PROGRAMMING PARAMETERISED GEOMETRIC OBJECTS IN A VISUAL DESIGN LANGUAGE

by

Omid Banyasad

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

at

Dalhousie University
Halifax, Nova Scotia
June 2006

DALHOUSIE UNIVERSITY

DATE: June 12, 2006

AUTHOR: Omid Banyasad

TITLE: PROGRAMMING PARAMETERISED GEOMETRIC OBJECTS IN A VISUAL DESIGN LANGUAGE

DEPARTMENT OR SCHOOL: Faculty of Computer Science

DEGREE: PhD CONVOCATION: October YEAR: 2006

Table of Contents

List of Tables

List Of Figures

Abstract

A homogeneous view of the components of a parameterised structured design and the programming constructs used for its creation in a design environment has been the major motivation for creating Language for Structured Design (LSD), a visual design language for programming parameterised geometric objects. LSD is based on Lograph, a visual logic programming language, and therefore deals with the syntactical aspects of integrating solids in the programming world, but has no notion of solids in the design world. Hence, LSD needs to utilise a capable solid modeling package in order to achieve its objectives. In another thread, as a visual programming language LSD could substantially benefit from a carefully designed visual programming environment.

Accordingly, the research reported here proceeds on two fronts to build the backbone of a debugging environment for LSD. First, we study the foundations of a solid modeling system to serve as the back-end of LSD. Second, we build the mechanics of a debugging environment to utilise the intended solid modeler in conjunction with the logic aspects of LSD programs.

To achieve our first objective, we build on the previous formal model of solids for LSD to support the computation of sample looks of objects, required for their depiction in LSD programs and during the execution of LSD, formally define single and multiple behaviours for solids, and the motion of solids. We also show how our formal model can be put into practice by developing a translator which translates solids created in LSD to their representations in the PLaSM language.

In order to achieve our second goal, we study certain practical aspects of Lograph which will serve as the underlying language for LSD. This includes extending Lograph to obtain Lograph+C which supports constraint processing, devising an execution mechanism for Lograph and designing an interpreter engine to provide desirable debugging features such as single step execution of LSD programs in both forward and reverse directions, just-in-time programming, lazy design component selection and undo of execution steps. We also present an automatic layout algorithm for dynamically adjusting the layout of queries during animated execution.

List of Abbreviations Used

B-Rep	Boundary Representation
CAD	Computer-Aided Design
CLP	Constraint Logic Programming
CP	Constraint Programming
CSG	Constructive Solid Geometry
CSP	Constraint Satisfaction Problem
E-Component	Explicit Component
I-Component	Implicit Component
LSD	Language for Structured Design
PBD	Programming By Demonstration
PIE	Progressive Interval Enumeration
PLaSM	Programming Language for Symbolic Modeling
RLE	Run Length Encoding

Acknowledgments

I am grateful to my advisor, professor Philip Cox, for his willingness to let me explore many unfamiliar and interesting research fields of my choosing. He allowed me to search and find my own space within the scope of the suggested topic and most importantly let me explore the field at my own pace. I am also grateful to him for the level of patience that he showed during the last year of my work especially at times when patience was not abundant. He continuously showed interest in new ideas and left no stone unturned when I sought his feedback. He carefully read and thoroughly commented on every bit of this thesis and made the transformation of this thesis to the final version from the initial draft a rewarding experience.

I would like to thank Paolo Bottoni for agreeing to serve as my external examiner and for traveling to Halifax from Rome on such a short notice. Andrew Rau-Chaplin and Trevor Smedley also should be mentioned for agreeing to serve as readers and members of my examining committee.

My deepest gratitude goes to my parents, Ali Baniasad and Zahra Mehri Sarvi. They have been the most important source of encouragement and support throughout my studies. Had it not been for their patience, understanding, steadfast support, and unconditional love, this work would have never been completed.

My brothers, Madjid and Mahdad, and my sisters, Parisa and Poupak, have played important roles in helping me during my studies through their continuous support and words of encouragement.

One of my peers, Simon Gauvin, should also be mentioned for being a good friend and for many interesting and lively discussions on a variety of subjects over the past few years.

1 Introduction

Design is one of the most significant human endeavours. It is an essential part of many activities such as creating buildings, manufacturing cars, planning landscapes, and making tools. It is also an integral part of many other practices where design subjects are not of a physical nature: for example, software engineering and network protocols.

Many human practices include design in some form. Design may involve several related activities, depending on the domain in which it is practised, such as planning, problem solving, testing, and prototyping. In most domains, design is a sequence of separate processes performed repeatedly to obtain desired results. The result of each cycle is used to improve the next cycle, and the process continues until the result converges to a state that satisfies the exact specifications. In more relaxed scenarios, the design specification is satisfied when the difference between the result and the specification becomes smaller than some tolerable error margin.

In some domains, design is the process of arranging a set of existing components to serve a specific purpose. For example, interior design deals with improving the quality of life and protecting the health, safety, and welfare of the occupants of an interior space. This is often achieved through the arrangement of interior components of a house or office building to achieve efficient use of space, light, heat and possibly other criteria. In other domains, design may include plans for creating new components as well as the relationship between those components. Building structures and manufacturing mechanical devices are two examples of such domains.

Design is part of many aspects of life and includes many different processes. The complexity of a design activity may be as simple as drawing a few sketches on a piece of paper for a new fashion, or as complex as the architectural design of a skyscraper, an aircraft, or an offshore oil platform.

Our understanding and expectations from design may vary according to the context in which it is exercised. For example, the designer of a hand-held tool may focus only on the functionality and cost effectiveness of the design, while the designer of a large scale software system should not only consider the functionality of the system, but also take into account other criteria such as scalability, security, maintainability, and other software engineering considerations.

Although the many definitions of the term "design" do not agree in detail, a notion common to them all is that design is a purposeful and "goal-oriented" activity aimed at producing some *descriptions* from a set of *specifications.* The resulting descriptions express the required artifact so that it can be manufactured, assembled, or constructed [54]. The description of a design contains its elements and their relationships, also known as the *structure* of the design. Specifications are the formulation of the *functions* that the description of a design is desired to achieve.

Broadly speaking, design is a creative process, however, many researchers differentiate between creative and non-creative design [23,33,54]. Design means different things to different people. In this work, we use the term "design" in its engineering and manufacturing sense.

1.1 Design Tools

Design is an integral part of engineering practice. Hence, good engineering relies on good design [130]. Manufacturing mechanical devices and creating building structures are two familiar domains in which design plays an important role and can benefit from well crafted tools.

Today, most manufacturing companies are hard-pressed by competitive markets to deliver their products more cheaply and quickly. This forces manufacturing companies to seek more cost-effective designs and faster production techniques. The need for better and faster design, in turn, implies a need for better design tools and methodologies [130].

While some designs are purely manifestations of the designers' creativity and artistic abilities, for example jewelry or fashion design, others may require more analytical and computationally intensive approaches. In such domains appropriate design tools and methodologies can effectively elevate a designer's productivity as well as improving the quality of the design.

1.1.1 Computer-Aided Design

Today, computers "assist" us in virtually every aspect of life and design is no exception. Computer-Aided Design (CAD) systems are software packages that provide designers with tools and environments for creating structured designs such as buildings [19], mechanical devices [5], and integrated circuits [122].

CAD systems have come a long way since their first commercial appearance. Earlier drafting packages, despite their shortcomings for computationally intensive operations, revealed the potential strengths of CAD systems for the design of structured artifacts. The ever increasing power of microprocessors, advancements in the field of computer graphics and the evolution of high-resolution displays during the past decade, have made the emergence of a new family of CAD systems possible. Today, a sophisticated 3-dimensional CAD package equipped with a solid modeler, allows the design engineer to model a design specification and visualise it in detail long before the manufacturing process begins. Many CAD systems employ domain-specific design verifiers and performance analysers to verify the correctness of a design description and to measure its efficiency according to some set of criteria. This saves companies a substantial amount of time and financial resources by early detection of flaws and design deficiencies.

The process of designing a structured artifact involves five seemingly separate activities: defining the logic and the functional behaviours of the artifact, synthesis, modeling, prototyping, and analysis. As mentioned earlier, today's sophisticated CAD systems, along with complex solid modelers and performance analysers, can effectively ease the drawing, modeling, prototyping, and analysis process of a design activity. However, they still lack the appropriate means for designers to solve design problems and define the logic and the functional behaviours of an artifact within the design environment. Hence, we add to the list the need for problem-solving capabilities within the design environment.

The need for an integrated environment for drawing, editing, modeling, and prototyping structured artifacts is recognised within the CAD community. Researchers in the design field anticipate the emergence of a new family of integrated CAD systems which are capable of testing, prototyping and analysis of a design based on parametric solid modeling [54,130].

A successful integration of design tools should provide a smooth and seamless transition from one stage of a design to the next. In some cases it may even be possible to merge two stages into one. For example, a parameterised design environment for mechanical devices with capabilities for simulating kinematic behaviours of parts might be a valuable tool for proof-of-concept design, early demos for customers, or early detection of flaws in a design.

Portability of design is also an important issue in today's design engineering practice. Many large manufacturing companies such as Boeing have multiple branches across the globe. An integration of geometrical and functional characteristics of design components would allow designers in distant time zones to transfer designs along with their functional and behavioural characteristics from one centre to another with minimum overhead, allowing one team to continue the work from where another left off, potentially allowing companies to reduce design life cycles [130]. A programming approach to parametric design allows the transfer of design decisions along with the design itself over electronic communication media [93], facilitating collaborative and distributed design efforts [56].

1.1.2 Mechanical Design Versus Digital Design

Although there are many design tools for both electrical and mechanical engineering, it has often been noted that design methods and CAD tools for mechanical design are not as mature as those for digital design. Whitney associates this gap with the fundamental differences between the two domains, despite their superficial similarities, and concludes that design tools and methodologies for mechanical engineering cannot be like those for digital design because of these differences [132].

In addition to the fundamental differences between the two fields, Woodwark [134] notes that the principal extra difficulties in mechanical CAD are due to its 3D nature and the need for motion modeling. Secondary contributors include the requirements for load-bearing and deformations, as well as fatigue, vibration, and friction noted by Whitney.

There are, however, more optimistic views of design tools for mechanical systems. For example, while acknowledging many of the points raised by Whitney, Antonsson in [4] notes that Whitney's comparison focuses on a very specific subcategory of mechanical design and concludes that the difficulties in the field of mechanical engineering highlighted by Whitney, should be considered as motivating factors for many interesting research initiatives.

1.2 Design Languages

There are several programming languages tailored for computer-aided design. For example, AML/X [86] is a general purpose programming language aimed at tackling problems in robotics, computer-aided design, and machine vision. Glide [45], Euklid [48], and RAPT [99] are other examples of pioneering systems for design engineering. There are also many unnamed design languages such as [25,26,60]. PLaSM [92] is an example of more recent efforts for integrating programming capabilities in a design environment.

Since the ultimate goal of CAD systems is to assist designers, there have been attempts to describe what an ideal CAD system should provide to maximise such assistance. Unfortunately, we are not aware of similar prescriptions for an ideal design language. The following points describe some of the important features that an ideal design language should support, gathered together from various sources. We acknowledge that this list may not be exhaustive.

- Descriptive creation of objects rather than prescriptive. A design will contain high level specification of an object. The details (design descriptions) and how they are computed is of lesser importance to the designer.
- Design validation checking. In an ideal design language, validity checking should be the responsibility of the system.
- Integrating structure and functions [130]. An ideal design language should provide means for integrating the structure (geometry) and function of a design.
- Parameterisation. Designing families of solids rather than single designs.
- Maintainability. Debugging and editing should be easy and intuitive and should be performed at a high abstraction level.
- A design language should provide capabilities for creating complex objects from simpler ones. This might be achieved through some predefined operations and algorithms that can be performed on objects, or by allowing the designer to define new operations.
- Reusability. Designs created in a design language should be modular so that they can be reused.
- A viable design language should be scalable.

- Conciseness. In a concise design language, a design representation is small in size and therefore easy to read from and write to memory and also easy to transfer over networks [108].
- A useful and usable environment. Although the design language environment does not extend the language, a well designed environment could be as important as the design language itself. A usable environment is one that is easy to use and consistent, in which the role of language components and controls are easily inferred [58].
- Problem solving capabilities [94]. The programming capabilities of a design language should go beyond simple parameterisation of solids and embody capabilities for solving design problems in a homogeneous environment which does not require the user to switch between different, dissimilar representations of a design.

1.3 Background

The work presented here is part of a continuing project aimed at creating new software tools for "design engineering". Here, we recapitulate the previous work on this project. In the remaining chapters, when necessary, we will provide more detailed descriptions of the background work related to the issue at hand.

It is noted in [38] that the once distinct areas of software design and the design of other artifacts have in recent years begun to converge. The former evolved from completely textual languages, to languages which employ graphical representations, some almost exclusively; while the latter has evolved from purely graphical drafting software to packages with the programming capabilities necessary for the specification of complex parameterised designs. Some examples are the design of digital circuits [122] and buildings [107].

Visual software tools for designing solids, such as AutoCAD [5] and MicroStation [19], provide support for solid modeling using concrete visual representations of the components of the objects being designed. Typically, such systems also provide a textual programming language for designing classes of objects the exact nature of which depends on the values of parameters. For example, AutoCAD facilitates the integration of modules written in a textual programming language such as C, and supplies AutoLISP [7] as the standard programming language.

Tools which provide for the specification of parameterised designs are, however, either inadequate visual representations of textual languages, or as observed in [38], completely textual [122], or sharply divided into graphical drafting and textual programming [7,19]. An initial attempt to address these shortcomings added picture data to a visual dataflow programming language, together with operations for transforming pictures [116]. Although the resulting language provided visual representations of both algorithms and design objects, these two representations and the related activities of programming and drafting, were still quite separate.

In an effort to remove this distinction, a Language for Structured Design (LSD) was proposed in [38]. LSD is based on a visual logic programming language, Lograph [36], extended by the introduction of "explicit components", a logical manifestation of solid objects, and a new execution rule to combine them.

The original LSD has just one operation on solids. However, exactly what operations are required in a design language will depend on the domain of application, requiring LSD to be extensible. In order to extend LSD, however, it was first necessary to define the notions of "design space" and "solid" in just enough detail for their key characteristics to be captured in the revised language. Accordingly, a formal model characterising objects in a design space was presented in [37]. The proposed design space consists of a normal multi-dimensional geometric space augmented with extra dimensions representing properties. This model also included definition of operations on solids as well as the notion of selective interfaces that allows solids to declare what operations can be applied to them. These definitions were used to generalise LSD [37].

1.4 Research Objectives

The perceived need for a homogeneous view of the components of a structure and the programming constructs used for their parameterised creation in a design environment has been the major motivation for creating LSD. As noted in [38], LSD deals with the syntactical aspects of integrating solids into the programming world and has no notion of solids in the design world. Hence, an implementation of LSD needs to utilise a capable solid modeling package in order to achieve its objectives. By a "solid modeling package" we mean a solid modeling kernel extended with additional capabilities to provide services suitable for LSD. For example, the same way that Autodesk Inventor [6] extends the ACIS kernel from Spatial Technologies [32,120], and SolidWorks [119] or Bentley Systems' MicroStation [19] extend the Parasolid kernel [113] from Unigraphics.

In [38], simulation of the behaviour of solids was characterised as an important aspect of designing solids. In order to provide for the simulation of the behaviour of solids, a formal definition of behaviour is clearly required. We will extend the formal model in [37], originally used for the generalisation of LSD, to capture the notion of behaviour.

In another thread, we note that LSD, like any other Visual Programming Language (VPL) would substantially benefit from being implemented in a carefully designed environment. In fact, the programming environment can be as important as the VPL that it encapsulates. For example, an interactive debugger which allows a programmer to observe and analyse the behaviour of an LSD program at run-time through the use of features such as execution pause and dynamic execution rollback, would be a valuable part of an LSD implementation.

Our goal in the work reported on here, based on the above observations, has been to further the development of LSD with the long-term goal of implementing an LSD-based design environment. Accordingly, we have advanced on two fronts. First, we have considered issues, both practical and formal, involved with attaching a solid modelling kernel to an LSD implementation; and second, we have investigated the problems entailed in building a debugging environment in which to embed LSD and an associated solid modeler.

The solid modeler selected for an implementation must satisfy certain requirements, dictated by characteristics of the LSD language, and by the properties we would like to see in a debugging environment. These requirements are as follows

- Support for parameterised solids
  - Low-level (geometric) parameterisation. For example, parameterising the geometry of an explicit component such as length, width, or height of a block solid.
  - High-level parameterisation. For example, a "bearing" solid may become a roller-bearing or a ball-bearing based on the value of a parameter.
  - Iteration. For example, a rack with *n* tiers, or a cog with *n* teeth.
- Support for predefined and user-defined geometric operations on solids.
- Capability for reducing solids. Reduction of a solid refers to the process of finding a minimum set of parameters, an instantiation of which will produce an instance of the family of objects to which the solid corresponds. The reduction of a solid should also take into account the interfaces that the solid must expose, if any.
- Support for higher-level constraints beyond geometric ones. Constraints add an additional level of abstraction above parametric geometry and will also allow for a more descriptive approach to geometric programming vis-à-vis prescriptive.
- The solid modeler should maintain intermediate geometric computations so that they can be reused as LSD backtracks in its search for a solution. Without access to such intermediate results, the solid modeler kernel would be forced to recompute each object as a result of backtracking.
- A mechanism for the computation of a "sample look" for explicit components when the program contains references to such components that are not fully specified. Note

that in LSD, the user can incorporate explicit components into a program that are parameterised without necessarily specifying the values for the parameters of the component.

Although many off-the-shelf commercial solid modeling packages are available, none fully satisfy the above requirements. Hence, a solid modeling system with the minimum features listed above is one of the foci of this work.

As noted earlier, the formal model in [37] introduces a generalised notion of operation that allows new solids to be created by applying operations to solids. The notion of behavioural solids, defined in Chapter 5, depends on operations, which in turn rely on constraints. This leads to the conjecture that a constraint-based solid modeling package would be a good choice for an LSD implementation.

The above two observations suggest that a solid modeler with a front-end based on a Constraint Logic Programming (CLP) language could be an appropriate choice for interfacing with and providing services to LSD. We explore the feasibility of building such a solid modeler using a CLP extension of Lograph, which we call Lograph+C. Note that the heart of such a solid modeler must consist of a solid modeling kernel capable of geometric computations.

Lograph, as it is defined, is non-deterministic. Hence, for the usual reasons of efficiency, programmability and so forth, to obtain a practical Lograph+C as the basis of the intended solid modeler, we need to devise a deterministic execution model, with support for LSD debugger features which include

- Continuous and single step animated execution of LSD programs.
- Ability to dynamically and freely step forward and backward in execution.
- Just-in-time programming, which is the postponement of the definition of fragments of a program till their first use at run-time (resembling the similar feature of Prograph [98]).

Another desired feature for the LSD debugger is automatic layout of query graphs as execution proceeds. This is essential for preventing the query graph from becoming a clutter of nodes which would require the user to manually rearrange the graph after each execution step. To this end, we will present an automatic layout algorithm for the LSD debugger.

Given the above requirements and desired features for LSD and its debugging environment, this thesis contributes to the theory and design of LSD by extending the formal model for solids in [37] to

- Formally define the role of constraints.
- Introduce the notion of reduction of a solid which is the minimisation of the number of variables over which a solid is defined.
- Introduce the notion of sample look for a solid and its automatic generation.
- Define solid behaviours.
- Formally define motion as a specific class of behaviours for solids with moving parts.
- Support high-level parameterisation and transformation of solids.

Once the above extensions are made to the formal model, the LSD semantics are refined to capture the extended formalism.

This thesis also contributes to certain practical aspects of LSD by

- Devising an execution mechanism for Lograph.
- Designing and prototyping an interpreter engine for Lograph which supports
    - execution rollback
    - single step and animated execution of queries
    - just-in-time programming
- Introducing constraints to Lograph to obtain Lograph+C.
- Designing a debugger environment for Lograph.
- Designing an automatic layout algorithm for Lograph debugger.

1.5 Organisation

In Chapter 2, we give an overview of Lograph, followed by an introduction to LSD, illustrated with an example of designing parameterised keys. We also show how LSD can provide a convenient platform for solving certain classes of design problems which require both symbolic computation and parameterised design specification.

In Chapter 3, we consider issues related to the extension and implementation of Lograph. We discuss the need for the deterministic execution of Lograph and describe how it can be achieved, and explain how Lograph can be extended to obtain a visual constraint logic programming language. This chapter also includes the design of an interpreter engine for Lograph which supports features such as single step execution of programs, forward and reverse execution, and just-in-time programming.

Chapter 4 discusses the need for automatic layout, discusses related cognitive issues, then presents an algorithm for achieving it.

Chapter 5 reviews the formal model for solids presented by Cox and Smedley [37] then introduces extensions in order to satisfy some of the requirements listed above. In particular, we define a decomposition of solids into factorings to facilitate the computation of sample looks for solids, consider the reduction of solids, formally define behaviour, and show how motion can be modeled.

Chapter 6 describes how extensions to the formal model presented in Chapter 5 are captured in LSD. This is followed by an introduction to common solid modeling techniques and an investigation in detail of the use of PLaSM as a solid modeling kernel for LSD.

Chapter 7 summarises our progress to date, and lays a path for future research.

2 Solids in Logic

Complex geometric objects are often composed of smaller components, the relationship between which is a result of design decisions and the overall functionality of the design. Cox and Smedley in [38] noted that in logic programming, terms (data) and literals (the components of algorithms) are uniformly represented, and the execution mechanism transforms terms via unification. This observation led to the conjecture that visual logic programming might provide a basis for a design language in which design components and algorithm components are homogeneously represented in an integrated environment, and to a proposal for a Language for Structured Design (LSD).

The language underlying LSD is Lograph, a general, non-deterministic, visual logic programming language [36], which as noted in [38], has some properties that make it particularly suitable as the basis for a design environment, unlike various other visual logic languages [2,47,70,87,89,96,102,112,121]. First, the semantics can be realised as graph transformations, and second, unification is replaced by two execution rules that reveal the details of unification rather than treat it as one large step [35]. Together, these properties allow an execution to be viewed as a movie depicting the morphing of a query into a result, an important property for a design language, where animating the assembly of objects is a desirable feature. Animation of execution of a logic program is not unique to Lograph, for example Pictorial Janus [64] also provides animated execution of programs.

Lograph is extended to LSD by the addition of *components* and *operations,* logical manifestations of *solids* and *operations* on them in design spaces, formally defined in [37].

Since LSD is a programming language based on logic, it naturally provides problem solving capabilities. In this chapter, after presenting a description of LSD, we demonstrate how these capabilities can be employed to solve design problems, thereby extending the functionality normally associated with CAD systems to include high level design specification. This work was originally reported in [15].

Since Lograph syntax and semantics are essential to our discussion, in the next section we give a brief overview of Lograph. The presentation is based on the more detailed descriptions in [36].

2.1 Lograph Syntax and Semantics

Lograph is a visual representation of *flat Horn clauses* which are a specific form of first-order predicate calculus formulae. The semantics of flat Horn clauses are defined by a set of deduction rules called *Surface Deduction.* The discussion of surface deduction, its soundness and completeness can be found in [35,66].

Lograph is a general purpose visual logic programming language based on first order predicate calculus and in that respect there are many similarities between Lograph and Prolog [67]. We will trade on these similarities, noting parallels between the two languages to informally introduce Lograph.

A *function cell* in Lograph corresponds to a term in Prolog and is depicted as an icon. Figure 2.1 illustrates a function cell named •. The small circles attached to a function cell are called *terminals* which are analogous to variables in Prolog. The terminal at the peak of the curved face of a function cell is called the *root terminal.* The number of terminals at the flat side of a function cell is called the *arity* of the function cell. The function cell can have two possible orientations or . The two possible orientations of a function cell are for layout purposes and, regardless of the orientation of a function cell, its terminals are ordered from left to right. A function cell with arity 0, also called a *constant*, has the simpler representation, <name> or <name>, where <name> is the name of the cell. A constant in Lograph is analogous to a constant term in Prolog.

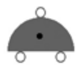


Figure 2.1: A function cell.

In Prolog, data structures can be constructed from terms and constants, similarly, in Lograph data structures can be constructed from function cells and constants. For example, Prolog provides a special notation for lists; namely, the string [item-1, item-2, ..., item-n] represents the term •(item-1,•(item-2,...•(item-n,[])...)). Similarly, in Lograph one can use function cells named • to create lists, and the constant [] to signify the empty list. The root terminal of a • cell represents the list, while its first and second terminals correspond to the head and tail of the list respectively. For example, the graph in Figure 2.2 corresponds to the Prolog term •(a,•(b,•(1,•(2,[])))) and represents the list [a,b,1,2]. The line segments that connect terminals are called *wires.* A wire connecting two terminals in Lograph is analogous to two occurrences of a variable in Prolog. Hence, saying "terminal A is connected to terminal B" is the same as saying "terminals A and B are the same".

In Lograph, as in Prolog, a list can be abbreviated. For example, the list in Figure 2.2 can be denoted [a,b,1,2] .

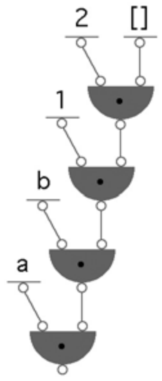


Figure 2.2: A list in Lograph.

A *literal cell* in Lograph is analogous to a literal in the body of a clause or a query in Prolog. A literal cell is depicted by a rounded rectangle bearing the name of the literal. The number of the terminals attached to a literal cell is called the *arity* of the literal cell.

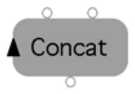


Figure 2.3: A literal cell.

The terminals of a literal cell are represented by small circles ordered clockwise around the cell's perimeter starting from the arrowhead icon. Figure 2.3 shows a literal cell named **Concat** with arity three.

A Lograph *definition* is analogous to a set of Prolog clauses that define a predicate, and consists of a set of cases of the same name and arity, where a case is analogous to a clause. A *case* in Lograph is depicted as a rounded rectangle with a list of terminals attached inside, called the *head terminals* of the case. The head terminals are ordered clockwise starting from the arrowhead icon. A case contains a network of literal and function cells connected by wires. The top left corner of a case bears the name of the definition to which the case belongs in a small rectangle. The number of head terminals is called the *arity* of the case. Figure 2.4 illustrates the three cases of the definition **Concat** to which the **Concat** literal cell of Figure 2.3 corresponds.

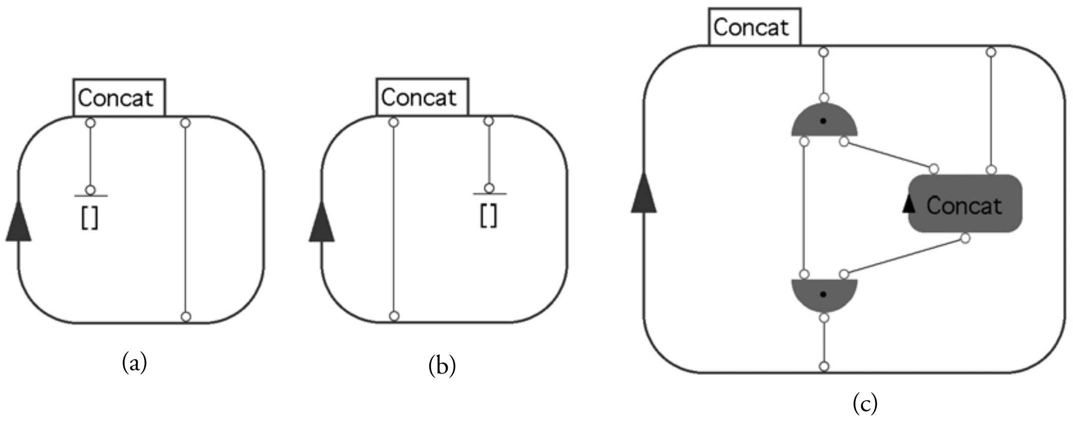


Figure 2.4: Cases of the definition Concat.

Just as the execution of a query in Prolog aims to produce the empty clause, the goal of a Lograph execution is to reduce a query to an empty graph. A *query* in Lograph is a network of cells, none of the terminals of which occur in the program. Figure 2.5(a) is an example of a query. The reduction of a query to an empty graph is achieved through the application of three Lograph execution rules, described below.

