * 2 b ) ) ( * 1 ( - a ( * 2 b ) ) ) )
          - explanation:
             ( replace ( * 1 ( - a ( * 2 b ) ) ) ( - a ( * 2 b ) ) )
          - relevants:
             ( * 1 ( - a ( * 2 b ) ) ) / Object
             1 / Object
             ( - a ( * 2 b ) ) / Object
             a / Object
             ( * 2 b ) / Object
             2 / Object
             b / Object
             * ( 2 b ) / Relation
             - ( a ( * 2 b ) ) / Relation
             * ( 1 ( - a ( * 2 b ) ) ) / Relation

      simpler ( ( * 1 ( - a ( * 2 b ) ) ) ( * 1 ( - a ( * 2 b ) ) 1 ) )
          - explanation:
             ( remove 1 )
          - relevants:
             1 / Object

       simpler ( ( - a ( * 2 b ) ) ( * 1 1 ( - a ( * 2 b ) ) ) )
          - explanation:
             ( replace ( * 1 1 ( - a ( * 2 b ) ) ) ( - a ( * 2 b ) ) )
          - relevants:
             ( * 1 1 ( - a ( * 2 b ) ) ) / Object
             1 / Object
             1 / Object
             ( - a ( * 2 b ) ) / Object
             a / Object
             ( * 2 b ) / Object
             2 / Object
             b / Object
             * ( 2 b ) / Relation
             - ( a ( * 2 b ) ) / Relation
             * ( 1 1 ( - a ( * 2 b ) ) ) / Relation

       simpler ( ( + ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) )
         ( + ( - a ( * 2 b ) ) ( * 1 ( * 3 c ( + d e ) ) 1 ) ) )
          - explanation:
            ( replace ( * 1 ( * 3 c ( + d e ) ) 1 ) ( * 3 c ( + d e ) ) )
          - relevants:
```

```
        ( * 1 ( * 3 c ( + d e ) ) 1 ) / Object
        1 / Object
        ( * 3 c ( + d e ) ) / Object
        3 / Object
        c / Object
        ( + d e ) / Object
        d / Object
        e / Object
        + ( d e ) / Relation
        * ( 3 c ( + d e ) ) / Relation
        1 / Object
        * ( 1 ( * 3 c ( + d e ) ) 1 ) / Relation

simpler ( ( * 7 ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) )
  ( * 7 ( + 0 ( - a ( * 2 b ) ) 0 ) ( * 3 c ( + d e ) ) ) )
    - explanation:
       ( replace ( + 0 ( - a ( * 2 b ) ) 0 ) ( - a ( * 2 b ) ) )
    - relevants:
       ( + 0 ( - a ( * 2 b ) ) 0 ) / Object
       0 / Object
       ( - a ( * 2 b ) ) / Object
       a / Object
       ( * 2 b ) / Object
       2 / Object
       b / Object
       * ( 2 b ) / Relation
       - ( a ( * 2 b ) ) / Relation
       0 / Object
       + ( 0 ( - a ( * 2 b ) ) 0 ) / Relation

simpler ( ( + ( * 3 ( - a ( * 2 b ) ) )
        ( * ( - a 2 )
        ( * ( - a ( * 2 b ) ) ( + ( * 3 c ( + d e ) ) z ) ) ) )
  ( + ( * 3 ( - a ( * 2 b ) ) )
        ( * ( - a 2 )
         ( + ( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) )
              ( * ( - a ( * 2 b ) ) z ) ) ) ) )
    - explanation:
       ( replace ( + ( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) )
         ( * ( - a ( * 2 b ) ) z ) )
        ( * ( - a ( * 2 b ) ) ( + ( * 3 c ( + d e ) ) z ) ) )
    - relevants:
       ( + ( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) )
   ( * ( - a ( * 2 b ) ) z ) ) / Object
       ( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) ) / Object
       ( - a ( * 2 b ) ) / Object
       a / Object
       ( * 2 b ) / Object
       2 / Object
       b / Object
```

```
    * ( 2 b ) / Relation
    - ( a ( * 2 b ) ) / Relation
    ( * 3 c ( + d e ) ) / Object
    3 / Object
    c / Object
    ( + d e ) / Object
    d / Object
    e / Object
    + ( d e ) / Relation
    * ( 3 c ( + d e ) ) / Relation
    * ( ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) ) / Relation
    ( * ( - a ( * 2 b ) ) z ) / Object
    ( - a ( * 2 b ) ) / Object
    a / Object
    ( * 2 b ) / Object
    2 / Object
    b / Object
    * ( 2 b ) / Relation
    - ( a ( * 2 b ) ) / Relation
    z / Object
    * ( ( - a ( * 2 b ) ) z ) / Relation
    + ( ( * ( - a ( * 2 b ) ) ( * 3 c ( + d e ) ) )
( * ( - a ( * 2 b ) ) z ) ) / Relation
```