Since a query provides the starting point for execution of a program, we use the phrases "execution of a program" and "execution of a query" interchangeably.

The *Replacement* rule replaces a literal cell with a copy of the body of one of the cases of the definition with the same name and arity as the literal cell, if such a definition exists. The terminals of the head of the case and the corresponding terminals of the literal are connected in the process. By *connecting* two terminals, we mean that every occurrence of one of them is replaced by a new occurrence of the other. For example, the query in Figure 2.5(b) is obtained from that in Figure 2.5(a) by replacing the **Concat** literal with the recursive case of the **Concat** definition illustrated in Figure 2.4(c).

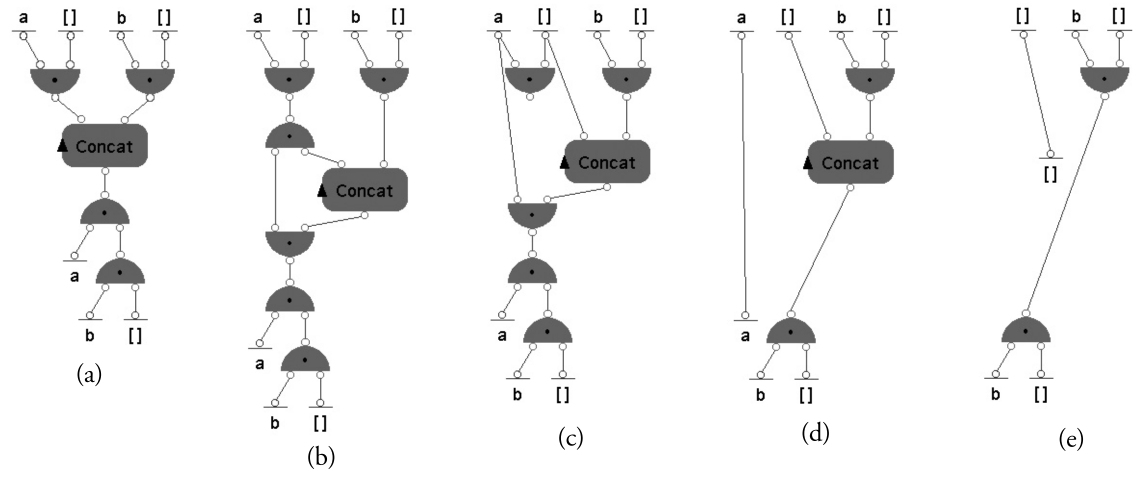


Figure 2.5: Execution steps of a query in Lograph.

The *Merge* rule can be applied to two function cells if they have the same name, arity and root terminal. First the corresponding terminals of the two function cells are connected, then one of the cells is deleted. Figure 2.5(c) shows the query after a merge rule is applied on the query in Figure 2.5(b).

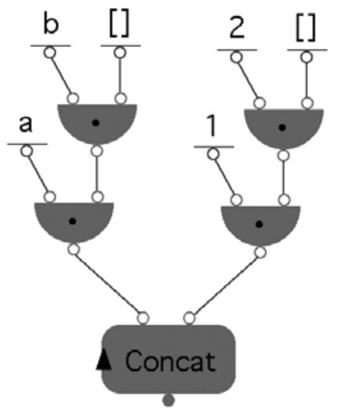
The *Deletion* rule applies to a function cell, the root terminal of which has no other occurrences, removing the cell from the query. One application of delete rule, followed by one application of merge rule and then another deletion will transform the query in Figure 2.5(c) into the query in Figure 2.5(d).

Note that in Figure 2.5(d) an application of the merge rule followed by deletion will remove the two $\frac{\circ}{\mathrm{a}}$ constants from the query. Then replacing the **Concat** literal with the case in Figure 2.4(a) will transform the query into the one in Figure 2.5(e). It is easy to see that applications of the merge and deletion rules in any order will transform the query in Figure 2.5(e) into an empty graph, indicating the successful execution of the query in Figure 2.5(a).

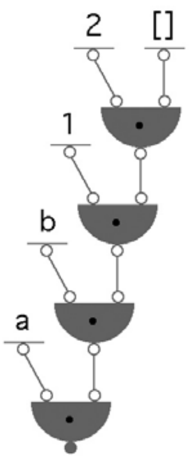
In Prolog, a failure occurs when two variables cannot be unified because they are bound to terms that begin with different functions. The analogy in Lograph occurs when the query is transformed into a graph containing two function cells with their roots connected, and with different names or arities. Such cells cannot be merged, and therefore prevent the query from being transformed into the empty graph.

As we noted earlier, a successful execution in Lograph transforms a query into an empty graph. However, this is not always what we want from execution. For example, in Figure 2.6(a) the third terminal of the **Concat** literal is not attached to anything else, and corresponds to an anonymous variable in Prolog. Given the intended meaning of **Concat**, executing this query will result in this terminal eventually being attached to a structure that represents the concatenation of the two lists [a,b] and [1,2], as shown in Figure 2.6(b). In order to block the transformation of such structures into the empty graph, we introduce *guard terminals*, represented by solid circles, •, rather than empty ones . The role of a guard terminal is to prevent application of the deletion rule. A function cell, the root of which is a guard terminal, is not subject to the deletion rule. Furthermore, the "guarding" property is inherited through replacement and merge rules.

The presence of the guard terminal in Figure 2.6(b) results in termination of execution. Although this graph cannot be transformed into an empty graph, we do not consider this a failure.



(a)



(b)

Figure 2.6: A query with a guard terminal.

2.2 LSD

LSD is an extension of Lograph, a general-purpose visual logic programming language, and is intended for designing structures rather than general purpose programming; hence the names of LSD entities were chosen to be suggestive of their roles in the design world.

LSD provides capabilities both for programming and for designing parameterised components of complex objects in a homogenous environment. A component designed in LSD represents a family of solids, individual members of which can be realised by providing specific values for design parameters. For example, a particular kind of parallelogram may be defined in a 2-dimensional design world by a fixed height and variable base lengths. If the parallelogram is incorporated in a larger design, its bases may become constrained to some specific values when the parallelogram is fused to other shapes. Parameterising designs using the programming features of LSD in a descriptive fashion, and manipulating designs in a solid world, are features of LSD that will be exploited in the following sections.

We will briefly and informally introduce LSD using the example of designing keys, and for a formal and comprehensive discussion, refer the reader to [38].

First we deal with the representation in LSD of solid objects by considering the example of keys. A key solid consists of four types of components, a *handle*, a number of *bits*, a number of *levellers*, and a *tip*. Solids have *open edges*, indicated in Figure 2.7 by arrows, along which they can be bonded to other solids. The solid lies to the right side of the arrow. We will omit these arrows in later diagrams.

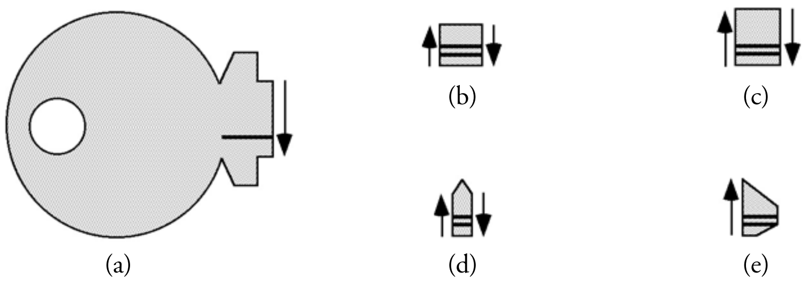


Figure 2.7: Components of a key solid with two possible bit cuttings.

The handle and tip, shown in Figure 2.7(a) and (e), have no logical significance in the functionality of a key. A bit is a solid of a fixed width and height with open edges at its sides. In our example, as illustrated in Figure 2.7(b) and (c), a bit comes in two possible sizes, differing only in height. A leveller is a solid of fixed width as in Figure 2.7(d). The heights of the sides of a leveller can be independently varied. When the side of a leveller is fused to another solid, the height of the leveller at the binding side is constrained to the height of the other solid. Once both sides of a leveller are bonded to other solids, the leveller will create a ramp between the tops of those solids. The solid bonded to the left of a leveller may be a bit or a handle. The solid bonded to the right of a leveller may be a bit or a tip. The height of the tip at its left side is variable and will be set to the height of the solid to which it is bonded.

An LSD *program* is a set of *designs*. A design is a set of cases which define the structure of an artifact. A *case* consists of a name, a head and a body. The *head* of a case is an ordered list of terminals of length n for some integer $n \geq 0$, where n is called the *arity* of the case. The terminals are represented by small circles arranged around the perimeter of the case, starting from the clockwise-pointing arrow called the *origin* of the case. A terminal is either a simple terminal, or an *edge terminal*, represented by ▪▪▪▪▪▪▪ .The *body* of a case is a network of cells interconnected by wires and bonds. A *bond* connects two edges, where an *edge* is either an edge terminal, or an open edge like those indicated by arrows in Figure 2.7. A *cell* is either a function cell or a component.

Figure 2.8 depicts a program consisting of two designs, **Partial-Key** and **Key**. **Partial-Key** expects to receive a list of integers on its head terminal, corresponding to bits, and recursively describes a partial key as the component obtained by bonding a bit corresponding to the head of its input list to a partial key corresponding to the tail of the input list. The handle in the single case of the design **Key**, and the tip in the non-recursive case of **Partial-Key**, are *explicit components* (e-components). The round rectangle named **Partial-Key** is an implicit component representing an invocation of design **Partial-Key**. An *implicit component* (i-component) consists of a name and a list of terminals arranged around the perimeter starting from the *origin,* a clockwise-pointing arrowhead which may be placed anywhere on the perimeter. The grey stripe connecting the open edge of the handle and i-component **Partial-Key** is a bond. A bond connecting two e-components fuses those two components at their corresponding open edges during execution.

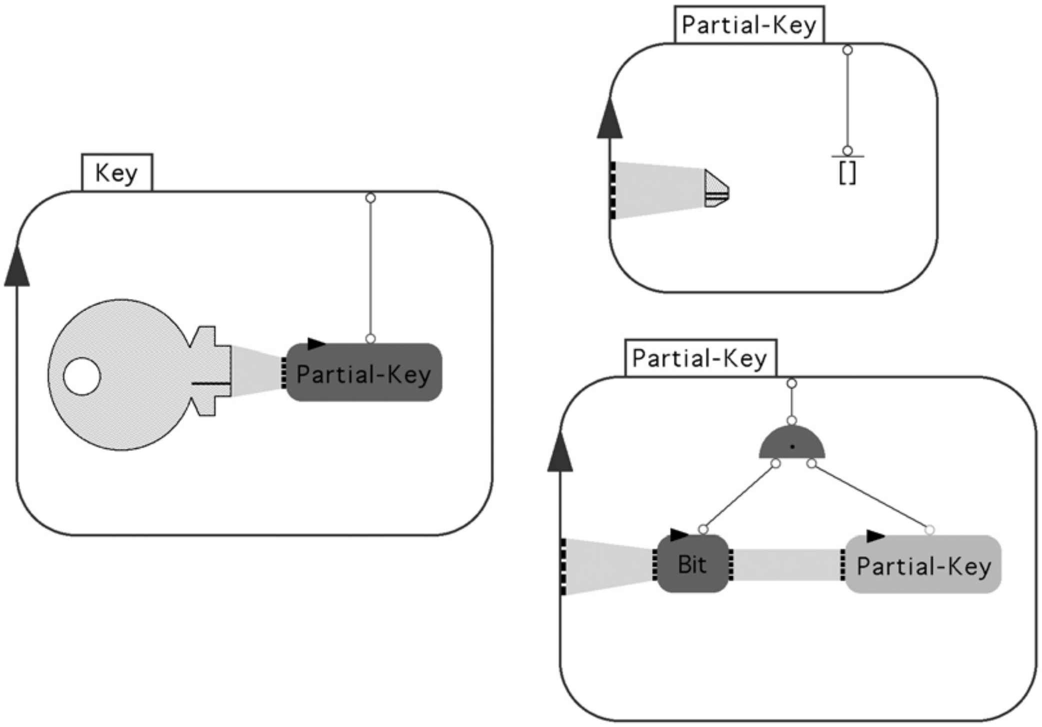


Figure 2.8: The LSD designs Key and Partial-Key.

Each design and i-component has a *signature* consisting of its name together with a list of the types of its terminals. For example, the design **Partial-Key** and the i-components **Partial-Key** in Figure 2.8 all have signature (**Partial-Key**, (simple, edge)).

As the reader may have noticed, the two i-components in the recursive case of design **Partial-Key** are painted in two different shades of the same colour. This is an implementation issue that will be clarified later.

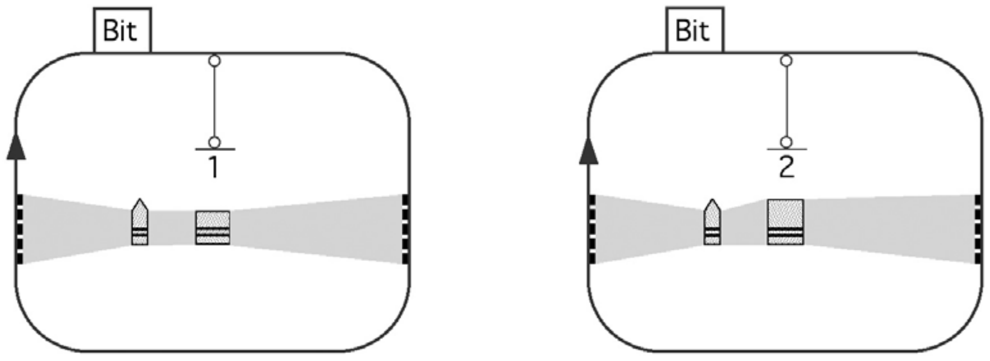
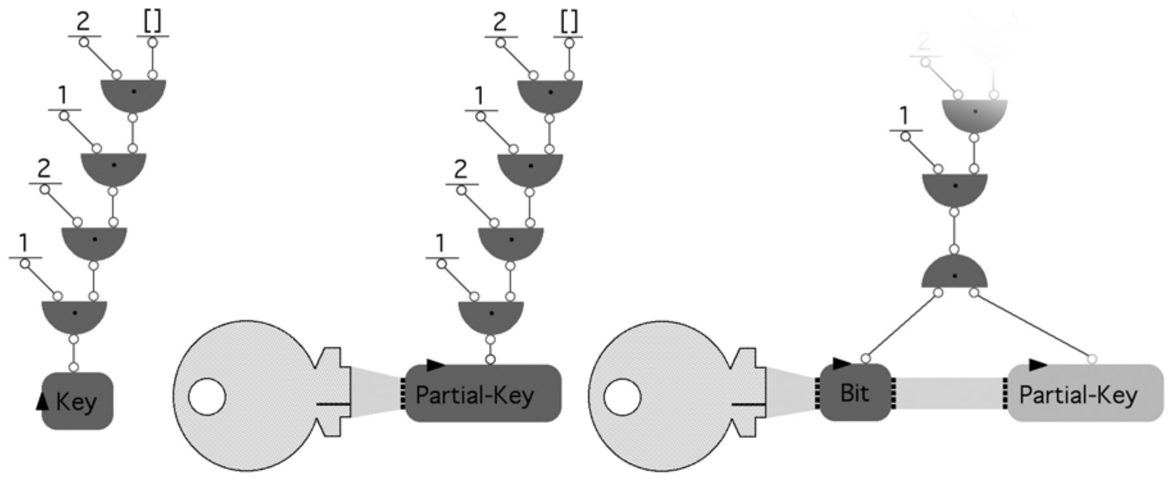


Figure 2.9: Two cases of the design Bit.

The implicit component **Bit** in the recursive case of **Partial-Key** is an invocation of a design **Bit** shown in Figure 2.9, defined by two cases corresponding to the two sizes of bit.

The process of building a component according to the designs in a program is called *assembly*. Assembly transforms a *design specification*, a network of function cells and components, using four execution rules: *replacement*, *merge*, *deletion* and *bonding*. For example, Figure 2.10(a) depicts a design specification for a key with four bits of sizes 1, 2, 1 and 2.The specification in Figure 2.10(b) is obtained by applying the replacement rule to the one in Figure 2.10(a), replacing the i-component **Key** with a copy of the body of the case of design **Key**. During this process, connections (bonds or wires) are established by matching terminals of the replaced i-component with corresponding terminals in the head of the case.

Applying replacement to the i-component **Partial-Key** in Figure 2.10(b) using the recursive case of design **Partial-Key** transforms the specification to the specification in Figure 2.10(c), parts of which have been omitted for compactness. This new specification contains two function cells with the same name and arity, connected by their root terminals. Such cells are subject to the merge rule, which merges them into one, creating new connections by identifying corresponding terminals of the two cells, as shown in Figure 2.11(a).



(a) (b) (c)

Figure 2.10: A key specification, and the replacement rule.

The specification in Figure 2.11(a) is transformed to the one in Figure 2.11(b) by the deletion rule, which removes function cells with unconnected root terminals. Next, the i-component **Bit** in Figure 2.11(b) is replaced with the case of design **Bit** on the left of Figure 2.9, resulting in the specification shown in Figure 2.11(c). One application of merge and one of deletion removes the two constants named 1 which are connected by their roots, producing the specification in Figure 2.12. One of the consequences of the most recent replacement is that the specification now contains bonds which are incident on open edges rather than edge terminals. Such bonds can be executed, joining together the e-components along their open edges. In our example, executing the two executable bonds produces the specification in Figure 2.12(b) containing a new e-component, a partially defined key with one bit and no tip.

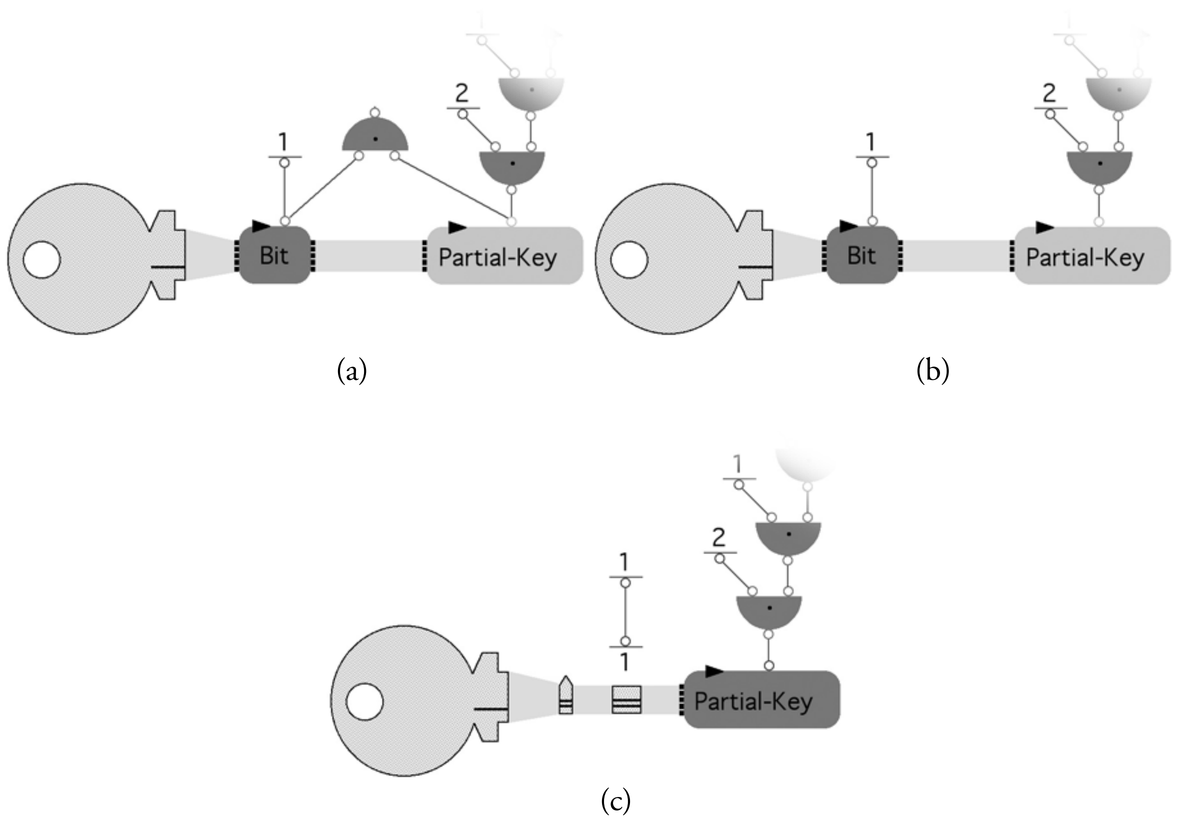


Figure 2.11: The assembly process.

Continuing execution will transform Figure 2.12(b) into the e-component in Figure 2.12(c) by adding more bits until all items in the list in the original specification are consumed. Assembly stops when a specification is produced that cannot be further transformed, as depicted in Figure 2.12(c).

The reader has no doubt noticed that by choosing the "correct" case of **Bit** in the replacement that transformed Figure 2.11(b) into Figure 2.11(c), we guaranteed that the merge and deletion rules would apply to obtain Figure 2.12(a). If we had made the wrong choice, the graph would have included the constants $\frac{\circ}{1}$ and $\frac{\circ}{2}$ connected by their roots. All subsequent graphs in the sequence would also contain this subgraph since none of the execution rules apply to it. Consequently, although the final, untransformable specification would contain a key, it would also contain these unremovable connected constants, and others resulting from "incorrect" choices. Such assemblies are considered to be unsuccessful.

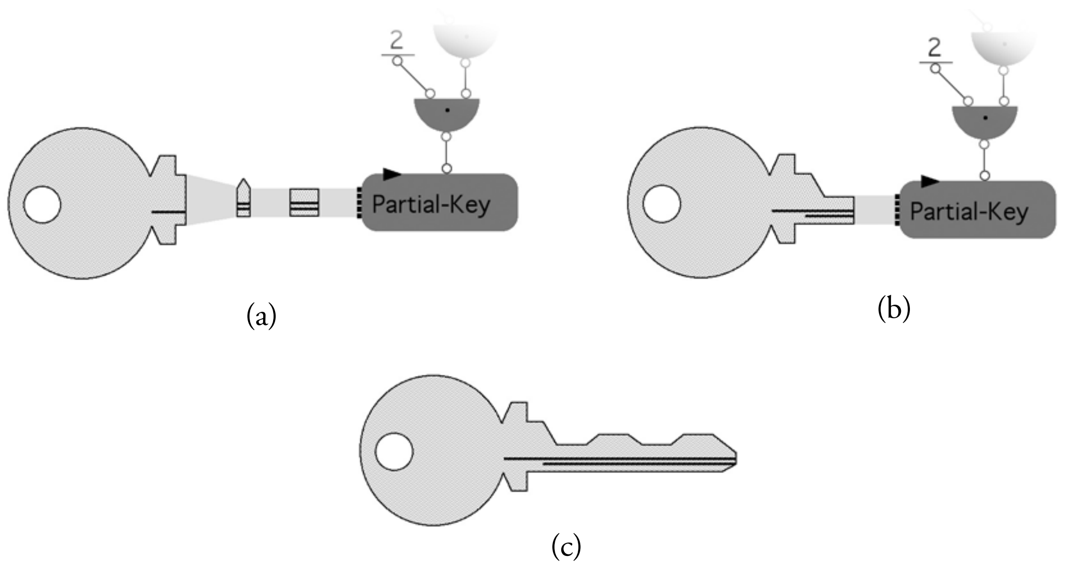


Figure 2.12: Key assembly process.

2.3 Relating Lograph to LSD

As we noted earlier, LSD extends Lograph by including solids and operations on solids. Many of the computations required in the synthesis phase of design may be purely symbolic or arithmetic. Such computations can be performed by the Lograph subset of LSD, as follows. In LSD, a *definition* is a design, no case of which contains an i-component that is not a literal. A *literal* is an i-component with the same signature as a definition. Hence a literal is an i-component that does not introduce any e-components, either directly or recursively. A Lograph program is one that consists of definitions. Note that "successful" execution of a specification that consists only of function cells and literals results in the empty graph.

2.4 Design Synthesis

In this section, we explain how the problem solving capabilities provided by the Lograph subset, together with the design specification aspects of LSD can provide a homogeneous programming/design environment for solving some structured artifact design problems. We will use *masterkeying* as an example to illustrate the process. A comprehensive discussion of the masterkeying problem and the analysis of its complexity can be found in [49] on which we have based our definitions.

2.4.1 Masterkeying

Masterkeying is allowing several different keys to operate one lock. It is used in a systematic way to regulate access to areas according to key holders' privileges. In a master key system, there is always one master key with the highest access privilege that can operate every lock in the system, and there may be sub-masters of various levels that can open subsets of the set of locks. A chain key is one that can operate only one lock. A more precise definition of the masterkeying problem follows.

A *key* is a solid that can open a *lock*. A lock is also a solid consisting of a *body*, and a *plug*. The body has a set of *chambers*. Inside every chamber is a *spring* and a *pin*. Every spring is attached to the end of the chamber at one end and the pin at the other. Every pin is cut through at some level. The end of a pin connected to the spring is called the *driver pin* and the end in the plug end is called the *key pin*. A plug is a rotating cylinder contained inside the body. A plug has a *key way* and a set of *holes* through which the pins can slide. When a key is inserted in the keyway, the pins are pushed towards the body a distance depending on the shape of the key. A key can *open* a lock when the key pushes all the pins in such a way that the cut in each pin is aligned with the shear line between the body and the plug. A wrong key will leave at least one pin across the shear line, preventing the plug from turning. Figure 2.13 illustrates the side view of a lock and one of its keys.

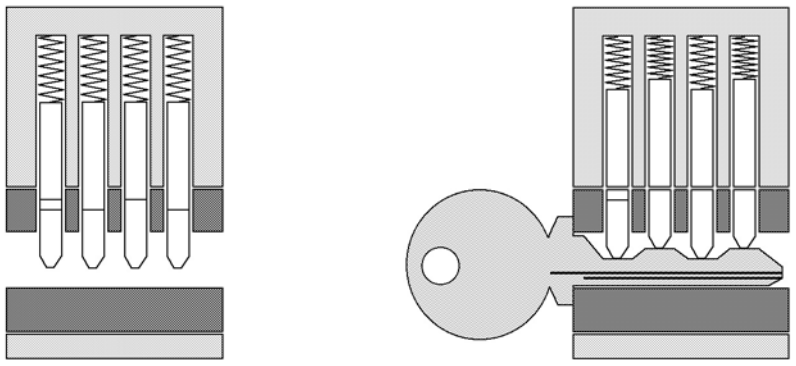


Figure 2.13: A side view of a lock and one of its keys.

In a master key system, there is at least one pin with more than one cut, allowing more than one key to open the lock. The two parts of a pin at the body side and the key side are still called driver and key pin respectively, and the parts in between are called *master pins* or *spacers.*

A locking system is a tuple $\mathcal{L}$=(s_1,s_2,…,s_k) where $k > 0$ is the number of chambers, and for each i ($1 \leq i \leq k$), $s_i \geq 1$ is the maximum number of possible cut levels for the pin in the i^{th} chamber. A *key in* $\mathcal{L}$ is described by a *bitting vector,* which is a k-tuple (b_1,b_2,…,b_k) such that $1 \leq b_i \leq s_i$ for each i ($1 \leq i \leq k$). A *lock in* $\mathcal{L}$ is described by a *bitting array* which is a k-tuple (C_1,C_2,…,C_k) such that C_i is a non-empty subset of {1,2,…,s_i} for each i ($1 \leq i \leq k$). For example, the key and lock in Figure 2.13 are described by the bitting vector (1,2,1,2) and bitting array {{1,2},{2},{1},{2}}.

A key with bitting vector (b_1,b_2,...,b_k) *opens a lock* with bitting array (C_1,C_2,…,C_k) iff $b_i \in C_i$ for all i such that $1 \leq i \leq k$.

Given a list of n keys such that the bitting vector of the j^{th} key in the list is (b_{j1},b_{j2},…,b_{jk}), a lock that is opened by every key in the list is defined by the *induced bitting array* $\mathcal{A}$=(C_1,C_2,…,C_k) where $C_i = \{b_{ji} \mid 1 \leq j \leq n\}$ for all i between 1 and k. For example, an induced bitting array for keys defined by bitting vectors (1,2,2,1), (2,2,2,1), and (1,1,2,1) is {{1,2},{1,2},{2},{1}}.

A *key-lock matrix* $\mathcal{X} = (x_{ij})$ is an $n \times m$ matrix where n and m are the numbers of keys and locks in the system, respectively, $x_{ij} = 1$ if key i can open lock j and $x_{ij} = 0$ otherwise. Table 2-1 shows a key-lock matrix for a system with three keys and two locks.

Table 2-1: A key-lock matrix

	Lock 1	Lock 2
Master key	1	1
Chain key 1	1	0
Chain key 2	0	1

An *implementation* of a key-lock matrix $\mathcal{X}$ is a list $(\mathcal{B}_1,\ldots,\mathcal{B}_n)$ of n bitting vectors describing n keys, and a list of $(\mathcal{A}_1,\ldots,\mathcal{A}_m)$ of m bitting arrays describing m locks such that key i can open lock j if and only if x_{ij} of $\mathcal{X}$ is 1. For example, the list of bitting vectors (1,2,1,2), (2,2,1,2), and (1,2,2,2) together with the list of bitting arrays {{1,2},{2},{1},{2}} and {{1},{2},{1,2},{2}} is a possible implementation of the key-lock matrix of Table 2-1.

The *masterkeying problem* is as follows. Given a key-lock matrix $\mathcal{X}$ and a locking system $\mathcal{L}$, find an implementation of $\mathcal{X}$ such that the bitting vectors and arrays of the implementation describe keys and locks in $\mathcal{L}$.

In [49], Espelage and Wanke show that finding an optimal solution for the masterkeying problem is NP-complete. There are, however, a number of heuristic methods for solving this problem that sacrifice optimality in favor of practicality. One of these heuristic methods assumes a bitting vector for the top master key then derives the bitting vectors of the lower level keys. We adopt this heuristic in an LSD program to find an implementation of the key-lock matrix of Table 2-1. We also assume a maximum number of cut levels for a pin.

2.4.2 Keys and Locks in LSD

In Section 2.2 we showed how a key solid can be expressed in LSD terms. We use a similar approach to define a lock, as illustrated in Figure 2.14. **Lock** is a design with a single case that defines a lock in terms of a list specifying the bitting array. The e-component occurring in this design is the front shell of the lock. The recursive case of the design **Partial-Lock** detaches the head of the input list and uses it to add an appropriate chamber to the assembled partial lock, then recursively generates the remaining portion of the lock according to the remaining items in the tail of the input list. The e-component in the non-recursive case of **Partial-Lock** is the back end shell of the lock.

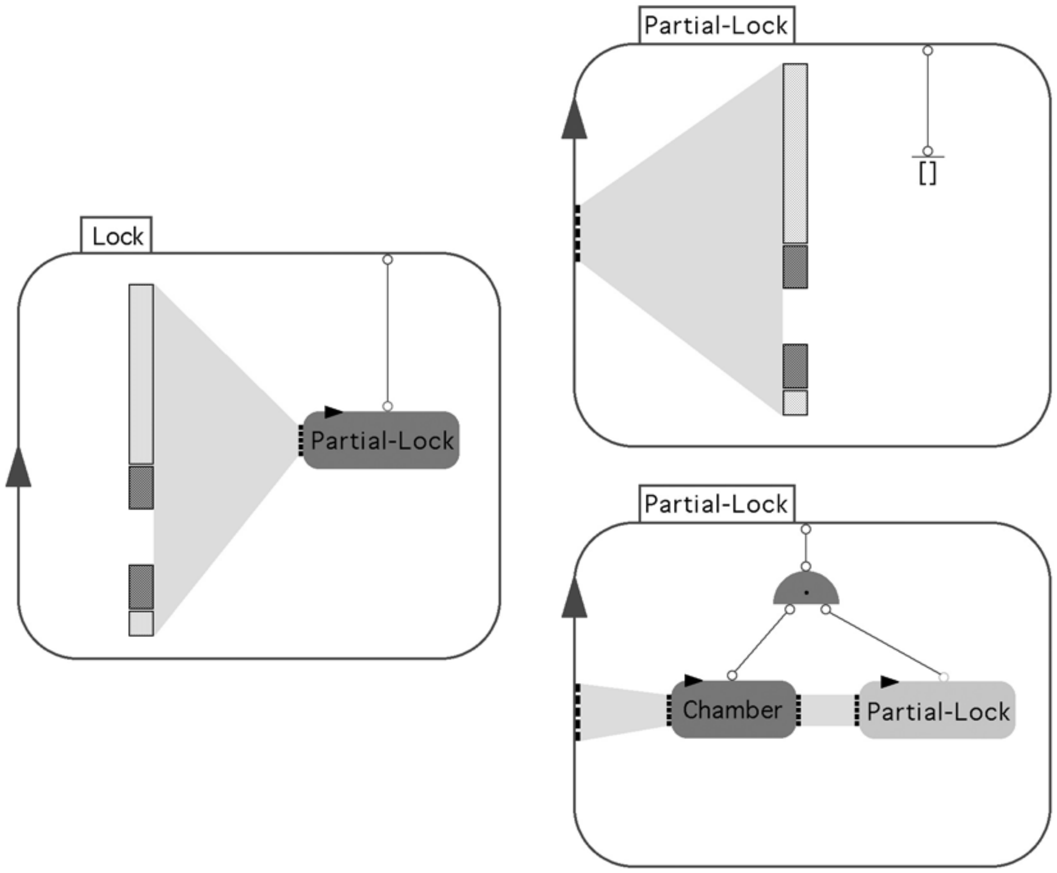


Figure 2.14: Lock and **Partial-Lock** cases.

The implicit component **Chamber** in Figure 2.14 corresponds to the design depicted in Figure 2.15. Each of the three cases labelled (a), (b), and (c) corresponds to one out of three possible pin cut configurations. Case (d) in Figure 2.15 corresponds to a null chamber.

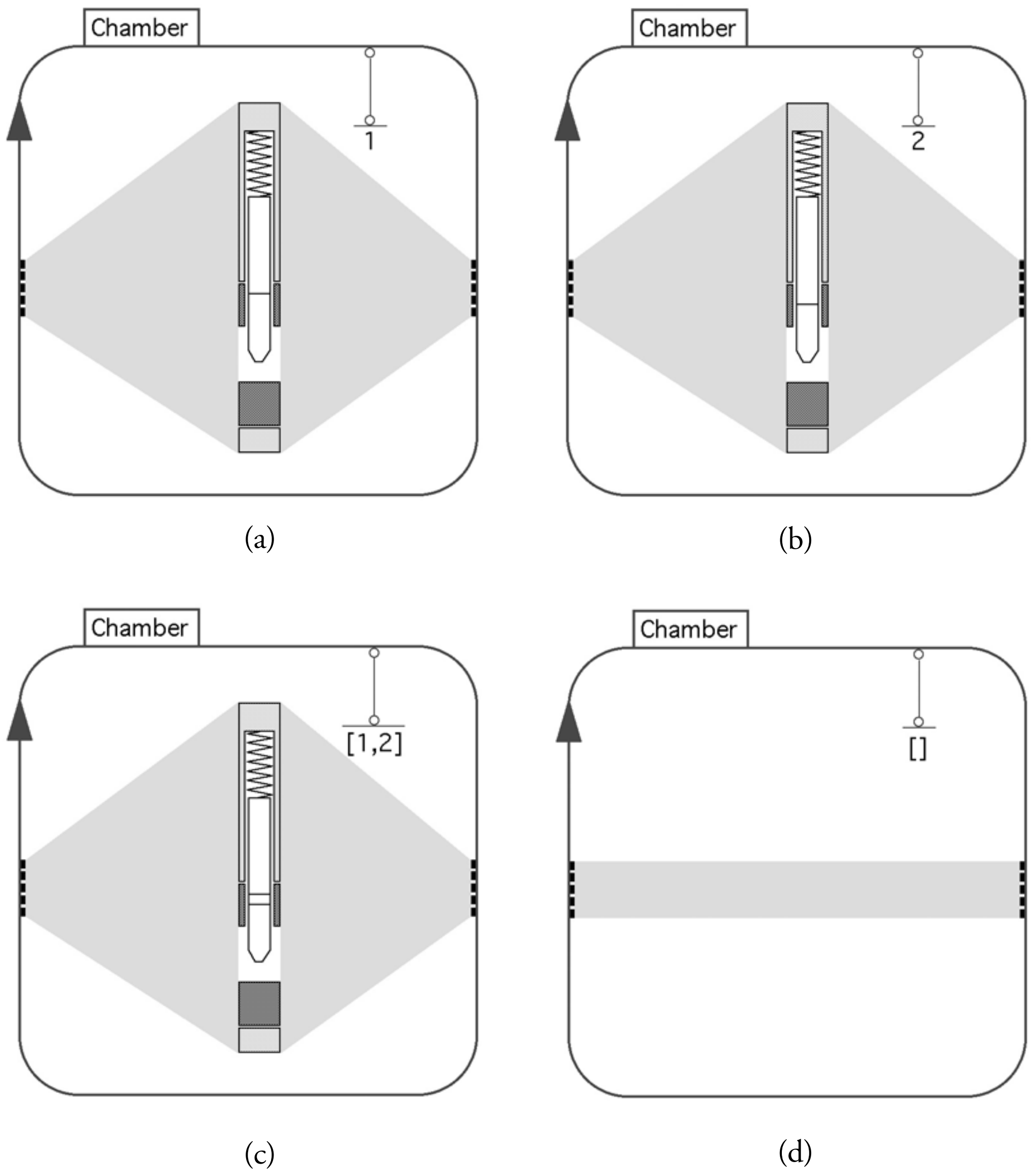


Figure 2.15: Chamber design

In order to express a masterkeying problem in LSD, we define two additional literals: **Open** and **Member**, the former checks whether a key can open a lock and the latter defines the relation "member of list". Their definitions are shown in Figure 2.16. Note that in LSD, an unconnected terminal is analogous to an anonymous variable in Prolog.

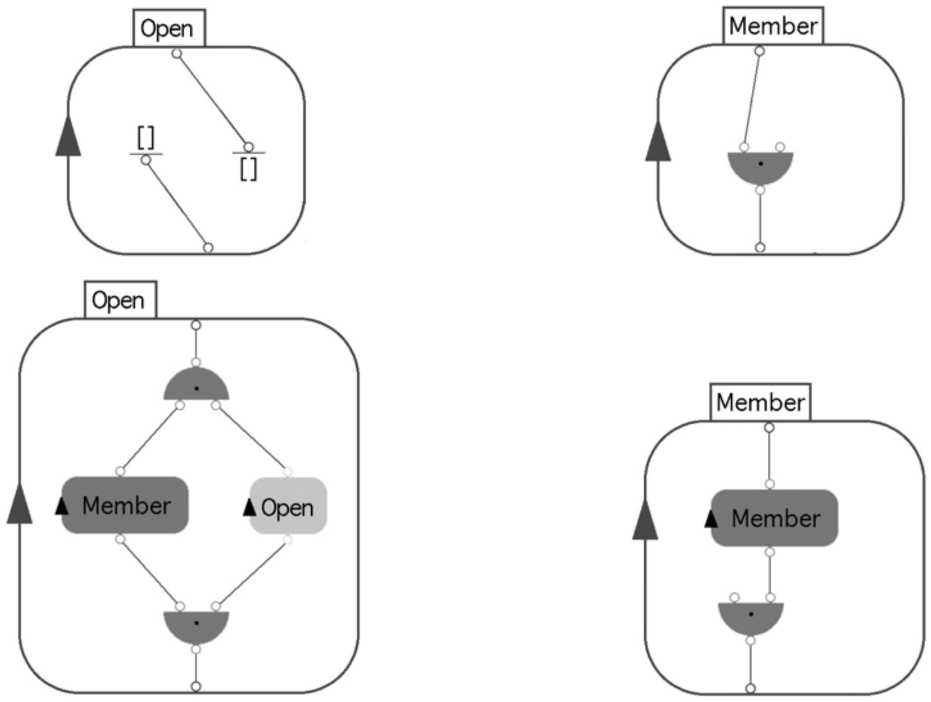


Figure 2.16: Open and Member definitions.

2.4.3 Key-Lock Graph

Now that we have the necessary definitions, we can construct the specification in Figure 2.17 which expresses a solution to a masterkeying problem in which there are three keys (a master key and two chain keys) and two locks. In this specification, the **Open** literals correspond to the 1's in the key-lock matrix of Table 2-1, and the *crossed* **Open** literals correspond to the 0's. Note that crossed lines on a literal denote negation.

The graph in Figure 2.17 is, therefore, a visualisation of the key-lock matrix of the masterkeying problem to be solved, assuming a bitting vector [1,2,1,2] for the master key. Execution will produce bitting vectors for the two chain keys and induced bitting arrays for the two locks, which serve as parameters to the corresponding **Key** and **Lock** i-components, from which corresponding e-components are assembled. Figure 2.18 shows the final result after execution of the specification is complete.

As we mentioned earlier, the different shades of i-components in a design or specification is an implementation consideration rather than a language one. The different shades in which i-components are painted indicate the order in which the components will be executed using the replacement rule, where darker components are executed first. When a replacement rule introduces a new set of i-components, the new components are painted in shades darker than existing components, giving them higher priority. After every replacement rule, all i-components in a specification are recoloured to comply with the ordering mechanism.

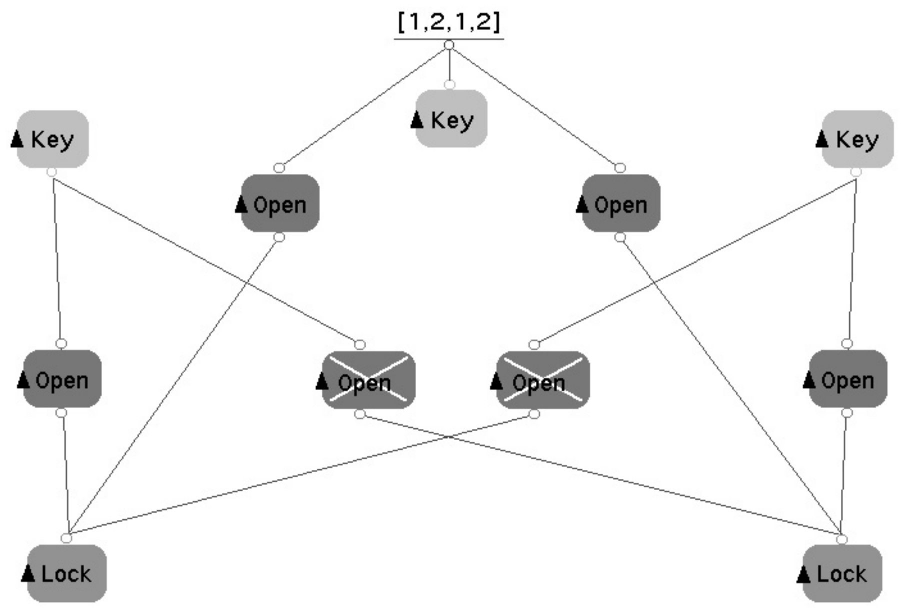


Figure 2.17: Masterkeying problem.

Note that all **Open** literals in the specification in Figure 2.17 should be executed before any **Key** or **Lock** i-component since they compute the parameters for assembly. This divides the execution of the specification into two parts, solving the design problem followed by constructing the keys and locks according to the computed parameters. From the designer's point of view, however, the diagram in Figure 2.17 simply combines problem-solving and construction in a single declarative specification. This homogenous expression of problem description and construction procedure relieves the designer from having to consider issues of execution order. Note, however, that a good choice of execution order may well result in the assembly process being faster and more efficient, as is the case in other logic programming languages. Given the standard order of the phases of the design cycle discussed earlier, it should be quite natural for a designer creating such specifications to impose the correct ordering on the components of a design. In Chapter 3, we will discuss the ordering of i-components in more detail.

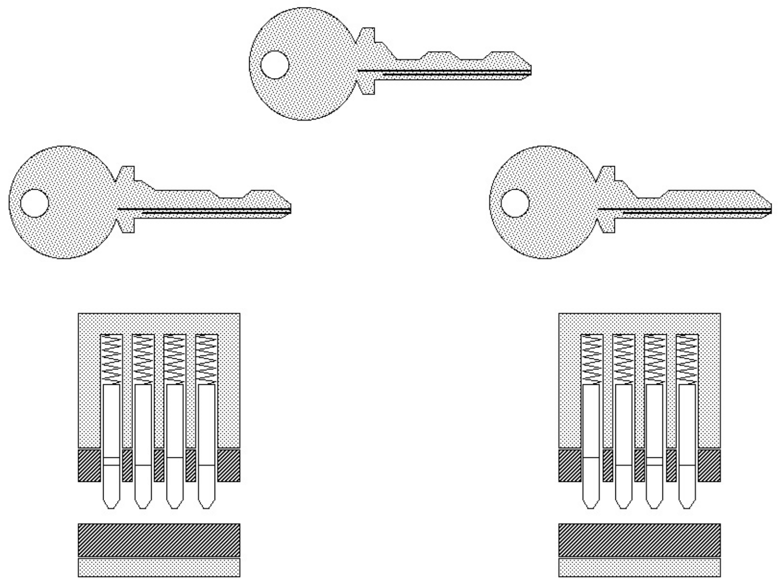


Figure 2.18: Query result.

2.4.4 Notes

We have demonstrated how LSD, a language initially proposed to unify the design and programming aspects of complex design specifications, can also provide a convenient platform for solving design problems. The visual nature of LSD, combined with the strengths of logic programming that it inherits from Lograph, provide a unique combination for solving such problems. We chose the masterkeying problem to present this combination since it provides a simple example of the class of design problems which require both symbolic problem solving and parameterised design specification.

3 Deterministic Lograph+C

In Chapter 1, we discussed a list of features for LSD. The root of each listed feature could be traced to either the LSD language itself or a proposed implementation including a debugger. The desired requirements listed for the debugger included a single step animated execution of LSD programs in both forward and reverse directions, and just-in-time programming as well as automatic layout of queries. A necessary feature is a deterministic execution mechanism.

In an earlier work reported in [14], we discussed the correspondence between Lograph and Prolog programs and the basics of an interpreter engine designed in Prolog for executing Lograph programs. This chapter expands on that work, describing in more detail how Lograph can become a viable visual constraint logic programming language on which LSD can be built.

3.1 Lograph Execution Model

Lograph is a non-deterministic language and so, therefore, is LSD as proposed in [38]. However, programs written in LSD should produce consistent results (solids) when executed. Note that although consistency of results could be achieved by confluence, support for meaningful execution rollback in the LSD debugger requires deterministic execution. We therefore need a deterministic execution model for Lograph, which is one of the issues addressed in this chapter.

The consideration of determinism raises a second issue. Since determinism in logic program execution implies ordering of the literals in the bodies of clauses, and ordering of clauses in the definition of a predicate, as in Prolog, how should such ordering be expressed in a visual logic programming language without resorting to a confusing network of lines?

In Lograph as described in the previous chapters, there are three sources of non-determinism: the choice of which execution rule to apply, which cell or collection of cells to apply it to, and for the replacement rule, which case of a definition to use. Because Lograph represents flat Horn clauses graphically, it expresses this non-determinism in a natural way: however, to make Lograph viable as a programming language, we must impose restrictions similar to those imposed on general, first-order, Horn clause resolution theorem-proving to obtain Prolog.

In Prolog, the clauses that define a predicate are linearly ordered, indicating the order in which they will be applied to a query literal. Similarly, the literals in each program clause are linearly ordered, and are executed by resolution in that order. The order in which the search space is traversed is therefore well defined, and is exploited by the Prolog programmer.

Clearly, since Lograph is a first-order Horn clause language like Prolog, we can aim for the same kind of implementation based on depth first search with backtracking instigated by failure, where failure in Lograph is defined by the occurrence of undeletable function cells, as discussed in Section 2.1. The restrictions we will make are analogous to the above restrictions inherent in Prolog: however, there are some important differences because Lograph is based on surface deduction rather than simple resolution. As in Prolog, we need to impose two orderings on Lograph to obtain a well defined traversal of the search space: specifically, the order in which cases of a definition should be tried in applying the replacement rule, and the order in which literal cells in a query should be replaced. In addition, we need to decide on the order in which the three execution rules are to be applied. We address the latter issue first.

3.1.1 Order of Transformation Rules

In this section we show that if a particular ordering of the transformation rules leads to an execution that reduces a query to an empty graph, then any ordering will do the same.

As mentioned in Section 2.1, the deletion rule is applied to function cells the root terminals of which have no other occurrences. Since a *deletable* function cell cannot participate in any other transformation rules, the order in which deletable functions are removed will have no effect on the rest of the execution.

Let us suppose that the replacement rule precedes the merge rule in the chosen rule ordering, and that the current query contains a pair {A, B} of function cells connected by their roots. Clearly A and B are *compatible* meaning that they have the same name, arity and root terminal, since otherwise the query cannot be reduced to the empty graph.

We have the following cases to consider: either

(a) the query contains some literal cells, or

(a') the query does not contain any literal cells, in which case, either

- (b) the roots of A and B are not connected to any other terminals, or
- (c) the roots of A and B are connected to the roots of some other function cells, or
- (d) the roots of A and B are connected only to non-root terminals of some other function cells.

In case (b), the only transformations that can be applied to A and B are a merge followed by a deletion. These transformations are independent of any others, and can therefore be applied immediately.

In case (c), since the query is eventually reduced to the empty graph, any function cell connected by its root to the roots of A and B must be compatible with them. Since every merge produces a function cell compatible with the merged cells, and therefore with any other compatible cells attached to them by root terminals, the order in which the cells in such a group are merged is irrelevant. Hence the merge of A and B can be performed before any other merges of cells in the group.

In case (d), suppose that the execution removes the “other” function cells before any other transformations occur. The “other” cells are removed either by (d1) deletion, or by (d2) merging followed by deletion.

In case (d1), we are left with an instance of case (b) or (c), so that merging A and B can be the next operation performed. Clearly the deletion of the “other” function cells does not depend on the presence of cells A or B, so the merging of A and B could be performed earlier.

In case (d2), we are left with an instance of case (b), (c) or (d). We deal with (b) and (c) as in the previous paragraph. As for (d), we need only note that cases (d) and (d2) cannot alternate forever since the transformations that occur in case (d2) strictly reduce the size of the graph, so we must eventually get case (b) or (c).

In case (a), no merging of A and B will occur until all replacements have been performed. Clearly, performing the replacements is not affected by the presence or absence of A or B, so we can merge A and B at any time.

Since, as we have seen, the transformation rules can be applied in any order, we need to consider the best order in which to apply them. Obviously, if we are doing a depth first search of the solution space as in Prolog, then we should discover “nonunifiability” early. In Lograph, this means applying the merge rule as early as possible. The deletion rule does not really affect this since it just plays the role of “garbage collector”; however, if we are interested in useful animated visualisations of executions, then we might want to apply it early as well, in order to reduce clutter.

Finally, since Lograph replaces unification with explicit transformation rules, there may be ways to apply them which are better suited to the application. For example it might be possible in some circumstances to “batch” merges and deletions, applying them only occasionally between sequences of consecutive replacements.

3.1.2 Ordering Cases

Just as Prolog orders the clauses of a predicate definition, we need to order the cases of a definition in Lograph. The easiest way to do this is simply to label each case with a sequence number. The ordering can also be visually represented as a list of icons each bearing the name of one of the cases. The case at the head of the list has sequence number 1 and will be tried first in the replacement of a literal cell in a query. The list of cases can be rearranged by dragging and dropping entries in the list.

3.1.3 Ordering Cells in Cases and Queries

As mentioned above, we need to specify the execution order of the cells in a case or query. Note that since merge and deletion can be applied at any time, we need concern ourselves only with the order of literal cells.

This leads us to an interesting problem. Like other visual languages, Lograph exposes the structure of algorithms without imposing needless sequentiality on them. However, we need to impose sequentiality for the sake of efficiency. This looks like the same problem that arises in implementing other visual languages, for example dataflow languages. In the case of dataflow languages, operations are partially ordered, and any linear order produced by topological sort will do [34]. In Lograph, however, the wires are not data flow links, so no suitable order can be automatically generated. It is therefore up to the programmer to specify an appropriate order. This is, of course, what Prolog programmers do, so deciding on literal cell ordering should not be too great a burden for the Lograph programmer. Furthermore, the ultimate goal of this project is to implement LSD where the domain is the design of structured objects. As noted in [15] the design process has a natural order to it, so imposing an ordering on the cells in a case should be quite natural for the designer/programmer.

One obvious solution is to add special connections between literal cells to indicate execution order, like the synchros of Prograph [34]. However these would be far more intrusive than such synchronisation links in a data flow language where they are needed only occasionally. In Section 3.3.2, we will discuss the ordering of cases in the Lograph editor in more detail.

3.2 Lograph+C

As we will see in Chapter 5, constraints are pivotal for defining operations in our solid modeler, hence in order to use Lograph as a basis for solid modeling, support for constraints in Lograph is essential.

Constraint Programming (CP) shares many concepts with logic programming. Jaffar and Lassez in [63] show that logic programming is just a particular type of CP.

Although Lograph is a reasonable candidate as the basis for a CP system because of its declarative nature and its roots in logic, like any other logic programming language, pure Lograph would be extremely inefficient at that task, requiring all the axioms for all the constraint domains on which it would operate. Instead, we will extend Lograph to obtain a constraint logic programming language by establishing mechanisms for specifying different classes of constraints in Lograph, then employing appropriate constraint solvers to satisfy those constraints.

Before getting into extending Lograph to support constraint programming, we will define constraints formally.

3.2.1 Constraint Programming

Constraint Programming (CP) deals with the specification, solution, and satisfaction of constraints. Constraint satisfaction is the process of finding solutions that satisfy a constraint. A *constraint satisfaction problem* (CSP) [129] is a tuple (**X**,**D**,**C**) where

- **X** is a finite set of variables.
- **D** is a set of finite domains, one for each variable.
- **C** is a set of formulae defining relationships between the variables in **X**.

A *punctual solution* to a CSP is an assignment of values from the domains in **D** to the corresponding variables in **X** such that all formulae in **C** are simultaneously satisfied. The *solution* of a CSP is the set of all punctual solutions.

In general, solving a constraint satisfaction problem could be accomplished by theorem proving as we have noted above. In practice, however, constraint satisfaction problems usually have quite specific characteristics such as domain restrictions, or very specific kinds of formulae, for problems involving arithmetic constraints. *Constraint solvers* are special algorithms or programs that find solutions to constraint satisfaction problems by exploiting such characteristics, thereby providing more efficient solution mechanisms. The Delta Blue algorithm [50] is an example of a constraint solver. Another example is provided by constraint logic programming systems which consist of a normal logic programming engine plus domain-specific constraint solvers, for example SICStus Prolog [126], Prolog III [29], and CHIP [43].

3.2.2 Constraint Types

Constraints can be used to formulate many problems, and different problems call for different types of constraints. For example an arithmetic constraint is a binary relation a**R**b, where $\mathbf{R} \in \{ <, >, \leq, \geq, =, \neq \}$ and a and b are arithmetic expressions composed of constants, variables and the usual arithmetic operators.

Based on the combination of these two sets (set of relations and set of operators) constraints can be divided into the following major categories

- Linear constraints
- Non-linear constraints
- Equality constraints
- Inequality constraints

An arithmetic constraint is called a *linear constraint* if both the arithmetic expressions in the constraint relation are linear arithmetic expressions. An arithmetic expression is *linear* if there is at most one variable in every multiplication and no division denominator is a variable. An arithmetic constraint is *non-linear* if at least one of the arithmetic expressions occurring in it is not linear.

Non-linear constraints are typically more complex to analyse and solve than linear constraints. Analytical solutions to non-linear constraints may not be easy to obtain; however, there are numerical methods that can be employed to compute approximate solutions.

3.2.3 Constraint Satisfaction Techniques

As we mentioned earlier, solving a constraint satisfaction problem could be accomplished by theorem proving through a systematic search. Consistency techniques, and stochastic methods are two other general approaches for constraint solving.

Consistency techniques [68,76,129] are based on removing inconsistent values from the domain of each variable. Consistency techniques mostly apply to binary constraints, where each constraint defines a relation between two variables. In consistency techniques, a set of constraints is often represented by a graph, where each node corresponds to a variable and each arc represents a binary relation between two variables. There are several levels of consistency such as arc consistency [131] and path consistency [59,79,80]. Higher degrees of consistency have been studied by Freuder [52] and Cooper [30].

Stochastic methods refer to a group of constraint satisfaction techniques that are based on random algorithms and heuristic methods. Algorithms in this group start with a random assignment of values to the variables and iteratively modify them to achieve consistency. Hill-climbing [129] and Tabu search [55] are two examples of algorithms in this category.

Solution techniques using rewrite rules simplify the set of constraints through the application of certain rules. The solution to a problem in this case is often represented as a set of simplified constraints, derived from the original ones. In numerical approaches, approximate solutions are found through some numerical methods such as fixed point analysis or Newton's method. Numerical approximation techniques have their own disadvantages such as stability and convergence problems. Another group of techniques is based on interval analysis methods initially investigated by Moore [81,82] and later suggested to be integrated in Prolog by Cleary [27] which inspired Older and Vellino to extend the main idea and implement it in BNR Prolog [90,91]. Using interval analysis, instead of finding punctual solutions, a solver finds consistent subregions of the constraint space . This process, referred to as *conservative narrowing*, is done so that no solution is lost [17,18].

3.2.4 Constraint Logic Programming

In logic programming, data is coded as uninterpreted structures called terms [53]. In many domains, however, the user needs more expressive and flexible primitive data objects and support for algebraic computations on them.

One of the aims of Constraint Logic Programming (CLP) is to include support for computation in specific domains. This is usually achieved by introducing special predicate and function symbols to the logic programming schema that have fixed interpretations over their specific domain[53]. In conjunction with every constraint domain in CLP systems, there must be a constraint solver, which uses techniques appropriate to the domain and independent from the resolution and unification processes of logic programming.

In CLP, the intermediate states of a computation that reduces a query to a result are called *computation states.* A computation state contains two components; a set of goals, and a set of constraint formulae called a *constraint store.* A computation state is successful when no goal in the goal set causes a failure and the constraint store is *consistent.* A computation state fails when the constraint store is inconsistent or at least one of the goals fails.

3.2.5 Constraints in Lograph+C

To introduce constraints to Lograph, we follow the example of constraint logic programming languages, adding elements that express constraints which are not processed by the normal logic engine. In the following we assume the existence of a set Σ of constraint solvers, and a set of domains. Every constraint solver in Σ is associated with a grammar which is used to determine the *acceptability* of a formula by the constraint solver. A formula is accepted by a constraint solver if there is a parse tree that yields the formula using the constraint solver's grammar.

A terminal is said to be *typed* iff it is associated with a particular domain, called the *type* of the terminal. A function or literal cell is said to be *typed* iff each of its terminals is typed. The *type* of a typed literal cell is the list of types of its terminals. The *type* of a typed function cell is the list consisting of the type of its root followed by the types of its other terminals.

A case is said to be *typed* iff all terminals in its head and body are typed. The *type* of a typed case is the list of types of its head terminals.

A *relation cell* in Lograph+C, as we will refer to our Lograph with added constraints, is a typed literal cell with no corresponding definition. For example, Figure 3.1 depicts a relation cell = of type (**R**,**R**). The types of the terminals are not shown on the diagram, but in an implementation could be revealed via a popup. The = cell in this example represents the arithmetic relation =. Since this relation is symmetrical, the arrow on the perimeter, defining the start of the terminals list, can be omitted.

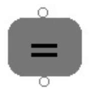


Figure 3.1: A relation cell.

A *constraint cell* is a typed literal cell corresponding to a constraint specification of the same name and type (see below). A constraint cell is distinguished from other literal cells by an icon in the background providing a visual indication of the solver to which the cell is acceptable (see below). Figure 3.2(a) shows a constraint cell named **Half-Plane** with arity 2 acceptable to a linear arithmetic constraint solver.

A *constraint specification* is a definition consisting of a single typed case, where the body of the case consists only of typed function and relation cells and the set of formula specified by the body is acceptable to the constraint solver associated with the corresponding literal cell. The type of a constraint specification is the type of its case.

Since a constraint specification has a single case, we can without ambiguity use phrases such as "a cell occurring in a constraint specification", or "the body of a constraint specification".

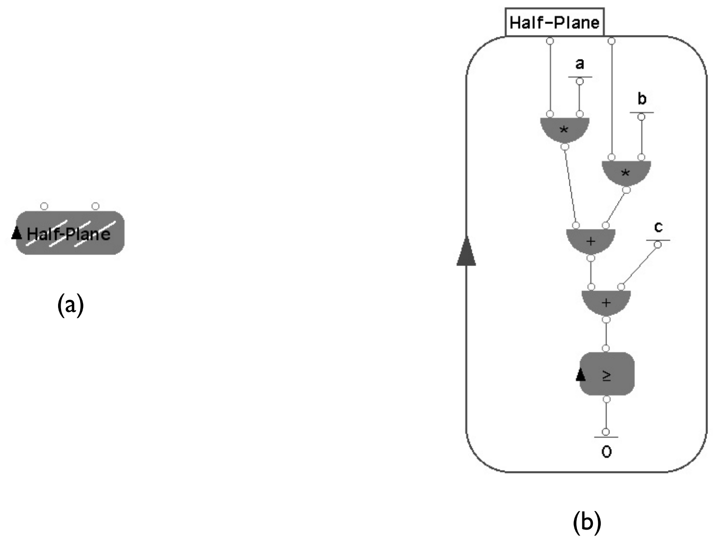


Figure 3.2: A constraint cell (a) and its specification (b)

For example, Figure 3.2(a) illustrates a constraint named **Half-Plane**, the specification of which is depicted in Figure 3.2(b). **Half-Plane** has type (**R**, **R**) and is acceptable to a *linear-solver* because, the function cells are from the set {+,–,÷,*} and are each of type (**R**) or (**R**,**R**,**R**), and the relation cells are all from the set {< , > , ≤ , ≥ , = , ≠ } and are each of type (**R**,**R**). **Half-Plane** defines a linear constraint $ax+by+c \geq 0$ when x and y variables are connected to the first and second terminals of **Half-Plane**, respectively. Note that the three parallel white line segments in the background of **Half-Plane** cell is the visual clue to what algorithm will be used to solve the constraint which is specified by reference *linear-solver.* Note that since a constraint cell is not subject to the replacement rule, the body of a constraint cell will never appear in a query and will never get executed by the inference engine. The body of a constraint cell is in fact the specification of a set of formulae that will be satisfied by an algorithm to which the specification is acceptable.

A query in Lograph is a network of literal and function cells. In Lograph+C, a query remains the same, however, some of the literals in the graph are constraint literals which are ordered according to the usual ordering mechanism of Lograph.

3.3 Lograph Programming Environment

The pictorial nature of Lograph provides an opportunity for a programming environment in which useful visualisations of the structure and execution of programs can greatly enhance the programming and debugging process. In this section, we describe the Lograph programming environment (partly existing and partly proposed), in terms of various cognitive issues.

Storey *et al.* [123] prescribe a hierarchy of cognitive issues that should be taken into consideration when designing *software exploration tools*. Although the cognitive aspects they discuss relate to graphical representations of static textual code, some of the issues they identify are relevant to the design of a visual logic programming editor and debugger.

We divide the cognitive issues concerning the design of a user interface for Lograph into two groups. The first concerns the creation and exploration of static Lograph code, including for example, issues related to increasing the role expressiveness of the editor environment, reducing viscosity, uncloaking data structure patterns, plans, and schemas in Lograph source code, and support for secondary notations such as labelling, commenting and the use of colour for focusing the user's attention.

Issues in the second group deal with the cognitive aspects of animated execution, for example, drawing the user's attention to a specific part of execution, preventing disorientation, preserving the user's mental map, and facilitating visual search.

3.3.1 Lograph Editor

Figure 3.3 illustrates two cases of a definition named member. Each case window has a toolbar at the top, a workspace to the left, containing a rounded rectangle, and a layer list to the right. The workspace and layer list provide two representations of literal-cell ordering, analogous to the two representations of multi-layered images in Photoshop [1]. The layer list is a list of icons similar to the list of layers displayed in Photoshop's "layers" palette. Each list corresponds to a layer containing one or more literal icons. The workspace displays the cells of the case as a series of layers, like the layers in a Photoshop image window, each containing some of the items that make up the whole image. In Lograph, each layer contains one or more literal cells. Cells in the top layer are to be executed first, followed by those in the next layer, and so forth. There is no ordering imposed on literals within a layer, so the programmer can group together literals which are independent and could be executed in parallel.

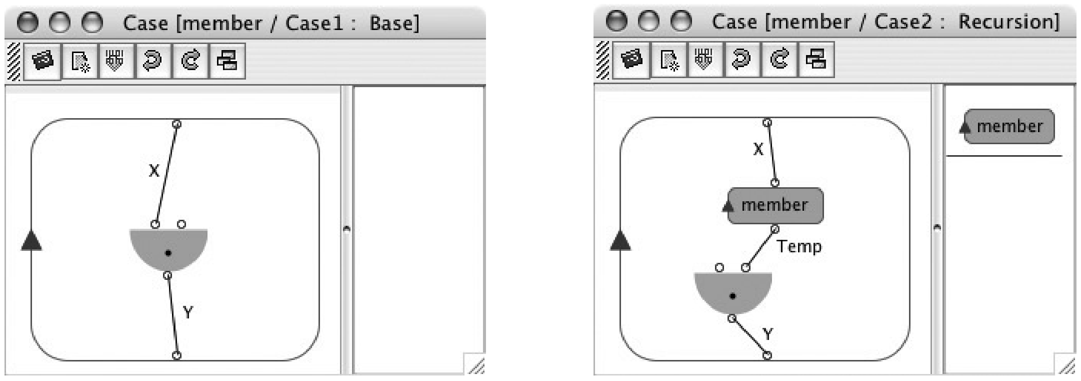

Case [member / Case1 : Base]
X
Y
Case [member / Case2 : Recursion]
member
X
member
Temp
Y

Figure 3.3: Definition **member**

As a clue to the ordering of layers, the literal cells are painted in a range of shades of one colour, the darkest shade applied to the front layer and the lightest to the back. The literal and function cells are transparent, giving a sense of depth to the layers. This is similar to the use of transparency to give the illusion of depth in the Macintosh OS X interface.

Literal cells are reordered by dragging their iconic representations in the list view from one layer to another. The relative shading of layers is adjusted whenever an existing layer is removed or a new layer is created. When the number of layers increases, the range of shades is subdivided, resulting in less differentiation between layers. Clearly, as the number of layers grows, the programmer may have to rely more on the layer list for ordering.

Since a query in Lograph has a similar structure to that of the body of a case, as we will see later, a query window is also layered. During execution, when a literal cell is replaced by a copy of the body of a case, the layers of the case body are placed in front of the existing layers of the query.

The list view and the layering scheme are two representations of the literal ordering. Although one may seem redundant in the presence of the other, we note that this could be beneficial when the range of shadings is finely divided across many layers. The user can hide the list view by clicking on the rightmost icon in the toolbar of a case definition window as illustrated in Figure 3.3.

We are not aware of any empirical evidence for or against multiple representations in programming languages and in particular visual languages. Shneiderman *et al.* [115] conjecture that multiple representations may be redundant if the user is familiar with one of the representations. Blackwell *et al.* [20] point out that this may not be true if "different representations highlight different types of information at the expense of others" or "understanding of a notation is often not complete and/or correct". Petre *et al.* [97] have hypothesised scenarios in which multiple representations might be used in software visualisation.

In Lograph, the two representations of literal ordering are complementary and have a non-intrusive nature considering that neither adds extra components to the graph. The list view can be hidden and we expect that programmers would rely mostly on the colouring scheme considering that programs are likely to have a limited number of literals in each case.

We mentioned in Section 2.1 that although the goal of a Lograph execution is to reduce a query to an empty graph, the definition of a "successful execution" of a query for practical purposes is slightly different. Accordingly, we introduced guard terminals to Lograph to prevent structures that have been computed by execution from being deleted. Once a terminal is selected to become a guard, drawn as a filled terminal, all its other occurrences will also become guards. Similarly, guard terminals propagate at run time. The first terminal of member at the left hand side of Figure 3.5 is an example of a guard terminal.

Just as the propagation of guard terminals at run time is reversible during backtracking or execution roll-back (explained in Section 3.3.2.1) instigated by the user, the propagation of guard terminals in the editor environment is also reversible. For example, when the user creates a new wire to connect a guard terminal to a non-guard terminal, all occurrences of the non-guard terminal become guards. Removing the same wire will transform the guard terminals which became guards through unification back to non-guard terminals. This ensures that the editing environment and the debugger are consistent. That is, similar actions in the editor and the debugger will have similar results although one is performed by the user and the other by the engine. The editor uses different internal annotations for the guard terminals annotated by the user and those propagated via unification. When a wire connecting two guard terminals is selected, the editor highlights the guard terminals that will be deleted if the wire is removed.

3.3.2 Lograph Debugger

A great number of software visualisation tools for Prolog have been developed. Software visualisation is the use of typography, animation, cinematography, and any other form of graphics to represent a program and its execution [101]. The aim of most Prolog visualisers has been to increase the user's understanding of the Prolog execution model either for pedagogical purposes, or to assist experienced programmers to debug Prolog programs. Examples of Prolog execution visualisers include Transparent Prolog Machine [46,47] and Textual Tree Tracer

[128]. Examples of other logic-based visualisers include ViMer a visual debugger for Mercury [24], Explorer developed for Oz [111], and the Execution Tree Viewer [21].

Visual Prolog tracers typically use AND/OR trees or an augmented form of them in order to visualise the internal state of a Prolog engine. The use of colour to highlight details of the unification process has also been encouraged [57] and attempted [89].

A known problem with graphical representations of the execution of Prolog is that since the code is textual, matching the output format of an execution tracer with the source code places a cognitive burden on the programmer [95]. Lograph, being visual, is not prone to this problem. We expect that the direct and obvious correspondence between the program and the query as it is transformed will significantly reduce the cognitive load on the programmer.

In Lograph, queries are created in a query window and executed in a query runtime window. A query window is very similar to a case window providing workspace, layer list, and the editing toolbar. The query runtime window is similar to a query window except that the editing toolbar is substituted by a control bar providing buttons for controlling the execution. The left hand side of Figure 3.5 shows a query window, and on the right hand side is a snapshot of its query runtime window when the first solution to the query is found.

A query can be executed in two different modes: **Run**, and **Debug**. The latter itself has two flavours, **Single Step** and **Animated**. In the **Run** mode, the result is computed by the Lograph interpreter and displayed in the query window with no intermediate illustrations of the progress of execution.

In **Animated** mode, the interpreter displays an animation of each rule application. The deletion of a function cell is animated by fading out the cell and releasing all attached wires, which shrink away from the disappearing function cell towards their other ends. The merge rule is animated by morphing two function cells into one. Animation of replacement is accomplished by expanding the replaced literal to the size of the case that replaces it, then fading in the body of the case together with the necessary connecting wires.

In **Single Step** mode, the interpreter animates each step but stops between steps. In both **Animated** and **Single Step** modes, backtracking is visualized by reversing the animations. In both **Single Step** and **Animated** modes, we have employed an automatic layout algorithm for adjusting the layout of the graph as it expands and shrinks. In Chapter 4, we will describe this algorithm in more detail.

The primary purpose of debugger tools for logic programming languages is to provide a finer account of the execution of a query. This often means providing a view of the internal state of the interpreter at run time. One of the aims of any debugger should be to reduce the cognitive load experienced by the user while navigating and inspecting the execution.

We have noted earlier that some of the cognitive issues listed by Storey *et al.* [123] are relevant to the design of Lograph's debugger. Accordingly, the debugger environment of Lograph aims to reduce a user's cognitive load by

- Facilitating navigation in execution.
- Providing orientation cues.
- Reducing disorientation.

We explain below how the Lograph debugger environment accomplishes each of the above.

3.3.2.1 Facilitate Navigation in Execution

The Lograph debugger supports two types of navigation: *directional* and *arbitrary*. Directional navigation allows a query to be run in two modes, **Do** and **Undo**. In **Do** mode, the literals in a query are expanded (replaced) as normal and backtracking is instigated by failure. Executing a query in **Undo** mode rolls back the execution animation, producing a reverse playback of normal execution. Note that reverse execution of backtracking looks like normal execution and the reverse execution of normal execution looks like backtracking, however, orientation cues will help the user to differentiate between the two. In **Do** and **Undo** modes, the animation of execution stops after the application of each rule. There are two other modes of execution, **Fast Forward** and **Rewind**. In the former, a query is run as in **Do** mode but in a continuous fashion until either a solution is found, execution is paused by the user, or the search tree is exhausted. In the **Rewind** mode, the query is run as in **Undo** mode until either a previously found solution is reached again, execution is paused by the user, or the execution is rewound all the way back to the original query.

Lograph supports arbitrary navigation of the execution by providing a graphical representation of the search tree depicting a top level view of the expansion of a query.

The search tree is drawn in a window separate from the query runtime window. Every node in the search tree corresponds to an application of the replacement rule and is represented by a miniature icon of the query graph at the time of the replacement. As an execution in **Do** mode proceeds, new nodes are added to the tree (except during backtracking). The current position node in the search tree is highlighted. Undoing the execution changes the visual effect of undone nodes from a solid colour to pale, thereby dividing the search tree into two parts. The part drawn in a solid colour represents the search space that has been traversed by the engine. The pale part corresponds to the parts of the search space that have been traversed at least once but because of roll-back, are not currently part of the execution. Rolling the cursor over a node enlarges the image icon to help the user more easily inspect the state of the query at that particular step. Clicking on a node in the search tree will make the query it represents the current execution state, be it in the past or in the future.

3.3.2.2 Orientation Cues

The Lograph debugger provides orientation cues by

- Indicating the current focus
- Showing how the execution arrived at the present state
- Indicating options for reaching new locations in the execution

The debugger indicates the current focus by animating the expansion of literals as they are replaced, highlighting the source of each failure as it is detected, and animating the merging of function cells.

The debugger represents the interpreter's execution mode, indicating how the execution arrived at the present state, with the four icons depicted in Figure 3.4 (a), (b), (c), and (d) corresponding to **Do-Expand**, **Do-Backtrack**, **Undo-Expand**, and **Undo-Backtrack**, respectively. Note that expand mode icons are actually drawn in green while those for backtrack mode are drawn in red.

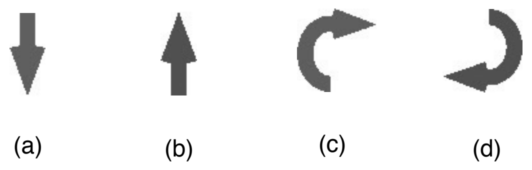


Figure 3.4: Execution mode icons

The debugger also indicates all possible directions in which the query can be executed from the current point on. This is achieved through buttons in the control bar of the query runtime window. These control buttons switch status as the result of changes in the interpreter engine. A combination of the activated and deactivated control buttons indicate the possible paths that execution can take from the current state. For example, the buttons in the control bar in Figure 3.5 indicate that the search for another solution can be continued by pressing the ⌃ button or that execution can be rolled back in either single step or rewind mode by pressing ⏴ or ⏪, respectively. Buttons ⏵, ⏩, ■, and ⌄ in the control bar drawn in their deactivated state in Figure 3.5 are for do, fast forward, pausing the execution, and finding a previous solution (if any), respectively.

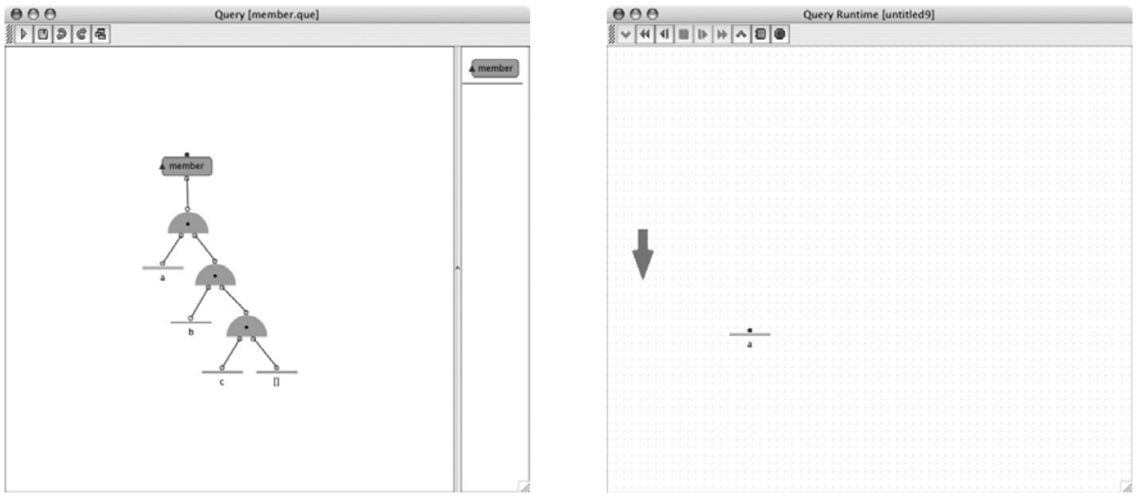


Figure 3.5: A query window and a query runtime window

3.3.2.3 Reducing Disorientation

The application of Lograph rules to a query may disorient the user if changes to the query graph are made abruptly. Discontinuity in the transformation of one layout to the next during the execution of a query will force the user to rebuild his or her *mental map* of the graph if the two layouts are dissimilar. We will give a more precise definition of mental map and graph similarity in Chapter 4. To reduce disorientation, the Lograph debugger

- animates the expansion of replaced literals
- animates the application of the merge rule
- animates the application of the deletion rule
- animates wire adjustments (explained in Chapter 4)
- automatically lays out the query in order to preserve the user's mental map

We will discuss Lograph's automatic layout algorithm and the rationale behind it in Chapter 4.

In the next section we will discuss the design of the back-end interpreter engine of Lograph, expanding on an earlier description presented in [14].

3.4 Interpreter Engine

Although flat Horn clause programs can be correctly executed by Prolog, the fine-grained view of unification provided by the surface deduction rules is lost. Consequently, direct Prolog execution of the flat clauses corresponding to a Lograph program cannot be used for the animated executions described above. This kind of execution is achieved by augmenting the flat clauses with "probes", inserted at appropriate places in a corresponding Prolog program, so that during the execution of a query, the interpreter can report to the graphical interface, the actions taken by Prolog that can be interpreted as equivalent to Lograph execution rules. Our interpreter deals with the merge and deletion rules explicitly, while search, backtracking, and replacement are provided by Prolog.

3.4.1 Translating Lograph to Prolog

In this section we describe how Lograph programs are transformed into flat Horn clauses which can be directly executed by Prolog.

To illustrate the translation process, we begin with a Lograph query. The textual representation of a query is a flat Horn clause consisting of a set of flat equalities of the form $x = f(\ldots)$ where x is a variable, and a list of flat literals $p_1(\ldots)$,$p_2(\ldots)$,$p_3(\ldots)$, … ,$p_m(\ldots)$, where each … stands for a list of variables. The order of the literals in the clause reflects the order of the corresponding literal cells in the layered structure of the Lograph query. The equalities are not ordered in any particular way since they represent function cells, and can be inserted anywhere in the clause. However, it would be natural to place the equalities at the beginning so merge and deletion can be applied to them before any application of replacement. Although this need not be the case, it will result in a more efficient execution by simplifying the query as soon as possible.

Based on the above, we start with the following Prolog query corresponding to the Lograph query.

```
        eqList,p1(…),p2(…),p3(…),…,pm(…).
```

where `eqList` is a list of equalities and `p1(…)` to `pm(…)` correspond to the literal cells in the query.

The body of the clause that represents a Lograph case has structure similar to that of a query. Translating a definition `p` to Prolog, produces a sequence of clauses with the following structure.

```
p(…):-
     eqs1,p11(…),p12(…),…,p1k₁(…).
p(…):-
     eqs2,p21(…),p22(…),…,p2k₂(…).
…
p(…):-
     eqsn,pn1(…),pn2(…),…,pnkₙ(…).
```

In the above, each clause corresponds to one case, and the order of the clauses represents the order of cases of the definition `p` set by the programmer in the editor environment. In the ith clause, the head `p(…)` corresponds to the head of the ith case; `eqsi` is the list of equalities corresponding to the function cells in the body of the case; `pi1(…),…,pik`$_i$`(…)` correspond to the k_i literal cells in the body of the case, their order obtained from the order of corresponding literal cells in the layered structure of the ith case window.

As an example, consider the translation to Prolog of the definition of member shown in Figure 3.3.

```
member(X,Y):-
     Y = •(X,_).
member(X,Y):-
     Y = •(_,Temp),
     member(X,Temp).
```

The translation described so far provides clauses and queries which will run correctly in Prolog. However, as we mentioned earlier, when executing a query in **Animated** or **Single Step** mode, the details of merge, deletion and replacement rules need to be visualised. In the next section we show how this is achieved.

3.4.2 Interpreting Lograph Programs

Lograph and Prolog are both based on Horn clauses. However, what distinguishes Lograph from Prolog is how it reveals the details of unification.

Once a Lograph program is translated to Prolog, the steps Prolog takes in executing the query must be refined into a series of applications of the three Lograph execution rules. This is essential for the Lograph editor to be able to provide a visualisation of the execution in **Single Step** or **Animated** mode.

The Lograph replacement rule is a simple version of resolution, in which only variable-to-variable substitution is required. Hence replacement can be handled directly by Prolog. Prolog deals with equalities by resolution with the clause `X = X.`, but Lograph deals with them by merge and deletion, which we therefore need to implement.

During the execution of a query, the interpreter must report to the editor environment when an execution rule is applied, identifying the cell or cells involved, and in the case of replacement, the clause which was used. This requires a one-to-one mapping between the Lograph components in the editor and items in the engine. This is accomplished by assigning unique identifiers to literal cells and function cells in the Lograph program, and their corresponding literals and equalities in the Prolog program.

The implementation consists of three modules. The *Editor* implements the visual programming environment of Lograph as described above. It includes an automatic layout algorithm for queries, described in Chapter 4, and an XML translator for saving and loading programs and queries. The Editor communicates with the *Model* module, which maintains structures representing Lograph programs and queries, translates them into Prolog for execution, and generates Ids. The Model module communicates with the *Interpreter Engine*, a Prolog program which consists of a set of predicates for executing the three Lograph execution rules and communicating with the Model. Editor and Model communicate in order to build and edit Lograph programs and queries. Model and Engine communicate to manage execution.

3.4.2.1 Replacement

To describe execution, we consider the following Prolog query, obtained from a Lograph query as described above:

```
eqList,p1(…),…,pm(…).
```

Model generates unique identifiers, `id1`,…,`idm`, for the literals in the above, modifying the query as follows.

```
eqList,p1(…,id1),…,pm(…,idm).
```

When replacement is applied to this query, it will be applied to `p1(…)`, replacing it with the body of the first clause for predicate `p1`. Translated directly from Lograph, this clause will have the form

```
p1(…):-
      eqs1,p11(…),p12(…),…,p1k(…).
```

However, literals in the query have been augmented with identifiers, so the head of the clause must be modified, by adding a new variable to match the Id in the query literal. Also, when this replacement is performed, Engine should tell Model that clause 1 for `p1` was used, and pass the identity of the replaced literal. In return, Model should generate Ids for the new goal literals introduced. This is achieved by a new literal added at the beginning of the body of the clause, and adding new variables to the body literals to accept the identifiers generated by Model. With these modifications, the clause has the following form.

```
p1(…,Id):-
      replace(Id,1,[Id1,Id2,…,Idk]),
      eqs1,
      p11(…,Id1),
      p12(…,Id2),
      … ,
      p1k(…,Idk).
```

When the literal `p1(…,Id)` in the query is replaced using this clause, the variable `Id` is instantiated to the constant `id`.

The `replace` predicate implements the required interface with Model. The list `[Id1,Id2,…,Idk]` contains new identifiers generated by Model for the literals in the body of the substituted clause.

In general, the ith clause corresponding to a Lograph definition `p` will have the form:

```
p(…,Id):-
     replace(Id,i,[Id1,Id2,…,Idj]),
     eqsi,
     pi1(…,Id1),
     pi2(…,Id2),
     … ,
     pij(…,Idj).
```

Next we consider the merge and deletion rules. Since these operate on function cells (equalities in the equivalent Prolog clauses), they also need to have identifiers so that when Engine reports the application of merge and deletion rules to Model, it can also report the cells involved. To account for these, the structure of a clause is further expanded, as follows:

```
p(…,Id):-
     replace(Id,i,[EqsIds,[Id1,…,Idj]]),
     matchIds(eqsi,EqsIds,EqsMatched),
     pi1(…,Id1),
     pi2(…,Id2),
     … ,
     pij(…,Idj).
```

Now let us recapitulate the replacement rule, describing the functionality added in the last step. When a replacement occurs, the Id of the replaced literal is already instantiated to a constant. The first literal in the body of the replacing clause, `replace`, will report to Model the Id of the replaced literal together with the case number of the clause. Model uses these to identify which literal cell in its query structure is being replaced. The case number will be used to correctly identify the case in the Lograph program, the body of which will replace the literal cell. This enables Model to generate Ids for the function and literal cells in the body of the replacing case. These Ids are then reported to both Editor and Prolog. Editor will use the Ids to identify the components that need to be redrawn. A successful execution of `replace` also instantiates `[Id1,…,Idj]` and `EqsIds` which are then assigned to the equalities in `eqsi` by execution of the `matchIds` literal.

3.4.2.2 Merge Rule

The next modification deals with the merge rule. Merge is applied to function cells with connected roots and identical names and arities. In the Prolog representation of Lograph queries, merge is applied to equalities which have the form $X = f(X_1,X_2,\ldots,X_s)$ and $X=f(Y_1,Y_2,\ldots,Y_s)$. One of the equalities is removed after variables X_j and Y_j are unified for all $1 \leq j \leq s$, and the Id of the removed equality is reported back to Model. Model responds appropriately depending on the mode in which the program is running.

The merge rule is applied on any two function cells in the query. Therefore, after a replacement, all the equalities in the query must be inspected for application of merge. This implies that in the body of each clause, all the equalities in the query must be available as well as the new equalities introduced by the clause itself. This is achieved by passing the list of equalities from one predicate to the next in the query, as follows:

```
p1(…,eqsList,EqList1,id1),
p2(…,EqList1,EqList2,id2),
… ,
pm(…,EqList(m-1),EqListOut,idm).
```

Note that in the original version of the query, the list of equalities `eqsList` preceded the first literal. It has now been inserted as a parameter to the first literal. The ith literal takes as a parameter `EqList(i-1)`, the list of equalities that exist once all preceding literals have been executed, and returns a modified list `EqListi`. With this in mind, clauses in the program must be similarly modified leading to the following structure.

```
p(…,EqListIn,EqListOut,Id):-
          replace(Id,i,[EqsIds,[Id1,…,Idj]]),
          matchIds(eqsi,EqsIds,EqsMatched),
          append(EqListIn,EqsMatched,Eqs),
          merge(Eqs,EqList0),
          pi1(…,EqList0,EqList1,Id1),
          pi2(…,EqList1,EqList2,Id2),
          … ,
          pij(…,EqList(m-1),EqListOut,Idj).
```

Here, the `append` literal attaches the incoming list of equalities to the list of equalities introduced by the clause itself. The new list, `Eqs`, is then passed to the `merge` literal which implements the merge rule, producing `EqList0`, a list of equalities, no two of which have the same left hand side (in Lograph terms, no two function cells connected at their roots).

The clauses that define the `merge` predicate also report to Model both successful and failed applications of merge. Recall that in Lograph, a failure occurs when two function cells connected at their root terminals cannot be merged. This can happen when the two function cells either have different names or unequal arities. When a failure is detected during execution of `merge`, the call to `merge` will also fail, causing backtracking. We discuss this further below.

3.4.2.3 Deletion Rule

Deletion removes from the query all function cells the roots of which have no other occurrences. This corresponds in the Prolog query to deleting equalities the left hand sides of which have no other occurrences. Deciding whether or not an equality is deletable, requires determining whether the variable on the left hand side has other occurrences. Prolog, however, does not keep track of the number of occurrences of a variable and therefore is not capable of making this determination.

To deal with this problem, we keep a count of the number of occurrences of each variable in the query. Every variable in our interpreter engine is represented by a pair, where the first element is the variable and the second is the number of occurrences. Maintaining variables in this form requires further modifications to the form of query and clauses, since whenever a replacement, merge or deletion takes place, the number of occurrences of the variables involved in the application of the rule needs to be updated. This leads us to the following format for the query.

```
          p1(…,eqsList,EqList1,varsIn,Vars1,id1),
          p2(…,EqList1,EqList2,Vars1,Vars2,id2),
          …
          pm(…,EqList(m-1),EqListOut,Vars(m-1),VarsOut,idm).
```

Here, `varsIn` is the list of variables occurring in the query. When the ith literal is executed, `Vars(i-1)` is the current list of variables in the query. Execution of the literal produces an updated list `Varsi`. Clauses in the program are further modified to include this process, as follows:

```
     P(…,VarsIn,VarsOut,EqListIn,EqListOut,Id):-
          replace(Id,i,[EqsIds,[Id1,…,Idj]]),
          unify(VarsIn,updateVars,Vars),
          matchIds(eqsi,EqsIds,EqsMatched),
          append(EqListIn,EqsMatched,Eqs),
          merge(Vars,VarsMerged,Eqs,EqsMerged),
          delete(VarsMerged,Vars0,EqsMerged,EqList0),
          Pi1(…,Vars0,Vars1,EqList0,EqList1,Id1),
          Pi2(…,Vars1,Vars2,EqList1,EqList2,Id2),
          …
          Pij(…,Vars(m-1),VarsOut,EqList(m-1),EqListOut,Idj).
```

In the above, `updateVars` consists of new variables introduced by this clause, together with variables that appear in the head of the clause. The counts of the latter need to be adjusted as a result of replacement of a literal in the query using this clause. The second element of each variable pair in `updateVars` is an integer (possibly negative), computed by subtracting the number of occurrences of the variable in the head of the clause from the number of occurrences in the body of the clause. Clearly, if a variable has the same number of occurrences in the head and the body of the clause, it can be omitted from `updateVars`.

3.4.2.4 Backtracking

As mentioned earlier, our interpreter implements merge and deletion explicitly while relying on Prolog for search, backtracking and the replacement rule. Although in case of failure Prolog will successfully undo the application of Lograph rules, it cannot undo the side effects of predicates which communicate with Model. There are three such literals, `replace`, and two other literals that report to Model the application of merge and deletion, along with the identifiers of the involved equalities. However, in order for Lograph to provide a visualisation of backtracking, our interpreter engine must also report to Model the undoing of replacement, merge and deletion during backtracking. We will show how the `replace` literal is modified to accomplish this. The other two literals which cause side effects can be modified in a similar way.

We introduce another literal called `undo_replace` which reports to Model the Id of the replaced literal and the number of the clause used in the replacement.

```
P(…,VarsIn,VarsOut,EqListIn,EqListOut,Id):-
          (replace(Id,i,[Eqs_Ids,[Id1,…,Idj]]) |
                                undo_replace(Id,i),fail),
          unify(VarsIn,updateVars,Vars),
          …
```

Backtracking into the OR structure causes execution of the `undo_replace` literal which reports the Id and the clause number to Model. This is immediately followed by execution of `fail` which stops execution of the clause body from being repeated.

As an example, consider the two cases of **member** in Figure 3.3. Model creates the following two clauses which it passes to Engine.

```
member(X,Y,VarsIn,VarsOut,EqListIn,EqListOut,Id):-
     (replace(Id,1,[EqListIn,[]]) | undo_replace(Id,1),fail),
     unify(varsIn,[v(_V1,1)],Vars),
     matchIds([e(Y=[X|_V1])],EqsIds,EqsMatched),
     append(EqsListIn,EqMatched,Eqs),
     merge(Vars,VarsMerged,Eqs,EqsMerged),
     delete(VarsMerged,VarsOut,EqsMerged,EqListOut).
member(X,Y,VarsIn,VarsOut,EqListIn,EqListOut,Id):-
     (replace(Id,2,[EqsIds,[Id1]]) | undo_replace(Id,2),fail),
     unify(varsIn,[v(_V2,1),v(Temp,2)],Vars),
     matchIds([e(Y=[_V2|Temp])],EqsIds,EqsMatched),
     append(EqsListIn,EqsMatched,Eqs),
     merge(Vars,VarsMerged,Eqs,EqsMerged),
     delete(VarsMerged,VarsOut,EqsMerged,EqList0),
     member(X,Temp,Vars0,VarsOut,EqList0,EqOut,Id1).
```

3.4.3 Running Programs in Debug Mode

As described earlier, the debugging environment of Lograph provides for the execution of a query in **Do** and **Undo** modes. In **Do** mode, Lograph employs depth-first search (inherited from Prolog) in order to solve a query. Literal cells are replaced using the cases of the corresponding definition, and, as in Prolog, failure causes subsequent cases to be tried in the order in which the user has arranged them. Following each replacement, the query is subjected to as many merge and deletion rules as possible before another replacement can take place.

In **Undo** mode, however, the effects of the **Expansion** (replacement) or **Backtracking** are undone in the order opposite to that in which they took place. Hence, the execution of a query in **Debug** mode can be in any of four possible modes: **Do-Expand**, **Undo-Expand**, **Do-Backtrack** and **Undo-Backtrack**. Note that **Do** and **Undo** refer to an execution control choice made by the user, whereas **Expand** and **Backtrack** correspond to the internal state of the engine.

The structure of the translated Lograph programs and queries presented in the previous sections provides for the execution of queries in the **Do** mode. In order to support the remaining two **Undo** modes, we must expand on the structure of the translated code and also extend the engine.

We refer to the user input as **Mode** and the internal state of the engine as **Direction**. **Mode** is a boolean where `true` corresponds to **Do** and `false` corresponds to **Undo**. **Direction** is also a boolean, where `true` means **Expansion** (replacement) and `false` means **Backtrack**.

Direction is recorded in a clause of the form `get_direction`(<value>), where <value> is true or false. The current clause for `get_direction` is retracted and asserted whenever the direction is toggled.

The undoing of the application of merge and deletion rules can be achieved by forcing a failure which instigates backtracking. Undoing replacement, however, is more problematic. We add a layer over Prolog's resolution rule to provide for the execution of a query in **Undo** mode. This is achieved through the use of two *meta-predicates*. The first meta-predicate, `replace`, deals with the boundary conditions (the first and the last clauses) of the replaced literal upon its initial invocation. If the execution is in **Do** mode, the literal will be replaced with its first case, otherwise it will be replaced with its last case. This suggests that in addition to the previous additional variables in the signature of a literal, each clause of a literal must also maintain the total number of cases that comprise the definition of the literal. Table 3-1 summarises the boundary conditions for the initial invocation of a literal in **Do** and **Undo** modes. Note that for clarity, the items in the last column of Table 3-1 are cross referenced to tags in Figure 3.6.

Table 3-1: Boundary conditions for the initial invocation of a literal.

Edit Mode	Execution Direction	CaseNumber	NewCaseNumber	Tag in Figure 3.6
Do	**Expand**	uninstantiated	1	A
Undo	**Backtrack**	uninstantiated	Max	D

As the reader may have already noticed, Table 3-1 lists only two cases of four possible combinations of edit mode and execution direction. This is because for the initial invocation of a literal the other two combinations will never occur. That is, when a literal is being invoked for the first time, it is either because the user has chosen **Do** mode while the engine is in **Expand** state or has chosen **Undo** when the engine is in **Backtrack** state.

The following literal shows the implementation of Table 3-1 which realises the first meta-call to the replaced literal.

```
replace(Literal):-
     Literal =..[Name|Args],
     Args=[_,Max|Rest],
     get_direction(Direction),
     bounds(Direction,NewCaseNumber,Max),
     NewArgs = [NewCaseNumber,Max|Rest],
     NewLiteral =..[Name|NewArgs],
     substitute(NewLiteral).
```

where `bounds` is defined by the following two clauses

```
bounds(true,1,_).
bounds(false,Max,Max).
```

Before any actual replacement, the engine checks whether the query must run in **Do** or **Undo** mode by a call-back to the user interface to obtain the value of **Mode** from the user input. The `get_direction` clause provides the current execution direction. Based on the combination of these two variables a choice for the next case is made.

Table 3-2 summarises the preparations for subsequent replacements of a literal after its initial invocation. Note that in this table, Execution Direction represents the state of the direction before the new case number is computed. Once the new case number is computed, direction will be set to New Execution Direction.

Table 3-2: Case selection for the recursive meta-call to a literal.

Edit Mode	Execution Direction	CaseNumber (1 < i < Max)	New CaseNumber	New Execution Direction	Tag	Action
Undo	**Expand**	1	-	**Expand**	B	fail
Do	**Backtrack**	Max	-	**Backtrack**	C	fail
Do	**Expand**	i	i	**Expand**	F	replace
Do	**Backtrack**	i	i+1	**Expand**	H	replace
Undo	**Expand**	i	i–1	**Backtrack**	E	replace
Undo	**Backtrack**	i	i	**Backtrack**	G	replace

The second meta-predicate `substitute` as listed in the following provides support for unlimited do and undo steps after the initial invocation of a literal. Note that the first clause substitutes the literal and the second clause recursively calls the literal with appropriate case number according to the user input and the internal state of the engine.

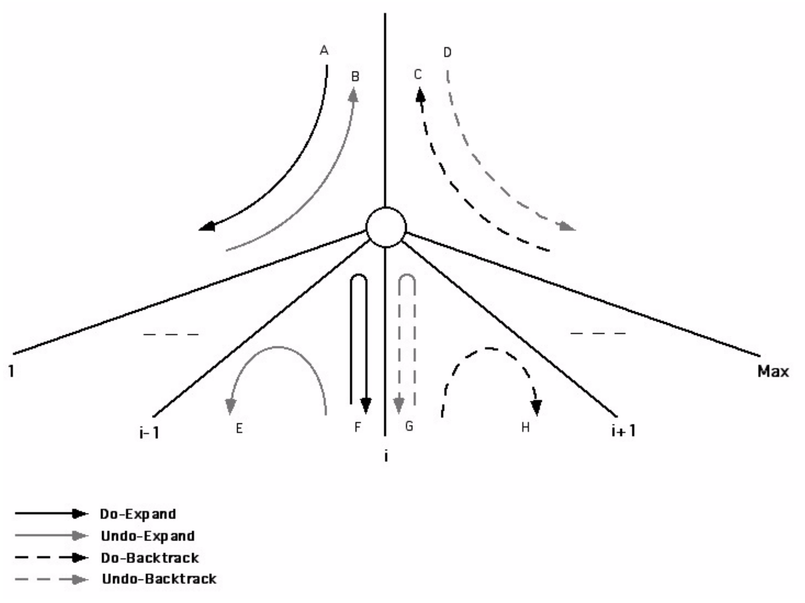


Figure 3.6: The control flow diagram for the replacement of a literal.

```
substitute(Literal):-
          Literal.
substitute(Literal):-
          Literal =..[Name|Args],
          Args=[CaseNumber,Max|Rest],
          get_direction(Direction),
          snap_out(Direction,CaseNumber,Max),
          ask_debug(Mode),
          compute(Mode,Direction,CaseNumber,NewCaseNumber),
          NewArgs = [NewCaseNumber,Max|Rest],
          NewLiteral =..[Name|NewArgs],!,
          substitute(NewLiteral).
snap_out(true,1,_):-!,fail.
snap_out(false,Max,Max):-!,fail.
snap_out(_,_,_).

compute(X,X,CaseNumber,CaseNumber).
compute(true,false,CaseNumber,NewCaseNumber):-
          toggle_direction,
          NewCaseNumber is CaseNumber+1.
compute(false,true,CaseNumber,NewCaseNumber):-
          toggle_direction,
          NewCaseNumber is CaseNumber-1.
```

3.4.3.1 Finding A Solution

As mentioned before, a successful execution of a Lograph query transforms the query into a solution, a not necessarily connected graph in which there are no literal cells, and at least one guard terminal, such that no function cells are connected by their roots, and the merge rule does not apply. Remember that guard terminals are used to prevent deletable function cells from being deleted.

We expand on the structure that represents terminals in the Engine to provide for guards. As we have described, a terminal is represented by a term including a variable and an integer representing the number of its occurrences. To these we add a list of terminal identifiers, each identifying an occurrence of the terminal in Editor. Every identifier is a term composed of a unique integer assigned to a terminal by the Editor and a boolean indicating weather the terminal is a guard or not. Clearly, when two terminals are unified, the union of the two lists of the terminal identifiers of the corresponding terminals will make up the list of terminal identifiers for the unified terminal. A call-back to Editor will report the propagation of guards as a result of unification.

Note that the propagation of guards after a replacement or merge is a side effect and will have no bearing on the execution of the rule itself. The propagation of guards will be undone during backtracking or undoing the application of an execution rule.

3.4.4 Finding All Solutions

As in Prolog, upon successful execution of a query, Lograph forces a failure to instigate backtracking in order to find more solutions. Analogously, in **Undo** mode, once a solution is found, forcing failure can be used in order to find previous solutions.

3.4.5 Just-in-Time Programming and User Inputs

Lograph handles literals that do not have any associated definition in two different ways depending on the mode in which it is running. In **Run** mode, before the execution begins, the user will be presented with a message indicating the literals in the query that do not have associated definitions and the execution will be aborted.

In **Debug** mode, before the initial invocation of each literal, a check for the existence of a definition is made. If such definition does not exist, it is created and a case window opens in which the user can build the first case of the definition. This feature of Lograph is inspired by similar just-in-time programming feature of Prograph [98]. Once the user finishes defining the new case(s) the Engine asserts the definition to the program and execution is resumed. Note that such a definition will not be retracted during backtracking.

Lograph also facilitates user inputs at run time. However, unlike Prolog that uses built-in or library predicates for this purpose, Lograph treats user input literals as a special type. A user input literal has a special icon and can have any name and any arity. The initial invocation of a user input literal will open a user input window similar to a case window. Although the user is not allowed to change the arity, cells of any type can be added. When the execution is resumed, the Engine will assert a new clause corresponding to the single case definition and immediately replace the literal with the newly defined case. Note that the user input window will not allow more than one case to be created. The asserted clause will be retracted during backtracking, allowing the user to try the query with different inputs without running the query multiple times.

One of the obvious advantages of Lograph's handling of the user inputs is that it saves the user the overhead of tokenising or parsing the input when it may contain both data and program (literal invocation). Also there is no need to reserve names for user input literals. The following is the implementation of the modified first meta-call to a replacing literal that deals with undefined and user input literals in **Debug** mode.

```
replace(Literal):-
            Literal =..[Name|Args],
            Args=[CaseNumber,Max|Rest],
            length(Args,Arity),
            current_predicate(Name/Arity),!,
            get_direction(Direction),
            bounds(Direction,CaseNumber,NewCaseNumber,Max),
            NewArgs = [NewCaseNumber,Max|Rest],
            NewLiteral =..[Name|NewArgs],
            substitute(NewLiteral).
replace(Literal):-
            Literal =..[Name|Args],
            Args=[_,_|Rest],
            last(Rest,Id),
            user_input(Id,Clause),!,
            (assert(Clause,Ref) | retract(Clause),fail),
            clause(Head,_,Ref),
            Head =..[NewName|_],
            NewLiteral =..[NewName|Args],
            NewLiteral.
```

3.5 Implementation Considerations

Our first implementation of Lograph was a proof-of-concept prototype, the core of which was a Java implementation of a standard logic programming interpreter. This had the usual advantages, such as ease of development and debugging, cross-platform executability and so forth. It also had the usual disadvantages, such as slow execution. Once past the proof-of-concept stage, we required a more "industrial strength" implementation, using more appropriate technologies and implementation techniques to obtain a fast and capable interpreter engine. Note that although a fast interpreter is not required when queries are executed in **Single Step** or **Animated** mode, which are mostly for debugging purposes, a fast interpreter for **Run** mode is still desired.

We chose the SICStus implementation of Prolog [126] as the basis for Lograph's interpreter. The release of SICStus Prolog current at the time of writing (version 3.12.2) provides a bidirectional interface to Java, as well as several constraint solvers.

3.6 Discussion

In this chapter, we have proposed a deterministic execution model for Lograph which, although styled after the linear execution of Prolog, does not force independent literals to be linearly arranged. We have also shown that there is some freedom in the ordering of rule applications, and have opted to apply execution rules in such a way that the size of the query graph is minimised. Other orderings may be advantageous in some situations, however.

We have described debugging features that combine forward execution and backtracking with user-control over roll-forward and roll-back, unlike other existing logic programming debugging environments. Such features can be carried over to LSD, providing detailed visualisation of the assembly of a design. We have also presented the design of the prototype interpreter engine, supporting these debugging features.

Although the proposed debugging facilities are intended to support visualisation of the assembly process in LSD, they reveal the details of logic program execution to an extent unavailable in other logic programming tools, and could therefore have pedagogical application.

Solids are 3D objects which manifest themselves in LSD programs as explicit components. An LSD program is therefore necessarily three-dimensional, requiring an editor that operates in 3D. In the 2D editing environment described in this chapter, however, we have used layering in the third dimension to deal with the ordering of literals in cases. Clearly we need to extend this notion to a three-dimensional editor. One possibility might be to use transparency instead of the shading employed in the 2D scheme. More opaque objects would be analogous to darker literals in the upper layers of a Lograph case or query, and more transparent objects would be analogous to literals in the deeper layers. Under execution, the fully opaque objects in a query are the ones that undergo transformation first, and objects become more opaque the closer they get to being executed.

4 Automatic Layout

One of the most important goals of a debugger for any language is to help the user to better comprehend the execution of a program. Therefore, debuggers usually provide the user with a detailed account of execution steps. In the case of LSD and Lograph, execution is a series of transformations of a query graph and as a result, execution steps can have a natural visual representation. However, as the query graph is transformed, the initial static structure of the graph is lost as new components are added to the query and existing ones deleted. Hence, a debugger for LSD and Lograph could substantially benefit from an automatic layout algorithm that adjusts the layout of the query graph during the execution. In this chapter, we will describe such an algorithm, originally reported in [11].

As discussed in Chapter 3, a query in Lograph can be executed in **Run** or **Debug** mode, the latter having two variations, **Single Step** and **Animated**. In **Run** mode, a query is invisibly executed and the result displayed in a query window. In **Debug** mode, the interpreter displays an animation of each rule application. Deletion of a function cell is animated by fading the cell away, and releasing all attached wires, which shrink from the disappearing function cell towards their other ends. The merge rule is animated by morphing two function cells into one. During replacement the replaced literal is expanded to the size of the case that replaces it, then crossfades with the body of the case. In **Single Step** mode, the interpreter animates each step but stops between them. In both **Animated** and **Single Step** modes, backtracking is visualised by reversing the animations.

In **Debug** mode, once the execution of a Lograph query commences, as the result of the application of Lograph rules, the query graph constantly changes. Therefore, a Lograph query in **Debug** mode is a highly dynamic graph to the extent that the initial user-defined layout of a Lograph query usually has nothing in common with its solution graph(s). However, the execution of a Lograph query in **Debug** mode can be perceived as the transformation of the query graph to that of a solution where each step in the execution corresponds to one or more steps in the transformation. The less difference between the layouts of each pair of consecutive steps in the transformation, the more *natural* the transformation will seem to an observer. This notion, known as *inbetweening* to animation artists, is one of several criteria taken into considerations when Lograph queries are run in **Debug** mode.

The modifications to a query graph during execution reveal the details of resolution and unification, and are therefore key to understanding execution and debugging code. Hence, the way in which a graph is transformed from one layout to the next during the application of an execution rule is as important as the initial and final layouts themselves. A verbose and animated account of transformations can significantly aid an observer to better understand why or how the query graph arrived at its current state. This is similar to the goal of traditional logic programming tracing techniques which aim to improve a programmer's understanding of the execution of code by revealing the internal states of the interpreter at run time. The graphical nature of Lograph, gives us even better opportunities for providing a meaningful visualisation of execution.

Automatic graph drawing has many applications, for which various general techniques have been developed [127]. Graph drawing techniques can be divided into two major categories. A*lgorithmic methods* apply algorithms to a static graph structure to satisfy certain aesthetic criteria, such as minimising the total number of edge crossings, or minimising the area required. The second group, known as *declarative methods*, attempt to satisfy user-specified constraints while determining graph layout.

A "postmortem" approach to graph drawing and animation is described in [42]. This technique, *foresighted layout adjustment*, applies when the changes to a graph are known in advance, rather than being determined on the fly. This algorithm tries to find a series of graph layouts that, when played in sequence, illustrates the transformation from initial to final layout. The key is to maximise the similarity between consecutive layouts.

Branke [22] and Cohen *et al.* [28], separately give comprehensive accounts of methods for dynamic graph drawing. Other approaches have attempted to generalise sample drawings provided by the user to determine how graphs should be arranged [39]. However, there have been very few attempts to exploit the semantics of a graph in determining appropriate layout [65].

4.1 Mental Map

In many applications that employ dynamic graph drawing, the user has an understanding of the underlying meaning of a graph undergoing transformation. In such applications it is important that changes to the graph preserve this "mental map". Although this term is frequently used as an intuitive concept, various attempts have been made to formalise this notion. For example, Misue *et al.* in [78] describe three formal models for an observer's mental map of a graph, *orthogonal ordering*, *proximity relations*, and *topology preserving*. Under the orthogonal ordering model, a graph transformation must preserve the vertical and horizontal relationships between each pair of vertices in the graph. A transformation preserves proximity relations if nodes that are close together stay close as the graph is changed. A transformation is topology preserving if the dual of the graph remains unchanged. The dual of a graph G is the graph in which the set of vertices are the regions of the plane delineated by the edges of G, and edges indicate adjacency of regions.

Lyons *et al.* in [74] aim to improve graph layout by employing a process called *cluster busting* for more evenly distributing nodes in a graph, while simultaneously trying to minimise the difference between two successive layouts by anchoring.

Graph drawing techniques such as those mentioned above employ some metric for quantifying the difference between two graph layouts, and attempt to minimise the difference in order to reduce the cognitive load inflicted when an observer is required to adjust his or her mental map as a result of a graph transformation. Since they consider only syntactic properties, they are general enough to be applied to any graph; however, they ignore the edges of the graph, and therefore the semantics.

4.2 Drawing Query Graphs

As outlined earlier, the execution of a Lograph query is depicted by the debugger as a transformation from query graph to solution graph in which each execution step corresponds to one or more graph transformation steps. Each execution step reveals a detail of resolution or unification, so being able to follow it as it happens is crucial to the programmer's understanding of the entire transformation from query to solution.

Our approach to the layout of query graphs employs various traditional graph layout techniques, but also accounts for the semantics of Lograph in order to better minimise distortions in the user's mental map. Note that preserving the user's mental map of a Lograph query is a notion local to each execution step. That is, although we aim to disturb the user's mental map as little as possible by minimizing the differences between consecutive layouts, we clearly cannot preserve it from the beginning to the end of an execution, or even across several steps. Our aim, therefore, is to make the inevitable complete transformation from initial query to solution, a smooth, observable process, drawing the user's attention at each step to the portion of the graph where changes occur.

In summary, the design of our layout algorithm is based on the following assumptions and observations.

- Since the layout of a query or body of a case is determined by the programmer, it is safe to assume that it embodies some aspects of the programmer's mental map. Therefore, the animation of execution should preserve such information as far as possible, at least in the short term. Note that although Prolog code is frequently indented in ways meaningful to the programmer, such secondary notation is ignored by Prolog tracers and debuggers.
- Constraining changes so that they occur at a fixed point, (the *locus*, which we have chosen to be the centre of the query window) will avoid having to scroll the entire graph before an application of the replacement rule, so that the user does not have to guess where on the screen the next changes will be applied. This increases the consistency of the environment by making animation behaviour more predictable and reducing the need for visual search and potential disorientation.
- The changes in the layout of a query graph should be minimised. Changing the position of too many nodes simultaneously may force the user to try to keep up with the changes in the layout of the graph at the expense of understanding the execution itself.
- Application of the replacement rule to the literals in a query results in the expansion of the graph, whereas the application of both merge and deletion decrements the number of nodes in the graph, thereby shrinking it.

We now consider these changes in detail, and describe how our layout algorithm deals with them.

4.2.1 Replacement

When a literal cell in a query graph is replaced by the body of one of the cases of the corresponding definition, the graph must be adjusted to make room for the replacing case, with minimal change to the layout. We also want to constrain the application of each replacement to be near the locus. To achieve this, we need to make sure that a literal subject to the replacement rule is in the vicinity of the focus point when the replacement occurs.

Like query literals in Prolog, literal cells in a Lograph query are ordered for execution as discussed in Section 3.1.3. Hence, each literal should be positioned in the graph depending on how soon it could be executed. Accordingly each literal cell in a query graph is constrained to be within a maximum distance of the locus, computed as a function of the position of the literal cell in the scheduled execution order. That is, literal cells further away in execution order, can be repelled further from the locus. Note that this maximum distance constraint must be satisfied to ensure that during replacement the replaced literal is always in the vicinity of the locus.

When the interpreter reports the replacement of a literal cell L with a case of the corresponding definition, the position in the query window of the centre of the replacing case must be determined. To accomplish this, the centre of the case is placed on the line which connects the locus to the corner of the literal L which is furthest from the locus, in such a way that the literal is inside the circle that encompasses the case. This ensures that when the replaced literal is expanded to the size of the replacing case, it is translated only minimally at the same time. Ideally, the centre of the case will coincide with the locus.

Once the centre of the replacing case has been determined, every cell in the query that intersects the area that will be occupied by the replacing case must be moved away from the replaced literal cell to make room for it to expand. The extent and direction of the translation of each cell is calculated using an algorithm based on Voronoi diagrams [41]. The *Voronoi diagram* of a set P of n points in the plane, called *Voronoi points*, is a subdivision of the plane into n regions, one for every point in P, in such a way that every point in a region is closer to the corresponding Voronoi point than to any other point in P. The Voronoi diagram of a graph is computed using the sweep-line algorithm due to Fortune [51].

In the following, a cell is *anchored* if its distance from the locus is already greater than or equal to its allowable maximum. The algorithm for adjusting the graph before replacement is as follows:

Let R be the replaced cell, then:

```
while   some cell A≠R in the query overlaps the replacing case, any cell closer to R than
        A, or an anchored cell
{
        Move each such cell A away from R along the line connecting the centre of R
        and the centre of A, until it no longer overlaps the case, or until its centre is at
        the boundary of its Voronoi region, whichever occurs first
}
```

Note that in practice, since no cells in a query overlap, the only cells that need to be inspected initially are those that overlap the replacing case. Similarly, in subsequent iterations, the cells involved are those that overlapped cells moved in the previous iteration.

Figure 4.1(a) depicts a query graph overlaid with the Voronoi diagram of the set of points consisting of the centres of the cells in the graph. The locus in this and other subfigures of Figure 4.1, is marked with cross hairs for explanatory purposes. Suppose that replacement is about to be applied to the **member** cell in this query. Figure 4.1(b) shows the Voronoi diagram of the query graph of Figure 4.1(a) together with the case of the **member** definition to be applied, positioned in the manner described above. Figure 4.1(c) shows how the three cells that overlap the case are to be moved. Two of them are moved outside the boundary of the replacing case, while the third is pushed to the boundary of its Voronoi region. Figure 4.1(d) depicts the new Voronoi diagram computed after the points are moved, and shows that the point that was prevented by the boundary of its Voronoi region from moving outside the case, must be moved again in this iteration. Figure 4.1(e) shows the new Voronoi diagram and the third translation required for the same point, which will this time move it beyond the replacing case.

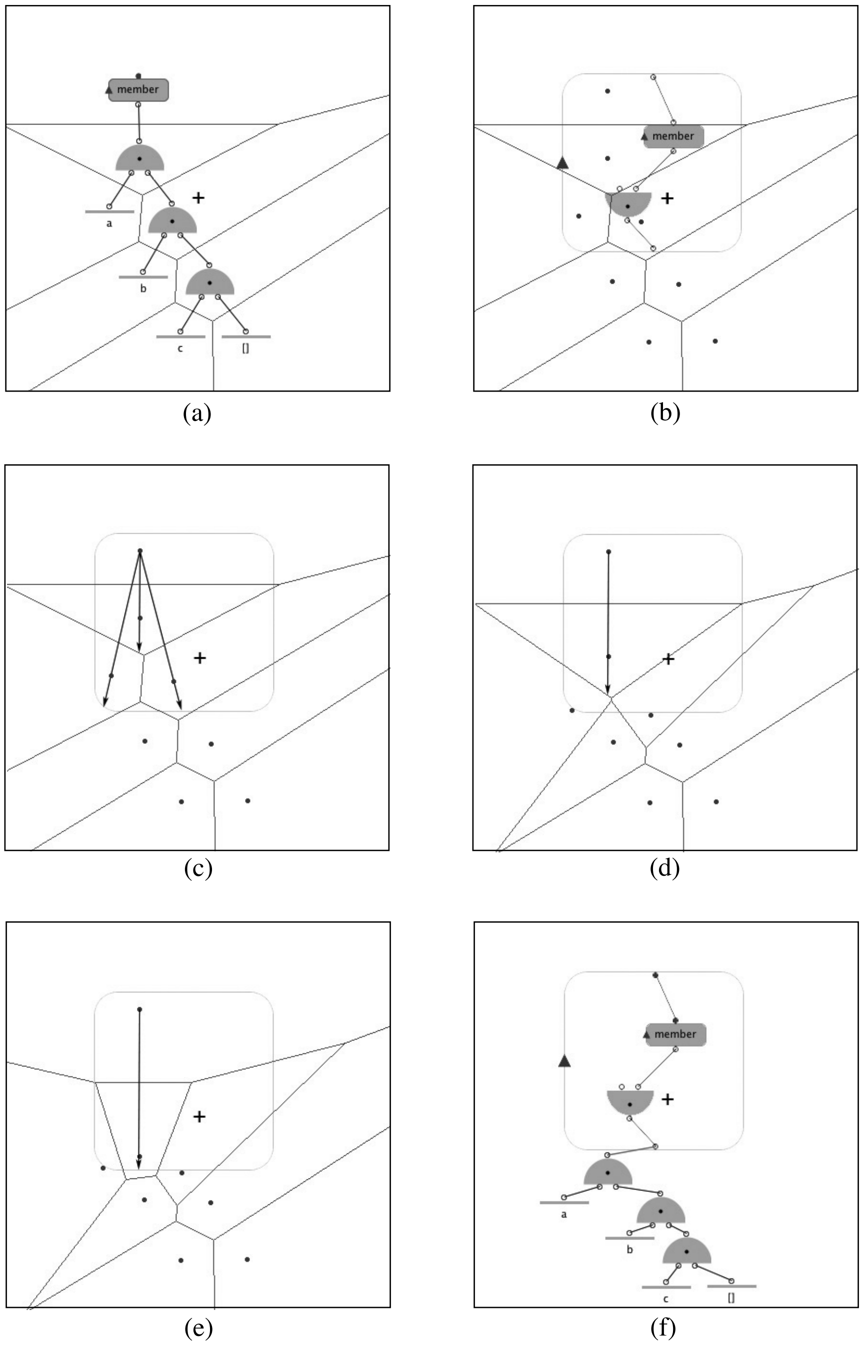


Figure 4.1: Expansion of a query graph

Further iterations will deal with overlaps which occurred as a result of the steps we have discussed so far, rippling other cells outwards, producing a slight expansion of the graph away from the replaced literal. Figure 4.1(f) shows the final layout after all overlapping cells have been pushed away from each other, and the actual replacement has been done.

Note that constraining cells to their Voronoi regions where it is possible to do so preserves proximity relations.

4.2.2 Foresighted Wire Adjustment

Once a case has replaced a literal in a query graph, the terminals of the head of the replacing case become orphans. A terminal is an orphan if it does not belong to a cell. For example, there are two orphan terminals in the graph of Figure 4.1(f), on the boundary of the inserted case body.

A connected graph of terminals and wires is called an *occur graph*. An occur graph with $n \geq 1$ terminals is *minimal* if it is a spanning tree. The Lograph editor guarantees the minimality of occur graphs in cases and initial query graphs by preventing the user from creating cycles.

After a literal has been replaced by a case body, each resulting orphan terminal connects two occur graphs, one originating from "inside" the replacing case, the other originating from "outside". In the following, we refer to each of these two occur graphs as *a side* of the orphan terminal's occur graph.

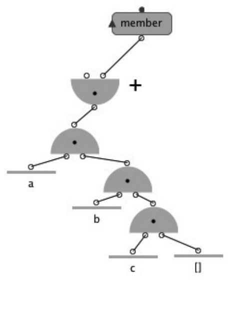


Figure 4.2: Wire adjustment

After the replacement of a literal, each orphan terminal is connected to $n \geq 0$ terminals on one side and $m \geq 0$ terminals on the other side. Orphan terminals are redundant and can be removed from the query. For example, removing the orphan terminals and the boundary of the replacing case in Figure 4.1(f) will result in the graph of Figure 4.2.

Clearly, removing an orphan terminal can be done quite simply; for example, by sliding the terminal down any of the wires incident on it until it coincides with the terminal at the other end. Although this guarantees that the occur graph will remain minimal since the numbers of edges and nodes are both decremented, it may not result in the best overall configuration from a graph comprehension standpoint. Our wire adjustment algorithm aims to highlight the application of the merge rule and potential unification failures by directly connecting the roots of the function cells involved in merge or failure with a single wire, while maintaining the minimality of each occur graph.

The algorithm takes this notion even further by biasing the wire adjustment towards a configuration that is likely to require less modification in the next few execution steps in order to highlight future merges and failures. If the algorithm determines that there is no possible merge or failure within the next few steps of the execution, it tries to increase the readability of the graph by exploiting data structure patterns.

Anticipating applications of the merge rule and unification failures can be achieved by peeking into the execution in the engine or simulating certain parts of the execution in the editor itself. We chose the latter approach in order to clearly separate the roles of the interpreter and editor, and to keep the interpreter simple. The look-ahead information can be used to gradually transform the current layout of the query to one which is better suited to the execution of the query in the next few steps. The algorithm proceeds as follows.

1. For an orphan terminal, traverse both sides of the occur graph marking each terminal as *root(*N*,n)*, *function(*N*,n)* or *literal* depending on whether it is the root of a function cell named N with arity *n*, a non-root terminal of a function cell named N with arity *n*, or a terminal of a literal cell. The extra information attached to the *root* and *function* marks is useful when

searching for data structure patterns, such as lists constructed as cascading binary function cells (eg. using the list constructor function, or the successor function).

Note that there are at most two root terminals in the occur graph of an orphan terminal, one on each side, otherwise two root terminals in the same side would have caused either a merge or a failure before the replacement took place. Note that we assume that the Lograph editor prohibits the user from connecting two root terminals in the body of a case since such a construction will lead inevitably to merging or failure.

2. For each *literal* terminal **t**, peek inside the first case of the definition corresponding to the associated literal. Let **t**' be the head terminal of this case corresponding to **t**.
3. Mark **t**' *anonymous* if its occur graph consists just of the node **t**'. Otherwise, mark all terminals in the occur graph of **t**' as in (1), except that terminals of literal cells are marked as *undetermined.*
4. Mark **t** with the "strongest" mark found in the occur graph of **t**', where marks are ordered from strongest to weakest as follows:
   anonymous > *root* > *function** > *undetermined*
   Here *function** indicates that **t** is tagged with not just a single *function* mark, but with the list of all *function* marks found in the occur graph of **t**'.

The above process will assign to each terminal in the occur graph of the orphan terminal one of the following marks:

root
*function** (see above)
undetermined
anonymous

Note that for each *literal* terminal in the above, look-ahead is confined to just one case of the definition for the associated terminal cell. Also, the algorithm is not recursively applied to terminals inside this case. This is because the aim is to adjust the wires to prepare for changes that will occur in the next few steps. Looking too far ahead, or peeking into other cases, would take into account events which are much less likely to occur, so the payoff would be marginal at best.

Next, the orphan terminal is removed and its occur graph modified based on the marks of two terminals t_1 and t_2, one from each side of the occur graph, chosen as follows. If one side contains a terminal r marked *root*(N,*n*), then t_1 = r; then if the other side contains a terminal s marked *root*(N,*n*) or *function(*N,*n*) then t_2 = s. Otherwise choose t_1 and t_2 arbitrarily.

The wires in the occur graph are now adjusted as described in column 3 of Table 4-1, based on the marks of t_1 and t_2 given in column 1, and the connectivity of t_1 and t_2 to the orphan terminal in column 2. For example, direct-direct indicates that t_1 and t_2 are each directly connected to the orphan terminal via single wires.

Table 4-1: Wire adjustment algorithm

Marks	Connectivity	Action
root-root	direct-direct	Float the orphan terminal towards whichever of t_1 or t_2 minimises the total length of the wires in the orphan terminal's occur graph
root-function		
root-root	direct-indirect	Float the orphan terminal towards whichever of t_1 or t_2 is indirectly connected to the orphan terminal. Eliminate the loop.
root-function		
root-root	indirect-indirect	Split the orphan terminal in two. Float one towards t_1 and the other towards t_2. Eliminate the loop.
root-function		
other	-	Float the orphan terminal towards a terminal on its occur graph which is not anonymous, does not create a loop, and minimises the total length of the wires in the orphan terminal's occur graph

Based on the marks of t_1 and t_2, an orphan terminal may be split into two orphans connected by a single wire where one carries all the wires connected to one side of the original orphan terminal and the other carries the wires from the other side. The two orphans then float towards t_1 and t_2.

Note that floating an orphan terminal towards a terminal to which it is not directly connected will introduce a cycle in the occur graph, which can be eliminated by removing one of the participating wires, thereby maintaining the minimality of the occur graph. It would be natural to choose one of the two wires that are directly connected to the chosen terminal since the user's attention will already be drawn to it by the orphan terminal floating in its direction.

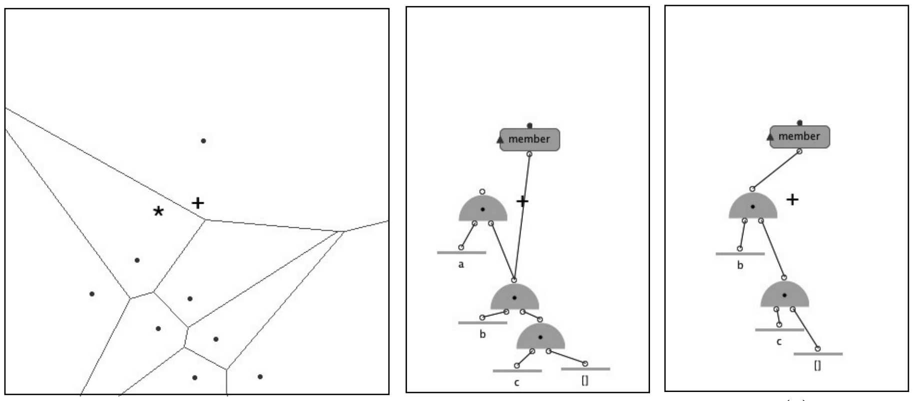
Finally, we note that in the *root-function* cases, the occur graph is structured in such a way as to identify function cells of the same name and arity that participate in a recursive structure, such as a list constructed from • cells.

4.2.3 Merge and Deletion

We mentioned earlier that applying the replacement rule will expand the area of the graph and will usually add nodes. If the body of the replacing case has no cells or one cell, the number of nodes in the query graph will be decremented or remain unchanged. Nonetheless, assuming that the bounding box of the replacing case of a literal is always larger than the icon of the replaced literal, the area of the graph will expand.

Applications of both merge and deletion decrement the number of nodes in a query graph. Hence, the layout algorithm treats them similarly. After a merge or deletion, the space occupied by the removed function cell can be reclaimed. The nearest cell occupying a Voronoi region neighbouring that of the removed cell which is farther from the locus than the removed cell will be tagged for translation. For example, in Figure 4.2, merging the two function cells connected by their roots will remove one of them. In this example the upper one is removed. The cell which is nearest to the removed cell but further from the locus is, in fact, the other function cell participating in the merge. Figure 4.3(a) shows the Voronoi diagram of the query graph of Figure 4.2 after the function cell is removed, with the position of the removed cell marked with an asterisk.

A tagged cell moves towards the position of the removed cell, and is constrained to remain within its Voronoi region. Before the actual translation of the tagged cell, the cell to be translated next is tagged in a similar fashion, the cell “a” in this example. The translation of the first tagged cell takes place and the algorithm iterates, terminating at the boundary of the graph. This effect will ripple out from the deleted function cell towards the boundary of the graph, moving cells inward. The overall result is that the graph shrinks towards the locus. Figure 4.3(b) shows the query graph after the first iteration of this algorithm. Figure 4.3(c) shows the final layout after the application of the merge, the subsequent deletion, and the layout adjustments.



(a) (b) (c)

Figure 4.3: Shrinking a query graph

4.2.4 Automatic Layout for Run Mode

The automatic layout algorithm described so far deals with the gradual transformation of a query graph to a graph representing a solution or simply an empty graph. In **Run** mode, however, the information about the gradual transformation is absent. Hence, a different approach for the layout of the solution graph is required. We observe that in **Run** mode, when a solution is found, the only information available to the graph layout algorithm is a list of variables and a list of equalities corresponding to terminals and function cells, provided by the interpreter engine. Although the equalities directly correspond to function cells, the wires connecting the terminals representing the multiple occurrences of variables cannot easily be inferred. However, we can assume the existence of a fully connected occur graph for each variable, and employ a spanning tree construction algorithm to minimise the number of wires required for drawing the solution. The criterion for choosing one spanning tree over another could be, for example, to minimise the total number of edge crossings in the layout of the solution graph. This would require the algorithm for the computation of each spanning tree to consider the spanning trees of other occur graphs to achieve its objective. Another approach might be to find the minimum spanning tree of a fully connected occur graph, where the weights of the edges in the graph correspond to their lengths for a tentative layout of the graph. This would provide a rich context for further investigation of an automatic layout algorithm in **Run** mode.

There are also other approaches to automatic layout of graphs based on the spring model algorithm [44,124], which could be customised to serve as an automatic layout algorithm for **Run** mode. This, however, requires further investigation.

Note that the interpreter engine tracks guard terminals regardless of the mode in which it is running.

4.3 Discussion

We have presented an automatic layout algorithm for queries in the Lograph debugger, the goal of which is to provide a smooth and natural transformation of a Lograph query graph to a solution by gradually adjusting the user's mental map while capitalising on certain aspects of the execution at run time.

A Lograph query graph and a solution usually have little or nothing in common. The goal of our automatic layout algorithm is to make the transformation of a query graph to a solution comprehensible by gradually adjusting the user's mental map as the execution proceeds. The algorithm attempts to achieve this by minimising and controlling the changes to a query graph by applications of the Lograph execution rules.

We have employed some traditional graph adjustment methods and augmented them with a foresighted wire adjustment technique which exploits the semantics of a query graph to better adjust the user's mental map as the changes to the geometry of the graph are taking place.

Since our ultimate goal is to implement LSD, which will require a 3D editor and debugger, we need to extend our layout algorithm to 3D space. This should be straightforward considering that the notions used in both the traditional graph adjustment techniques we have employed and foresighted wire adjustment can be extended to 3D.

Finally, as an aside, we note that it is generally agreed that a visual logic programming language would be beneficial for teaching logic programming [46,83]. We believe that, in addition to providing the underpinnings for LSD, Lograph might also provide a useful environment for teaching logic programming, and for building and debugging logic programs. Animating query transformations during both forward execution and backtracking may be a valuable substitute for typical textual tracing techniques. This, however, cannot be determined without empirical studies which have yet to be performed.

5 A Formal Model for Solids

At the heart of a visual design environment with programming capabilities must lie a solid modeler. Solid modeling deals with the creation, manipulation and interaction of solids in a design space. A more precise definition of solid modeling will be presented in Chapter 6. For now, it is sufficient to mention that a solid is a mathematical representation of a 3D object which includes its geometric specifications and may also include other properties such as mass, material, centre of gravity, elasticity, and electromagnetic characteristics.

It was mentioned in Chapter 1 that the original LSD, as a simple exploration of the notion of design in a visual language, had just one operation on solids. However, the selection of operations required in a design language depends on the domain of application, so LSD clearly needed to be extensible. As a basis for extensibility, a very general notion of design space was required, providing just enough detail to capture the essence of solids in logic. To meet this need, a formal model for characterising objects in a design space was proposed in [37], in which a design space consists of multidimensional space (normally 3D) augmented with extra dimensions representing properties. This formalisation also defines the concept of an operation on solids, and selective interfaces that allow solids to declare what operations can be applied to them. Based on this model, the required generalisation of LSD was made. It is important to note that this formalisation characterises solids and operations only to the extent required by LSD, so does not address any of the practical details of solid modeling. Nevertheless, we will refer to it as the "solid modeler" in the following.

The major motivation for providing the formal model in [37] was to generalise LSD with the notion of a general operation rather than extending LSD with an exhaustive list of operations. However, other requirements for the solid modeler, as listed in Chapter 1, were not addressed. In this Chapter, we will extend the solid modeler to formally describe the notions of sample look for solids, high-level parameterisation, reduction of solids, behaviours, and motion modeling. We also make some small but important adjustments to some of the definitions in [37] to better capture the notion of invalid objects.

Since our solutions to the aforementioned problems involve an extension to the definitions given in [37], in the next section, we will briefly summarise the core concepts presented there, modifying some of the original definitions to remove an anomaly in the way they dealt with invalid objects. The reader is encouraged to consult [37] for a thorough discussion. First, however, we define some notation.

If x and y are sequences, we denote the concatenation of x and y by $x{\bullet}y$. For example, if x is $x_1,x_2,\ldots,x_n$ and y is $y_1,y_2,\ldots,y_m$ then $x{\bullet}y$ means $x_1,x_2,\ldots,x_n,y_1,y_2,\ldots,y_m$. If x is a sequence and y is not a sequence, for consistency, by $x{\bullet}y$ we mean $x_1,x_2,\ldots,x_n,y$ and by $y{\bullet}x$ we mean $y,x_1,x_2,\ldots,x_n$. We use () to represent a sequence of length zero. If x is a sequence of length n and $1\leq i \leq n$, we denote by $\bar{x}_i$ the sequence of length $n-1$ obtained by removing the i^{th} element from x. If x is a sequence of length n and $p=p_1,p_2,\ldots,p_k$ is a sequence of distinct integers such that $1\leq p_j \leq n$ for all $1\leq j \leq k$, by x_p we mean the sequence $x_{p_1},x_{p_2},\ldots,x_{p_k}$ and by $\bar{x}_p$ we mean a sequence obtained after removing items in x_p from x. If x is a sequence of length n, we denote by x_{m*} and x_{*m} for some $m \leq n$, the length m prefix and length m suffix of x, respectively. If x is a sequence, we may also use x to denote the set of elements in x.

If **C** is a formula, by var(**C**) we mean the set of all free variables of **C**. If **C** is a formula, and a is a term, we use the standard notation $\mathbf{C}_x[a]$ for the formula obtained from **C** by substituting a for every occurrence of x. We extend this notation to sequences of variables and terms of the same length. That is, if $x_1,x_2,\ldots,x_n$ is a sequence of variables and $a_1,a_2,\ldots,a_n$ is a sequence of terms, then $\mathbf{C}_{x_1,x_2,\ldots,x_n}[a_1,a_2,\ldots,a_n]$ is the formula obtained from **C** by substituting a_i for every occurrence of x_i for each i ($1\leq i\leq n$). We might also combine single variables and sequences of variables in this notation. For example, $\mathbf{C}_{x,y}[a,b]$ means that each of x and y is either a single variable or a sequence, and each of a and b is either a single term or sequence of terms, of the same length as x and y, respectively. Another example, is $\mathbf{C}_{x\bullet y}[a]$ which means that a is a sequence of terms such that $|a| = |x\bullet y|$. $\mathbf{C}_{x\bullet y,z}[a,b]$ means that a is a sequence of terms such that $|a|=|x\bullet y|$, and b is a sequence of terms iff z is a sequence of variables of the same length as b.

We use a set of formulae as an alternate notation for a formula which is the conjunction of the formulae in the set such that none of them are conjunctions. The conjunctions of two or more sets of formulae is understood to be the conjunction of the formulae in the sets. If **C** is a set of formulae, by $\mathbf{C}^{=}$ we mean the set of equalities in **C**. If x is a variable we denote by $\mathbf{C}_x$ a set of formulae of the form $\{\ \mathbf{E}_x[u] \mid \mathbf{E}\in\mathbf{C}\ ,\ \mathrm{var}(\mathbf{E}_x[u]) \neq \varnothing\ \}$, where u is some constant. If x is a set of variables, $\mathbf{C}_x$ is defined in the obvious way. Clearly, $\mathbf{C}_x$ is not well defined since it depends on the choices of u. However, that will not be important in the context in which we will use this notation. If **C** is a set of formulae and $x\subseteq$ var(**C**), by $\mathbf{C}^x$ we mean the set of formulae $\{\ \mathbf{E} \mid \mathbf{E}\in\mathbf{C}\ ,\ x\cap\mathrm{var}(\mathbf{E})=\varnothing\ \}$. If x is a variable, $\mathbf{C}^x$ has the obvious meaning $\mathbf{C}^{\{x\}}$.

If Φ is a partial function from A to B, by scope(Φ) we mean the set $\{\ x \mid x\in$ A and Φ is defined for $x\ \}$ and by the image of Φ we mean the set $\{\Phi(x) \mid x\in \mathrm{scope}(\Phi)\}$.

If **C** is a formula, Γ is a partial function, and $x\bullet y\bullet u\subseteq$ var(**C**), $\Phi(x\bullet y\bullet z)=[\mathrm{C}]\ \Gamma(x\bullet z\bullet u)$ defines the partial function Φ such that $a\bullet b\bullet c\in$ scope(Φ) if there is a unique d such that $\mathbf{C}_{x,y,u}[a,b,d]$ holds and $a\bullet c\bullet d\in$ scope(Γ). If the partial function Γ can be defined uniquely and unambiguously by an expression **E**, then we can substitute **E** for Γ in the definition of Φ.

5.1 Formal Model

Solids are modeled in a *design space*. The design space is a normal Euclidean space augmented with an arbitrary but fixed finite number of real-valued *properties*. A solid is a function that maps a list of *parameter values* to a set of points in the design space constituting the volume of the solid. Therefore, each solid in the design space represents a family of objects, each realised by a particular choice of parameter values. According to [37], if a particular vector of values assigned to the variables over which a solid is defined produce the empty set of points, this result is considered to be invalid: that is, to correspond to an impossible object. However, the empty set can also represent a valid solid, for example when a solid results from computing the intersection of two non-intersecting objects. In order to rectify this anomaly, we modify some of the definitions in [37]. Each definition in this section is either the same as the corresponding definition in [37] and is reproduced for completeness, possibly with notational changes, or has been modified to deal with the anomaly. In the latter case, the modification will be explained following the definition.

Definition 5.1: A *design space in m dimensions over r properties* for some integers $m > 0$ and $r \geq 0$ is the set of all subsets of $\mathbf{R}^m \times \mathbf{R}^r$.

Definition 5.2: A *solid* in a design space $\mathcal{D}$ in *n variables* for some $n \geq 0$ is a partial function $\Phi\colon \mathbf{R}^n \rightarrow \mathcal{D}$ such that, if (v,p) and $(v,q) \in \Phi(y)$ for some $y \in \mathbf{R}^n$, then $p=q$. A *variable* of Φ is an integer i such that $1 \leq i \leq n$. Symbolic names may also be used to refer to the variables of a solid.

The intuition behind this definition is that something we normally think of as a solid can be characterised by a set of points in space, where each point has a unique value for each property associated with it.

Definition 5.2 differs from that in [37] by defining a solid to be a partial function instead of a function. This effectively makes the interpretation of the empty set of points unambiguous, allowing us to consider every set of points produced by such a function to correspond to a realisable object, whether or not the set is empty.

This definition also provides parameterisation of objects. The image of a solid corresponds to a family of real-world objects, each created by the partial function from a vector of variable values.

We will continue to refer to a partial function that defines a solid simply as a "function" when the meaning is clear in context.

Definition 5.3: A solid Φ in $\mathcal{D}$ is *valid* iff the image of Φ is not empty.

Note that a solid can generate the empty set of points as one of the members of the family that it defines. The image of a solid may even contain just the empty set, although such solid would be of no practical significance.

Definition 5.4: If Φ and Ψ are solids in a design space $\mathcal{D}$, Φ and Ψ are said to be *equivalent*, denoted $\Phi \equiv \Psi$, iff they have identical images.

This definition recognises that a family of objects in a design space may have more than one representation as a solid (partial function) and is simpler than its counterpart in [37], since there is no longer any need to eliminate invalid objects from the images of Φ and Ψ.

Consider a two dimensional design space over the three properties *temperature*, *colour*, and *material* where *temperature* is determined by *colour* and *material,* and *colour is* determined by *temperature* and *material.* A **Square** solid in this space can obviously be defined by a function Φ of variables $(b,c,l,\alpha,temperature,colour)$ or by another function Ψ of variables $(b,c,d,e,material,colour)$, where b, c, d, e, l and α are the geometric variables shown in Figure 5.1.

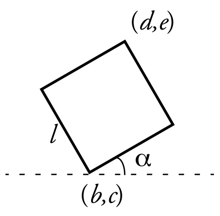


Figure 5.1: A **Square** in the design space.

Example 5.5: To further illustrate the definition, let $\mathcal{D}$ be a design space in 2 dimensions over zero properties: then the partial function **Disk** is a solid in three variables which characterises the family of disks with radius r and centre at (b,c) defined as

$$\textbf{Disk}(b,c,r) = [\ 0 \leq r\ ]\ \{\ (x,y) \mid x, y \in \mathbf{R} \text{ and } (x-b)^2+(y-c)^2 \leq r^2\ \}.$$

Definition 5.6: If $\mathcal{D}$ is a design space and n is a positive integer, an *n-ary operation in* $\mathcal{D}$ is a quintuple $((a_1,\ldots,a_n),(z_1\bullet\ldots\bullet z_n),\mathcal{F},\mathcal{L},\mathbf{C})$ where

- $a_1,\ldots,a_n$ and $z_1\bullet\ldots\bullet z_n$ are sequences of distinct variables such that for each i $(1\leq i \leq n)$, z_i is a sequence of variables and $a_1,\ldots,a_n$ and $z_1\bullet\ldots\bullet z_n$ are disjoint.
- $\mathcal{F}$ is a partial function from $\mathcal{D}^n$ to $\mathcal{D}$ such that $\forall x \in \text{scope}(\mathcal{F})$, if (e,g) and $(e,h) \in \mathcal{F}(x)$, then $g=h$.
- $\mathcal{L}=(\mathbf{L}_1,\ldots,\mathbf{L}_n)$ is a sequence of formulae, called *selectors,* such that $\text{var}(\mathbf{L}_i)=a_i\bullet z_i$. The integer $|z_i|$ is called the *size* of $\mathbf{L}_i$.
- $\mathbf{C}$ is an open formula, called the *constraint* of the operation, such that $\text{var}(\mathbf{C})=z_1\bullet\ldots\bullet z_n$.

This definition differs from that in [37] in that the function $\mathcal{F}$ is now defined as a partial function from $\mathcal{D}^n$ to $\mathcal{D}$, allowing the scope of the solid resulting from the application of an operation to its operand solids to be expressed, among other things (see below), in terms of the scope of $\mathcal{F}$.

Example 5.7: The **Bonding** operation in a 2D design space is defined by $((a_1,a_2),(p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8),\mathbf{fuse},\{\mathbf{edge1},\mathbf{edge2}\},\mathbf{bond})$, where $\mathbf{fuse}(x,y)=[\ x\cap y=l\ ](x\cup y)$, such that l is a line, $\mathbf{bond}=\{p_1=p_7\ ,\ p_2=p_8\ ,\ p_3=p_5\ ,\ p_4=p_6\}$, $\mathbf{edge1}=\mathbf{edge}_{a,b,c,d,e}[a_1,p_1,p_2,p_3,p_4]$, $\mathbf{edge2}=\ \mathbf{edge}_{a,b,c,d,e}[a_2,p_5,p_6,p_7,p_8]$, and $\mathbf{edge}(a,b,c,d,e)$ is true iff (b,c) and (d,e) are points in a, every point on the line between them is in a, every point to the right of this line is in a, and every point to the left of this line is not in a.

Definition 5.8: If Φ is a solid in n variables and **L** is a selector of size k of some operation, where var(**L**)=$a{\cdot}z$ and $|z|=k$, then *an L-interface to* Φ is a partial function $\phi{:}\mathbf{R}^n \rightarrow \mathbf{R}^k$ such that $\phi(u)$=[$u \in$scope(Φ) , $\mathbf{L}_{a,z}[\Phi(u),y)]$] y, and $\forall u,v \in$ scope(Φ), if $\Phi(u)=\Phi(v)$ then $\phi(u)=\phi(v)$.

This definition modifies its counterpart in [37] to account for our revised notion of validity of solids.

Example 5.9: Let **Disk** be as in Example 5.5. Since no two points in a disk satisfy the **edge** selector from Example 5.7, no **edge**-interface to **Disk** can be defined. Therefore, **Disk** cannot be an operand for the **Bonding** operation.

Definition 5.10: Let $\otimes=((a_1,\ldots,a_n),(z_1{\cdot}\ldots{\cdot}z_n),\mathcal{F},\mathcal{L},\mathbf{C})$ be an n-ary operation; where $\mathcal{L}$ = $(\mathbf{L}_1,\ldots,\mathbf{L}_n)$, and for each i ($1 \leq i \leq n$) let Φ_i be a solid in n_i variables, and ϕ_i be an $\mathbf{L}_i$-interface to Φ_i. A solid Ψ in $t=\sum_{i=1}^{n} n_i$ variables, called *the application of* $\otimes$ *to* $\Phi_1,\ldots,\Phi_n$ *via* $\phi_1,\ldots,\phi_n$ is defined as follows. If $y \in \mathbf{R}^t$ denote by y_1 the first n_1 elements of y, denote by y_2 the next n_2 elements of y and so forth, then $\Psi(y)$ is defined as

$$\Psi(y)=[\ \mathbf{C}_{z_1,\ldots,z_n}[\phi_1(y_1),\ldots,\phi_n(y_n)]\ ]\ \mathcal{F}(\Phi_1(y_1),\ldots,\Phi_n(y_n)).$$

This definition modifies its counterpart in [37] by properly accounting for the validity of a generated solid. The solid generated by an operation is defined for a particular selection of values for its variables provided that the constraint of the operation is satisfied by those values, each of the operand solids is defined for that selection of values, and the list of subsets of $\mathcal{D}$ is in the scope of $\mathcal{F}$. This condition, implicit in the definition, is made explicit in following lemma.

Lemma 5.11: If Ψ is the solid defined in Definition 5.10 and $u \in$ scope(Ψ), then $u_i \in$ scope(Φ_i) for each i ($1 \leq i \leq n$), and $(\Phi_1(u_1),\ldots,\Phi_n(u_n)) \in$ scope($\mathcal{F}$).

Proof. Since $u \in \text{scope}(\Psi)$, it has to be in the scope of the partial function defined by the expression $\mathcal{F}(\Phi_1(u_1),\ldots,\Phi_n(u_n))$. Therefore, $u_i \in \text{scope}(\Phi_i)$ for each i $(1 \leq i \leq n)$, and $(\Phi_1(u_1),\ldots,\Phi_n(u_n)) \in \text{scope}(\mathcal{F})$. □

Example 5.12: Assume that we want to attach the two semi ring-shape objects in Figure 5.2 using the **Bonding** operation. Although the constraint of the **Bonding** operation is true, the two objects cannot simultaneously satisfy the **Bonding** condition and be in the scope of **fuse** as defined in Example 5.7, and therefore the application of the **Bonding** operation to these objects will always fail. By failing, we mean that the image of the solid resulting from the application of **Bonding** to these objects is empty.

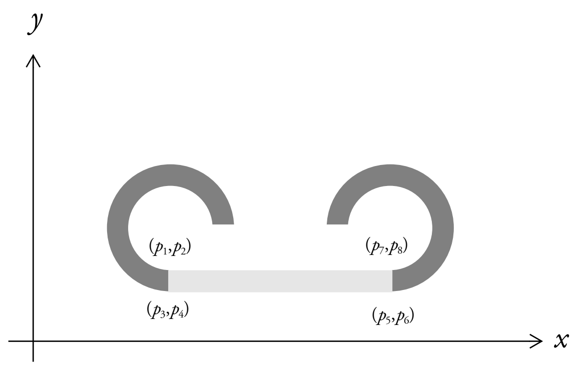


Figure 5.2: An unsuccessful application of **Bonding**.

In the next example, we show how the interfaces in the application of an operation are handled.

Example 5.13: Let **rectangle₁**(x_1,y_1,w_1,h_1)=**Rectangle**(x_1,y_1,w_1,h_1) and **rectangle₂**(x_2,y_2,w_2,h_2)= **Rectangle**(x_2,y_2,w_2,h_2) be two rectangle solids in a 2D design space over zero properties where **Rectangle**(x,y,w,h)=$[\ 0 \leq w\ ,\ 0 \leq h\ ]\ \{\ (a,b)\ \mid\ x \leq a \leq x+w\ ,\ y \leq b \leq y+h\ \}$. Also let **rightSide**$(x,y,w,h)=(x+w,y+h,x+w,y)$ and **leftSide**$(x,y,w,h)=(x,y,x,y+h)$ be **edge**-interfaces to **rectangle₁** and **rectangle₂**, respectively. The application of **Bonding** to **rectangle₁** and **rectangle₂** via **rightSide** and **leftSide** is the solid

$$\Psi(x_1,y_1,w_1,h_1,x_2,y_2,w_2,h_2) = [\ \mathbf{bond}_{p_1,\ldots,p_8}[\ x_1+w_1,\ y_1+h_1,\ x_1+w_1,\ y_1,\ x_2,\ y_2,\ x_2,\ y_2+h_2\ ]\ ]$$

$$\mathbf{fuse}(\mathbf{Rectangle}(x_1,y_1,w_1,h_1),\mathbf{Rectangle}(x_2,y_2,w_2,h_2))$$

$$= [\ x_1+w_1=x_2\ ,\ y_1+h_1=y_2+h_2\ ,\ y_1=y_2\ ]$$

$$\mathbf{fuse}(\mathbf{Rectangle}(x_1,y_1,w_1,h_1),\mathbf{Rectangle}(x_2,y_2,w_2,h_2)).$$

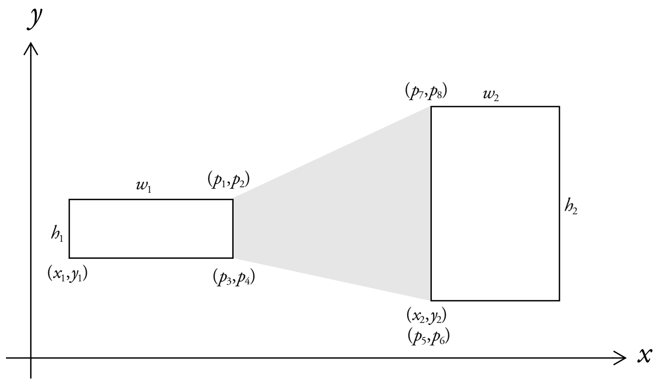


Figure 5.3: Application of **Bonding** to two rectangles

The values required for the constraint of an operation are computed from the operand solids by the interfaces. For instance, in Example 5.23, **edge**-interfaces **rightSide** and **leftSide** are used to compute the bonding edges of the two rectangles used by the operation for fusing the two rectangles together. However, it would be easier if the solids explicitly provided the information required by the constraint of an operation. That is, an interface is a simple mapping of the vector values used for the creation of the solid on to a selection of those values. When the definition of a solid includes all the variables required by an interface of an operation in the list of its input values, the solid is said to expose that interface, captured by the following definition modified from its counterpart in [37] to account properly for validity.

Definition 5.14: If Φ is a solid in n variables, **L** is a selector of size k of some operation, and ϕ is an **L**-interface to Φ, Φ is said to *expose* ϕ iff there exists a sequence $p_1,\ldots,p_k$ of variables of Φ, such that for every $y \in \text{scope}(\Phi)$, for all i $(1 \le i \le k)$ $\phi(y)_i = y_{p_i}$. Each of the variables p_i is said to be *required* by ϕ. The variables $p_1,\ldots,p_k$ are not necessarily distinct, and the sequence $p_1,\ldots,p_k$ is not necessarily unique.

In the remainder of this Chapter, we will further refine these definitions in order to achieve some of the requirements listed in Chapter 1.

5.2 Sample Look

As we discussed in Section 1.4, one of the solid modeler's requirements arising from the LSD debugger is support for a sample look for a solid: that is, a visual representation of some member of the solid's image. Providing a sample look for a predefined component can be achieved by assigning sample values to the variables that define the geometry and appearance of the corresponding solid; for example, size, position, orientation, colour, and possibly material. These values can then be used by the solid modeler to instantiate the designated member of the family of objects corresponding to the solid, capturing the essential visual aspects of the family. The sample look for a family of objects is only a visual aid for tracing the execution of an LSD assembly when it is executed in LSD's debugger environment. Note that the sample values chosen are of no logical significance.

The sample values for the variables of a predefined component could be assigned by the designer of the component and later be overridden by the programmer in order to customise the look. This conveniently solves the sample look problem for predefined explicit components and also gives the programmer the option to change sample looks on a case-by-case basis. However, it is not clear how such sample values should be selected or refined once an operation is applied to operand solids. In other words, although the above mentioned approach is useful for preassembled explicit components, it will not serve for explicit components assembled during execution. For instance, when a variable is instantiated to a value different from its sample value, or when a variable is unified with or constrained to another variable, the predefined sample value must be invalidated. Consider an example where two planar solids are to be attached with a bonding operation as illustrated in Figure 5.4(a). The first operand is a solid in five variables defining a square with fixed size and anchored in place defined as

Base(x_1,y_1,x_2,y_2,a)= [$x_1= .4$, $y_1= .2$, $a= .2$, $x_2=x_1$, $y_2=y_1+a$]

$$\{ (x,y) \mid x,y \in \mathbf{R} , x_1-a \leq x \leq x_1 , y_1 \leq y \leq y_1+a \}.$$

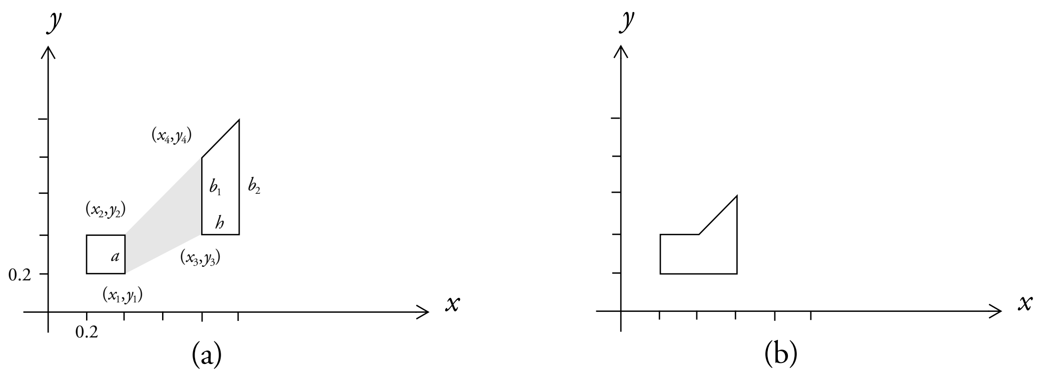


Figure 5.4: Sample looks of two trapezoids.

In this definition, the formula anchors the reference point of the square at (.4, .2) and fixes its size at .2 while the remainder defines a set of points comprising a square. Note that **Base** is a very restricted solid which could be expressed as a constant function. However, we find the above definition useful to describe the sample look computation process.

The second operand is a completely free right trapezoid defined by

RightTrapezoid$(x_3,y_3,x_4,y_4,b_1,b_2,h)$= [$x_3=x_4$, $b_1=y_4-y_3$, $b_1 < b_2 \leq 2b_1$, $0<h$]

$$\{ (x,y) \mid x,y \in \mathbf{R} , x_3 \leq x \leq x_3+h , y_3 \leq y \leq line \}.$$

Note that the scope of **RightTrapezoid** is constrained such that the long base of **RightTrapezoid** cannot be greater than twice the length of the short base. In this definition, *line* is the expression representing the long leg of the trapezoid defined by the line that passes through the points (x_3,y_3+b_1) and (x_3+h,y_3+b_2), that is;

$$line=y_3+b_1+(x-x_3)(b_2-b_1)/h$$

In Figure 5.4(a), **Base** is represented by its actual appearance, so its sample values are x_1= .4, y_1= .2, x_2= .4, y_2= .4, and a= .2, while **RightTrapezoid** is represented by a sample look defined by the sample values x_3= .8, y_3= .4, x_4= .8, y_4= .8, b_1= .4, b_2= .6, and h= .2.

Clearly, after the application of the bonding operation to **Base** and **RightTrapezoid**, the short base of **RightTrapezoid** will be constrained to the size of **Base** which is .2, different from the original sample value of b_1= .4. Consequently the sample value of the long base, b_2, of **RightTrapezoid** is invalidated.

In this section, we consider the problem of computing values with which to generate a sample look for an e-component resulting from the application of an operation to its operands.

One solution is to prohibit the variables of a solid from being dependant, thereby ensuring that changes in one variable will not affect the others. Since the lack of dependencies between the variables of a solid means that the scope of the solid is the cartesian product of subsets of **R**, one for each variable of the solid, the sample value for a particular variable can be chosen simply by selecting a value from the corresponding subset. However, this would put a major limitation on the solid modeler, severely limiting the set of representable objects. Instead, we introduce the notion of factoring to provide for the automatic computation of a sample look. As we will show later, factoring has other implications as well. For example, it can be used to restructure the representation of a solid in LSD so that it exposes different sets of interfaces.

5.2.1 Factoring

A geometric kernel that provides solid modeling for LSD, as part of its functionality, rejects any list of values that would result in creating an impossible object. Therefore, the definition of a solid as a partial function is a reasonable reflection of how geometric kernels operate. Although this responsibility for avoiding the creation of impossible objects could entirely be achieved by a chosen geometric kernel, there are other reasons for knowing the dependencies between the values passed to the kernel:

- computing values for a sample look, as discussed above

- finding parameter sets, and
- finding equivalent solids which expose required interfaces.

The example in the previous section required such information about the dependencies between variables, and expressing the solids using the notation used in Definition 5.10 made that information explicit.

In order to make the information about the dependencies between variables available, we extend the formal model of design space and solids so that the generative aspects of the employed kernel can be separated from the expression of the dependencies between values that are used by the kernel for creating instances of objects. This concept, called *factoring*, is used to express the dependencies between variables of a solid in a wrapper around a solid modeling kernel.

Definition 5.15: If Φ is a solid in n variables over $\mathcal{D}$, a *factoring* of Φ is a quintuple $(x,y,z,\mathbf{C},\Psi)$ where

- x, y, and z are disjoint sequences of distinct variables of lengths n, m and k, respectively, called the *input*, *internal* and *geometric* variables, respectively. (***a***)
- C is a set of formulae, called the *constraint*, such that $\text{var}(\mathbf{C})=x\bullet y\bullet z$. (***b***)
- $\forall s \in z$, s has only one occurrence in **C** and there exists a unique equality $\mathbf{E} \in \mathbf{C}$ such that $\text{var}(\mathbf{E})=\{s\}$ or $\text{var}(\mathbf{E})=\{s,l\}$ and $l \in x \cup y$ (***c***)
- Ψ is a solid in k variables in $\mathcal{D}$. (***d***)
- $\forall u \in \mathbf{R}^n$, if $u \in \text{scope}(\Phi)$, then $\exists\ v \in \mathbf{R}^m$ and $\exists\ w \in \mathbf{R}^k$ such that $\mathbf{C}_{x,y,z}\,[u,v,w]$ is true. (***validity***)
- $\forall u \in \mathbf{R}^n$, $\forall v \in \mathbf{R}^m$, $\forall w \in \mathbf{R}^k$, if $\mathbf{C}_{x,y,z}[u,v,w]$ is true, then $u \in \text{scope}(\Phi)$ iff $w \in \text{scope}(\Psi)$. (***conformity***)
- $\forall u \in \mathbf{R}^n$, $\forall v \in \mathbf{R}^m$, $\forall w \in \mathbf{R}^k$, if $u \in \text{scope}(\Phi)$ and $\mathbf{C}_{x,y,z}[u,v,w]$ is true, then $\Phi(u)=\Psi(w)$. (***identity***)

The factoring is said to be *perfect* if the following condition also holds:

- $\forall u \in \mathbf{R}^n$, $\forall v \in \mathbf{R}^m$, $\forall w \in \mathbf{R}^k$, if $\mathbf{C}_{x,y,z}[u,v,w]$ is true, then $u \in \text{scope}(\Phi)$. (***perfection***)

The intuition behind this definition is that values for the variables of a solid are checked for consistency by the constraint. Once consistency is ensured, Ψ is applied to the values computed by the constraint for its variables to obtain the desired object. An important property of a perfect factoring is that if $\mathbf{C}$ approves of the values provided for the input variables, corresponding values for the geometric variables are guaranteed to be acceptable to the partial function Ψ.

The four conditions *validity*, *perfection*, *identity*, and *conformity* are separated in order to emphasise their roles in the definition of a factoring. The validity condition indicates that if a vector of values is in the scope of a solid Φ, then this vector can be extended by adding values for the internal and geometric variables of the factoring in such a way that the extended vector of values satisfies $\mathbf{C}$. This condition ensures that a factoring does not reject a legitimate list of values. The perfection condition says that $\mathbf{C}$ can be used to determine if a list of values is indeed in the scope of Φ. This is the property of a factoring that will help us to compute a sample look for an e-component. The identity condition ensures that a factoring of a solid corresponds to the same family of objects as Φ. An obvious consequence of the identity condition is that the image of Φ is a subset of the image of Ψ. The conformity condition indicates that once a vector of values satisfies $\mathbf{C}$, the lists of input and geometric values both are in the scopes of Φ and Ψ, respectively, or both are not. The practical implication of this condition is that if a vector of values satisfies $\mathbf{C}$, and Ψ successfully generates an object, the object is in fact in the image of Φ.

A solid can have more than one factoring.

Example 5.16: Consider the solid Ψ generated in Example 5.13, defined by

$$\Psi(x_1,y_1,w_1,h_1,x_2,y_2,w_2,h_2) = [\ x_1+w_1=x_2\ ,\ y_1+h_1=y_2+h_2\ ,\ y_1=y_2\ ]$$

$$\textbf{fuse}(\textbf{Rectangle}(x_1,y_1,w_1,h_1),\textbf{Rectangle}(x_2,y_2,w_2,h_2)).$$

Clearly,

$((x_1,y_1,w_1,h_1,x_2,y_2,w_2,h_2),(),(z_1,\ldots,z_8),\mathrm{C},\mathrm{fuse}(\mathbf{Rectangle}(z_1,z_2,z_3,z_4),\mathbf{Rectangle}(z_5,z_6,z_7,z_8)))$ is a factoring of Ψ where

$\mathrm{C}=\{\, x_1+w_1=x_2\,,\, y_1+h_1=y_2+h_2\,,\, y_1=y_2\,,\, x_1=z_1\,,\, y_1=z_2\,,\, w_1=z_3\,,\, h_1=z_4\,,\, x_2=z_5\,,\, y_2=z_6\,,\, w_2=z_7\,,\, h_2=z_8\,\}$

and the last item in the tuple is an expression defining a partial function which is a solid.

Another factoring of Ψ is

$((x_1,y_1,w_1,h_1,x_2,y_2,w_2,h_2),(),(z_1,\ldots,z_8),\mathrm{X},\mathrm{fuse}(\mathbf{Rectangle}(z_1,z_2,z_3,z_4),\mathbf{Rectangle}(z_5,z_6,z_7,z_8)))$
where

$$\begin{aligned}\mathrm{X} = \{\, & x_1+w_1=x_2\,,\, y_1+h_1=y_2+h_2\,,\, y_1=y_2\,,\\ & 0\le w_1\,,\, 0\le h_1\,,\, 0\le w_2\,,\, 0\le h_2\,,\\ & x_1=z_1\,,\, y_1=z_2\,,\, w_1=z_3\,,\, h_1=z_4\,,\, x_2=z_5\,,\, y_2=z_6\,,\, w_2=z_7\,,\, h_2=z_8\,\}.\end{aligned}$$

Note that the latter factoring includes some formulae that guarantee that (x_1,y_1,w_1,h_1), $(x_2,y_2,w_2,h_2) \in \mathrm{scope}(\mathbf{Rectangle})$, making this a perfect factoring.

We now prove two useful results that relate factorings of solids and the application of operations.

Lemma 5.17: If $\mathcal{D}$ is a design space, $\otimes=((a_1,\ldots,a_n),(p_1\cdot\ldots\cdot p_n),\mathcal{F},\mathcal{L},\mathrm{C})$ is an n-ary operation in $\mathcal{D}$, $\Phi_1,\ldots,\Phi_n$ are solids in $\mathcal{D}$, Ψ is the solid resulting from the application of $\otimes$ to $\Phi_1,\ldots,\Phi_n$ via interfaces $\phi_1,\ldots,\phi_n$, and $(x_i,y_i,z_i,\mathrm{C}_i,\Psi_i)$ is a factoring of Φ_i for each i ($1\le i\le n$), then $(x,y,z,\mathrm{X},\Delta)$ is a factoring of Ψ where x is $x_1\bullet\ldots\bullet x_n$, y is $y_1\bullet\ldots\bullet y_n$, z is $z_1\bullet\ldots\bullet z_n$, and

- $\mathrm{X} = \mathrm{C}_{p_1,\ldots,p_n}[\phi_1(x_1),\ldots,\phi_n(x_n)] \cup \mathrm{C}_1 \cup \ldots \cup \mathrm{C}_n$, and
- $\Delta(z) = \mathcal{F}(\Psi_1(z_1),\ldots,\Psi_n(z_n))$

Proof. We assume that for all i and j, where $j \neq i$, $\mathrm{var}(\mathrm{C}_i)$ and $\mathrm{var}(\mathrm{C}_j)$ are disjoint.

Let $t=|x|$, $m=|y|$, and $k=|z|$. If $u \in \mathbf{R}^t$, then u can be written as $u_1\cdot\ldots\cdot u_n$ where $|u_i|=|x_i|$ for each i ($1 \le i \le n$). Elements of $\mathbf{R}^m$ and $\mathbf{R}^k$ can be similarly decomposed.

(*a*): Since for each i $(1 \leq i \leq n)$, $(x_i, y_i, z_i, \mathbf{C}_i, \Psi_i)$ is a factoring of Φ_i, x_i, y_i, and z_i are disjoint sequences of distinct variables, therefore, because of the above assumption, x, y, and z are disjoint sequences of distinct variables.

(*b*): Clearly var(X)=$x \bullet y \bullet z$.

(*c*): Suppose $s \in z$, then $s \in z_i$ for some i $(1 \leq i \leq n)$. Because of the definition of factoring $s \notin x_i$ and because of our assumption that var($\mathbf{C}_i$) and var($\mathbf{C}_j$) are disjoint for all $j \neq i$, $s \notin x_j$. Since the only variables that occur in $\mathbf{C}_{p_1,\ldots,p_n}[\phi_1(x_1),\ldots,\phi_n(x_n)]$ are those in $x_1,\ldots,x_n$, s does not occur in $\mathbf{C}_{p_1,\ldots,p_n}[\phi_1(x_1),\ldots,\phi_n(x_n)]$. Therefore, because of the definition of factoring, each $s \in z$ occurs in a unique equality $\mathbf{E} \in \mathbf{X}$ such that var($\mathbf{E}$)=$\{s\}$ or var($\mathbf{E}$)=$\{s,l\}$ and $l \in x \cup y$.

(*d*): Δ is a partial function from $\mathbf{R}^k$ to $\mathcal{D}$. Suppose that for some $w \in$scope(Δ), $(e,g),(e,h) \in \Delta(w)$. Then $w_i \in$ scope(Ψ_i) for each i $(1 \leq i \leq n)$, $(\Psi_1(w_1),\ldots,\Psi_n(w_n)) \in$scope($\mathcal{F}$) and (e,g), $(e,h) \in \mathcal{F}(\Psi_1(w_1),\ldots,\Psi_n(w_n))$ so by the condition on $\mathcal{F}$ imposed by the definition of operation, $g=h$. Hence, Δ is a solid in $\mathcal{D}$.

It remains to show that the validity, conformity and identity conditions are satisfied.

(***Validity***): Suppose that $u \in \mathbf{R}^t$ and $u \in$ scope(Ψ), then according to Definition 5.10, $\mathbf{C}_{p_1,\ldots,p_n}[\phi_1(u_1),\ldots,\phi_n(u_n)]$ is true, and according to Lemma 5.11, $u_i \in$ scope(Φ_i) for each i $(1 \leq i \leq n)$. Since $(x_i, y_i, z_i, \mathbf{C}_i, \Psi_i)$ is a factoring of Φ_i, and $u_i \in$ scope(Φ_i), by the validity condition we can conclude that there exist v_i and w_i such that $(\mathbf{C}_i)_{x_i, y_i, z_i}[u_i, v_i, w_i]$ is true for each i $(1 \leq i \leq n)$. Therefore, $\mathbf{X}_{x,y,z}[u,v,w]$ is true where v is $v_1 \bullet \ldots \bullet v_n$, and w is $w_1 \bullet \ldots \bullet w_n$, proving that the validity condition holds.

(***Conformity***): Suppose that $u \in \mathbf{R}^t$, $v \in \mathbf{R}^m$, $w \in \mathbf{R}^k$, and $\mathbf{X}_{x,y,z}[u,v,w]$ is true.

If $u \in$ scope(Ψ), then according to Lemma 5.11, $u_i \in$ scope(Φ_i) for each i $(1 \leq i \leq n)$, and $(\Phi_1(u_1),\ldots,\Phi_n(u_n)) \in$ scope($\mathcal{F}$). Since $\mathbf{X}_{x,y,z}[u,v,w]$ is true, then $(\mathbf{C}_i)_{x_i, y_i, z_i}[u_i, v_i, w_i]$ is true for each i $(1 \leq i \leq n)$, and since $u_i \in$ scope(Φ_i), by the conformity condition, $w_i \in$ scope(Ψ_i) and by the identity condition $\Phi_i(u_i)=\Psi_i(w_i)$, for each i $(1 \leq i \leq n)$. Since $w_i \in$ scope(Ψ_i) for each i $(1 \leq i \leq n)$, and $(\Psi_1(w_1),\ldots,\Psi_n(w_n)) \in$ scope($\mathcal{F}$), $w \in$ scope(Δ).

If $w \in \text{scope}(\Delta)$, then $w_i \in \text{scope}(\Psi_i)$ for each i $(1 \leq i \leq n)$, and $(\Psi_1(w_1),\ldots,\Psi_n(w_n)) \in \text{scope}(\mathcal{F})$. Since $(\mathbf{C}_i)_{x_i,y_i,z_i}[u_i,v_i,w_i]$ is true and $w_i \in \text{scope}(\Psi_i)$ for each i $(1 \leq i \leq n)$, by the conformity condition on the factoring $(x_i,y_i,z_i,\mathbf{C}_i,\Psi_i)$ of Φ_i, $u_i \in \text{scope}(\Phi_i)$, and by the identity condition $\Phi_i(u_i)=\Psi_i(w_i)$, for each i $(1 \leq i \leq n)$. Therefore, $(\Phi_1(u_1),\ldots,\Phi_n(u_n)) \in \text{scope}(\mathcal{F})$. Since $u_i \in \text{scope}(\Phi_i)$ for each i $(1 \leq i \leq n)$, and $(\Phi_1(u_1),\ldots,\Phi_n(u_n)) \in \text{scope}(\mathcal{F})$, u is in the scope of the partial function defined by the expression $\mathcal{F}(\Phi_1(x_1),\ldots,\Phi_n(x_n))$. Since $\mathbf{C}_{p_1,\ldots,p_n}[\phi_1(u_1),\ldots,\phi_n(u_n)]$ is true, and u is in the scope of the partial function defined by the expression $\mathcal{F}(\Phi_1(x_1),\ldots,\Phi_n(x_n))$, $u \in \text{scope}(\Psi)$, establishing the conformity condition.

(***Identity***): Suppose that $u \in \mathbf{R}^t$, $v \in \mathbf{R}^m$, $w \in \mathbf{R}^k$, $u \in \text{scope}(\Psi)$ and $\mathrm{X}_{x,y,z}[u,v,w]$ is true. Since $u \in \text{scope}(\Psi)$, then u is in the scope of the partial function defined by the expression $\mathcal{F}(\Phi_1(u_1),\ldots,\Phi_n(u_n))$. Therefore, $u_i \in \text{scope}(\Phi_i)$ for each i $(1 \leq i \leq n)$. Since $\mathrm{X}_{x \bullet y \bullet z}[u \bullet v \bullet w]$ is true, $(\mathbf{C}_i)_{x_i,y_i,z_i}[u_i,v_i,w_i]$ is true for each i $(1 \leq i \leq n)$, and since $(x_i,y_i,z_i,\mathbf{C}_i,\Psi_i)$ is a factoring of Φ_i, and $u_i \in \text{scope}(\Phi_i)$, by the conformity condition $w_i \in \text{scope}(\Psi_i)$ and by the identity condition $\Phi_i(u_i)=\Psi_i(w_i)$ for each i $(1 \leq i \leq n)$. Therefore,

$$\begin{aligned}\Psi(u) &= \mathcal{F}(\Phi_1(u_1),\ldots,\Phi_n(u_n))\\ &= \mathcal{F}(\Psi_1(w_1),\ldots,\Psi_n(w_n))\\ &= \Delta(w)\end{aligned}$$

establishing the identity condition. □

Lemma 5.18: If in Lemma 5.17, $(x_i,y_i,z_i,\mathbf{C}_i,\Psi_i)$ is a perfect factoring of Φ_i for each i $(1 \leq i \leq n)$, and $\mathbf{C}_{p_1,\ldots,p_n}[\phi_1(u_1),\ldots,\phi_n(u_n)]$ implies that $(\Phi_1(u_1),\ldots,\Phi_n(u_n)) \in \text{scope}(\mathcal{F})$, then $(x,y,z,\mathrm{X},\Delta)$ is a perfect factoring of Ψ.

Proof. Since $\mathrm{X}_{x,y,z}[u,v,w]$ is true $(\mathbf{C}_i)_{x_i,y_i,z_i}[u_i,v_i,w_i]$ is true for each i $(1\leq i \leq n)$, so for each i, $u_i \in \text{scope}(\Phi_i)$ since $(x_i,y_i,z_i,\mathbf{C}_i,\Psi_i)$ is a perfect factoring of Φ_i, and $\Phi_i(u_i)=\Psi_i(w_i)$ by the identity condition. Also, since $\mathrm{X}_{x,y,z}[u,v,w]$ is true, $\mathbf{C}_{p_1,\ldots,p_n}[\phi_1(u_1),\ldots,\phi_n(u_n)]$ is true. Therefore,

$(\Phi_1(u_1),\ldots,\Phi_n(u_n)) \in \text{scope}(\mathcal{F})$, so that $(\Psi_1(w_1),\ldots,\Psi_n(w_n)) \in \text{scope}(\mathcal{F})$. Hence $w \in \text{scope}(\Delta)$ and since $\mathrm{X}_{x,y,z}[u,v,w]$ is true, $u \in \text{scope}(\Psi)$ by the conformity condition. □

In practice , the solid inside a factoring could be a function call to a geometric kernel for creating an object. The constraint in a factoring, realised as a wrapper around the chosen kernel, reveals some of the dependencies between the variables of the factored solid. If a factoring is perfect, the constraint of the factoring adds further restrictions to the scope of the solid inside the factoring such that once a vector of values is consistent in the constraint, then that vector is guaranteed to produce a valid object, facilitating the automatic generation of sample values for the sample look of a solid.

In Example 5.16, we showed how a factoring can be constructed for a solid defined by a partial function. Now we will show how, given a factoring for a solid, a partial function defining the factored solid can be constructed. Let $(x,y,z,\mathbf{C},\Psi)$ be a factoring of Φ, and $k=|z|$. Then according to the definition of a factoring, for each i $(1 \leq i \leq k)$, z_i occurs in a unique equality $\mathbf{E}_i$ in $\mathbf{C}$. If we rearrange $\mathbf{E}_i$ into the form $e_i=z_i$ where e_i is an expression, then Φ can be expressed as $\Phi(x)= [\mathbf{C}^z]\ \Psi(e_1,\ldots,e_k)$.

The sample values for the variables of predefined components are chosen by the designer as discussed earlier, and would be a solution to the constraint of any perfect factoring of the corresponding solid. When an operation is applied to factorings of its operand solids, a new set of sample values must be computed for the sample look of the new solid, represented by the factoring computed as shown in the above lemmas, such that all the constraints in the factoring are satisfied. Note that the constraints may originate from the factorings of operand solids or from the constraint of the operation. Once a list of sample values are computed, they are passed to a renderer for drawing. The sample value generation process will have no effect on the internal state of the geometric kernel and does not force the kernel to actually instantiate an object. Sample values are used only by a renderer which could be completely independent from the kernel.

Clearly, an algorithm that determines new sample values for the variables will operate according to certain criteria. For example, one criterion could be to keep the new values for the free variables as close as possible to their original values. Hence, the problem of finding a sample look is reduced to a constraint satisfaction problem.

An interesting property of a factoring with respect to sample values is that only the geometric variables need be assigned sample values, as they are the only values used by geometric kernels.

Now, using the notion of factoring, we revisit the previous example of bonding **Base** and **RightTrapezoid**.

Base can be factored to $(u_1,v_1,w_1,\mathrm{C}_1,\Phi_1)$ where u_1 is (x_1,y_1,x_2,y_2,a), v_1 is (), and w_1 is (z_1,z_2,z_3) and

$\mathrm{C}_1 = \{\ x_2{=}x_1\ ,\ y_2{=}y_1{+}a\ ,\ x_1{=}\ .4\ ,\ y_1{=}\ .2\ ,\ a = .2\ ,\ z_1{=}x_1\ ,\ z_2{=}y_1\ ,\ z_3{=}a\ \}$ and

$\Phi_1(w_1){=}\ [\ \mathrm{C}_1\ ]\ \{\ (x,y)\mid x,y\ \in \mathbf{R}\ ,\ z_1{-}z_3 \le x \le z_1\ ,\ z_2 \le y \le z_2{+}z_3\ \}$. The sample values for **Base** are given as $z_1{=}\ .4$, $z_2{=}\ .2$, and $z_3{=}\ .2$.

RightTrapezoid is factored to $(u_2,v_2,w_2,\mathrm{C}_2,\Phi_2)$ where u_2 is $(x_3,y_3,x_4,y_4,b_1,b_2,h)$, v_2 is (), and w_2 is (z_4,z_5,z_6,z_7,z_8) and

$\mathrm{C}_2 = \{\ x_3{=}x_4\ ,\ y_4{-}y_3{=}b_1\ ,\ b_1 < b_2 \le 2b_1\ ,\ 0 < h\ ,\ z_4{=}x_3\ ,\ z_5{=}y_3\ ,\ z_6{=}b_1\ ,\ z_7{=}b_2\ ,\ z_8{=}h\ \}$ and

$\Phi_2(w_2)\ =\ [\ \mathrm{C}_2]\ \{\ (x,y)\mid x,y\ \in \mathbf{R}\ ,\ z_4 \le x \le z_4{+}z_8\ ,\ z_5 \le y \le z_5{+}z_6{+}(x{-}z_4)(z_7{-}z_6)/z_8\ \}$ and the sample values are $z_4{=}\ .8$, $z_5{=}\ .4$, $z_6{=}\ .4$, $z_7{=}\ .6$, and $z_8{=}\ .2$.

Recall that the **Bonding** operation in a 2D design space is defined by $((a_1,a_2),(p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8),$**fuse**$,\{$**edge1**$,$**edge2**$\},$**bond**$)$, where **fuse**$(x,y){=}[\ x{\cap}y{=}l\ ](x{\cup}y)$, such that l is a line, **bond**$=\{p_1{=}p_7\ ,\ p_2{=}p_8\ ,\ p_3{=}p_5\ ,\ p_4{=}p_6\}$, **edge1**=**edge**$_{a,b,c,d,e}[a_1,p_1,p_2,p_3,p_4]$, **edge2**= **edge**$_{a,b,c,d,e}[a_2,p_5,p_6,p_7,p_8]$, and **edge**$(a,b,c,d,e)$ is true iff (b,c) and (d,e) are points in a, every point on the line between them is in a, every point to the right of this line is in a, and every point to the left of this line is not in a.

Also let **squareSide**$(x_1,y_1,x_2,y_2,a)=(x_2,y_2,x_1,y_1)$ and **trapSide**$(x_3,y_3,x_4,y_4,b_1,b_2,h)=(x_3,y_3,x_4,y_4)$ be **edge**-interfaces to **Base** and **RightTrapezoid**, respectively. Note that **Base** exposes **squareSide** and **RightTrapezoid** exposes **trapSide**. The solid resulting from execution of the **Bonding** operation can be factored to (x,y,z,X,Ψ) where $x=u_1\bullet u_2$, $y=v_1\bullet v_2$, $z=w_1\bullet w_2$ and

$$\begin{aligned}\mathrm{X} &= \mathrm{C}_1 \cup \mathrm{C}_2 \cup \mathbf{bond}_{p_1,\ldots,p_8}[x_2,y_2,x_1,y_1,x_3,y_3,x_4,y_4]\\ &= \{\, x_2=x_1 \,,\, y_2=y_1+a \,,\, x_1= .4 \,,\, y_1= .2 \,,\, a= .2 \,,\, z_1=x_1 \,,\, z_2=y_1 \,,\, z_3=a,\\ &\quad x_3=x_4 \,,\, y_4-y_3=b_1 \,,\, b_1 < b_2 \le 2b_1 \,,\, 0 < h \,,\, z_4=x_3 \,,\, z_5=y_3 \,,\, z_6=b_1 \,,\, z_7=b_2 \,,\, z_8=h,\\ &\quad x_2=x_4 \,,\, y_2=y_4 \,,\, x_1=x_3 \,,\, y_1=y_3 \,\}\\ &= \{\, x_1=x_2=x_3=x_4= .4 \,,\, y_4=y_2=y_1+a \,,\, y_3=y_1= .2 \,,\, a= .2 \,,\, y_4-y_3=b_1 \,,\, b_1 < b_2 \le 2b_1 \,,\, 0 < h \,,\\ &\quad z_1=x_1 \,,\, z_2=y_1 \,,\, z_3=a \,,\, z_4=x_3 \,,\, z_5=y_3 \,,\, z_6=b_1 \,,\, z_7=b_2 \,,\, z_8=h \,\} \text{ and}\end{aligned}$$

$$\Psi(z)= [\,\mathrm{X}\,]\ \mathbf{fuse}(\Phi_1(z_1,z_2,z_3,z_4),\Phi_2(z_5,z_6,z_7,z_8)).$$

Sample values for some of the geometric variables are fixed, that is $z_1= .4$, $z_2= .2$, $z_3= .2$, $z_4= .4$, $z_5= .2$, and $z_6= .2$. The constraints involving the remaining geometric variables simplify to two equalities $z_7=b_2$, and $z_8=h$, and three inequalities $.2 < b_2 \le .4$, $0 < h$. A new sample value for z_7 can be assigned depending on how the constraint solver operates. For example, if the criterion for the assignment of new sample values is to keep them as close as possible to the sample values they replace, the new sample value for z_7 would be .4. However, in the absence of this criterion any value for z_7 in the solution of X would do. Lastly, since neither z_8 nor h depends on any other variable in **X**, the previous sample value of z_8 could be preserved. In summary, the sample values for the sample look of the new solid are $z_1= .4$, $z_2= .2$, $z_3= .2$, $z_4= .4$, $z_5= .2$, $z_6= .2$, $z_7= .4$, and $z_8= .2$, as illustrated in Figure 5.4 (b).

Example 5.19: To further illustrate the computation of sample values for a new solid, let (u,v,w,C,Φ) be a factoring of the solid **Disk** in Example 5.5 where u is (x_1,y_1,z_1), v is (), w is (b_1,c_1,r_1),

$\mathbf{C} = \{ 0 \le z_1 , b_1 = x_1 , c_1 = y_1 , r_1 = z_1 \}$, and

$\Phi(b_1,c_1,r_1) = [0 \le r_1] \{ (x,y) \mid x,y \in \mathbf{R} \text{ and } (x-b_1)^2 + (y-c_1)^2 \le r_1^2 \}$.

Also let the sample values for the sample look of **Disk** be $b_1=0$, $c_1=0$, and $r_1=1$. An operation named **Punch** that punches a hole represented by one **Disk** at the centre of another **Disk** is defined by $((a_1,a_2),(x_1,y_1,z_1,x_2,y_2,z_2),p-q, \{\mathbf{centre1},\mathbf{centre2}\},\mathbf{align})$ where $\mathbf{align} = \{ x_1=x_2 , y_1=y_2 , z_2<z_1 \}$ and

$\mathbf{centre1}(a_1,x_1,y_1,z_1) = \mathbf{centre}_{a,b,c,r}[a_1,x_1,y_1,z_1]$,

$\mathbf{centre2}(a_2,x_2,y_2,z_2) = \mathbf{centre}_{a,b,c,r}[a_2,x_2,y_2,z_2]$,

and $\mathbf{centre}(a,b,c,r)$ is true iff a is a **Disk** with centre (b,c) and radius r. The solid resulting from the application of **Punch** to two disks **Disk$_1$** and **Disk$_2$** with similar factorings $((x_1,y_1,z_1),(),(b_1,c_1,r_1),\mathbf{C},\Phi)$ and $((x_2,y_2,z_2),(),(b_2,c_2,r_2),\mathbf{C},\Phi)$ and sample values $b_1=0$, $c_1=0$, $r_1=1$ and $b_2=0$, $c_2=0$, $r_2=1$, respectively, called **Ring**, has a factoring $((x_1,y_1,z_1,x_2,y_2,z_2),(),(b_1,c_1,r_1,b_2,c_2,r_2),\mathbf{X},\Psi)$ where

$\mathbf{X} = \{ 0 \le z_1 , b_1=x_1 , c_1=y_1 , r_1=z_1 , 0 \le z_2 , b_2=x_2 , c_2=y_2 , r_2=z_2 , x_1=x_2 , y_1=y_2 , z_2<z_1 \}$ and

$\Psi(b_1,c_1,r_1,b_2,c_2,r_2) = [\, \mathbf{X} \,]\ \Phi(b_1,c_1,r_1) - \Phi(b_2,c_2,r_2)$.

Since none of the variables in **X** are bound to constants, no immediate conclusion about the new sample values for the variables can be made. Fortunately in this case, most of the original sample values for the variables are consistent with the constraints in **C** which yields the following new sample values $b_1=0$, $c_1=0$, $b_2=0$, and $c_2=0$. The original sample values for r_1 and r_2, however, are not consistent with the inequality $z_2<z_1$. Again, the constraint solver's criteria will determine how two values for r_1 and r_2 are picked to satisfy the constraints. A possible choice is $r_1=1$ and $r_2= .5$.

Finding new sample values for the sample look of a solid resulting from the application of an operation is performed only when an LSD program executed in **Debug** mode requires rendering of e-components for the animation of an execution of a program, and for displaying components resulting from geometric computations. In **Run** mode, sample look computation is required only at the very last step of the execution when any resulting free e-components need to be rendered.

5.2.2 Relaxed Factoring

The intuition underlying Definition 5.15 is that if all the conditions that constrain the variables of a solid are extracted from the geometric kernel, a list of sample values computed by a constraint solver working in conjunction with LSD is guaranteed to generate a member of the image of the solid, that is, an object that can be constructed in the design space. This is captured in Definition 5.15 by the perfection condition. However, in practice, the perfection condition is too strong to be easily attainable. For example, the bonding operation used for attaching right trapezoid and box in our earlier example, is generic enough to be used to attach any two solids if they expose the necessary interfaces. But the condition in the bonding operation does not check that operand solids do not overlap somewhere other than line along which they are bonded. Hence, although the two solids could be bonded, the geometric kernel may still reject the resulting object as illustrated in Example 5.12.

A factoring which is not perfect, called a *relaxed factoring*, is still useful in practice. For example, the solver and the geometric kernel might cooperate to find a list of sample values on which they both can agree. When a list of values consistent in $\mathbf{C}$ is not in the scope of Ψ, the geometric kernel could report a failure back to the solver and request a different list of values. Such a dialogue between the geometric kernel and the constraint solver would ensure that the selected sample values are both consistent in $\mathbf{C}$ and in the scope of Ψ. Several negotiations between the solver and the kernel might be required.

5.2.3 Sample Value Assignment Criteria

As we have mentioned, different criteria may be used for the assignment of sample values to the uninstantiated variables in the constraint store used for the computation of sample values. The aim of a sample value assignment criterion is to ensure that values selected for the uninstantiated variables in the constraint store are not only consistent, but also satisfy some additional conditions. This is to ensure that the sample value assignment process can account for certain cognitive factors that can be formulated in the assignment algorithm. For example, the sample value assignment algorithm may introduce a new constraint for the assignment of values to a cylinder to make sure that the ratio of radius to height of a visual representation of a free cylinder are within some desired range. This may be achieved by a *weak constraint* defined by the designer of a predefined component for the algorithm employed for the computation of its sample values. However, this should not in any way change the behaviour of an execution involving the component. Such weak constraints are used only by the constraint solver computing sample values and do not imply any additional restrictions on the solid itself. This mechanism gives the designer of components more influence over how the sample value assignment algorithm works.

In practice, the sample value assignment algorithm is more critical than it may appear at first. While the algorithm's aim should be to capture and reveal the essential and common visual aspects of a family of solids, this should not be done arbitrarily. The choice of sample values and the transition from one set of values to another as the result of the application of an operation should be made in such a way that the resulting visualisations do not imply any extra semantics beyond those inherent in the program. In other words, the algorithm should not mislead the user.

5.3 Transformation and High-level Parameterisation

In a solid modeler kernel, operators that manipulate the configuration of a solid typically accept values that define the extent of the transformation. For example, a scaling factor, a translation vector, or a list of rotational angles for scaling, translating or rotating an object, respectively.

The obvious way to implement a transformation in LSD is by using an operation in which the function $\mathcal{F}$ is unary. Considering the domain of $\mathcal{F}$ in Definition 5.2, it is clear that such a transformation operation could not be parameterised. To remedy this problem one might instead use a transformation operation in which the extent of the transformation is extracted from a second operand solid, designed specifically for this purpose. The parameterisation of such special purpose solids would then provide the means for parameterising transformation operations. For example, a translation operation could be implemented as an arrow-shape solid, the length of which specifies the extent of the translation. Another possible implementation of transformations in LSD could rely on each solid exposing a set of transformation interfaces, one for each possible transformation. The extent of a transformation could then be set by appropriately constraining variables of the solid. However, this method would require every solid to expose a set of interfaces for transformational operations.

In Chapter 1, we listed high-level parameterisation of solids as a requirement for an LSD solid modeler: in particular, we listed iterations in which an operation generates a stack of copies of a solid according to an integer parameter. An iteration programmed in LSD may require several designs, so could be quite bulky. Although in some cases this may be important in order to illustrate the details of an iterative design, in other cases where no important design details are involved, it may become intrusive, in which case a parametrised operation analogous to a counted loop would be useful.

The following definition takes both the above issues into account by extending the definition of operation in a design space to allow for parameters in addition to the solid operands.

Definition 5.20: If $\mathcal{D}$ is a design space and n is a positive integer, an *n-ary operation in* $\mathcal{D}$ is a quintuple $(((a_1,\ldots,a_n),q),(z_1\cdot\ldots\cdot z_n\cdot p),\mathcal{F},\mathcal{L},\mathbf{C})$ where

- $a_1,\ldots,a_n$, $z_1,\ldots,z_n$ and $\mathcal{L}$ are as in Definition 5.6
- p and q are disjoint sequences of distinct variables such that $|q| \le |p|$ and $p\cdot q$ is disjoint from $(a_1,\ldots,a_n)\cdot z_1\cdot\ldots\cdot z_n$. Variables in p are called the *extra variables* of the operation.

- $\mathcal{F}$ is a partial function from $\mathcal{D}^n \times \mathbf{R}^{|q|}$ to $\mathcal{D}$ such that $\forall x \in \text{scope}(\mathcal{F})$, if (e,g) and $(e,h) \in \mathcal{F}(x)$, then $g=h$.
- **C** is an open formula, called the *constraint* of the operation, such that $\text{var}(\mathbf{C})=z_1 \cdot \ldots \cdot z_n \cdot p \cdot q$, and for each $s \in q$, there exists a unique equality $\mathbf{E} \in \mathbf{C}$ such that $\text{var}(\mathbf{E})=\{s\}$ or $\text{var}(\mathbf{E})=\{s,l\}$ and $l \in p$.

This definition formalises the concept of parameterised operations provided by some geometric kernels, for example, for translating, scaling, rotating, reflecting, skewing, and repeating objects. Clearly, we now need to update the definitions and Lemmas that rely on the definition of an operation. For example, Definition 5.10 and Lemma 5.17 and Lemma 5.18.

Definition 5.21: Let $\otimes = (((a_1,\ldots,a_n),q),(z_1 \cdot \ldots \cdot z_n \cdot p),\mathcal{F},\mathcal{L},\mathbf{C})$ be an n-ary operation; where $\mathcal{L} = (\mathbf{L}_1,\ldots,\mathbf{L}_n)$, and for each i $(1 \leq i \leq n)$ let Φ_i be a solid in n_i variables, and ϕ_i be an $\mathbf{L}_i$-interface to Φ_i. A solid Ψ in $t = \sum_{i=1}^{n} n_i + |p|$ variables, called *the application of* $\otimes$ *to* $\Phi_1,\ldots,\Phi_n$ *via* $\phi_1,\ldots,\phi_n$ is defined as follows. Suppose $d \in \mathbf{R}^{|q|}$ and $y \in \mathbf{R}^t$. If we denote by y_1 the first n_1 elements of y, denote by y_2 the next n_2 elements of y and so forth, then $\Psi(y)$ is defined as

$$\Psi(y) = [\mathbf{C}_{z_1,\ldots,z_n,p,q}[\phi_1(y_1),\ldots,\, \phi_n(y_n),y_{*|p|},d]]\ \mathcal{F}((\Phi_1(y_1),\ldots,\Phi_n(y_n)),d).$$

Lemma 5.22: If Ψ is the solid defined in Definition 5.21 and $u \in \text{scope}(\Psi)$, then $u_i \in \text{scope}(\Phi_i)$ for each i $(1 \leq i \leq n)$, and $((\Phi_1(u_1),\ldots,\Phi_n(u_n)),c) \in \text{scope}(\mathcal{F})$ for some c.

Proof. If $u \in \text{scope}(\Psi)$, then according to Definition 5.21 $\mathbf{C}_{z_1,\ldots,z_n,p,q}[\phi_1(u_1),\ldots,\phi_n(u_n),u_{*|p|},c]$ is true for some c, and $u \cdot c$ is in the scope of the partial function defined by the expression $\mathcal{F}((\Phi_1(u_1),\ldots,\Phi_n(u_n)),c)$. Therefore, $u_i \in \text{scope}(\Phi_i)$ for each i $(1 \leq i \leq n)$, and $(\Phi_1(u_1),\ldots,\Phi_n(u_n)),c) \in \text{scope}(\mathcal{F})$. □

Example 5.23: The **Scale** operation in a 2D design space $\mathcal{D}$ can be defined by $(((o),(d_1,d_2)),(a,b),\mathcal{S},\{\mathbf{operand}\},\mathbf{C})$ where $\mathbf{operand}(o)=\mathbf{true}$, $\mathbf{C}=\{\ 0<a\ ,\ 0<b\ ,\ a=d_1\ ,b=d_2\ \}$, and $\mathcal{S}$ is a function from $\mathcal{D} \times \mathbf{R}^2 \rightarrow \mathcal{D}$ which given an object and a pair of scalar values, resizes the

object in each dimension to the extent of the corresponding scalar value. An application of this operation to **Rectangle** from Example 5.13 is defined by

$$\Psi(x,y,w,h,a,b) = [\ 0<a\ ,\ 0<b\ ,\ a=d_1\ ,\ b=d_2\ ,\ 0\leq w\ ,\ 0\leq h\ ]\ \mathcal{S}(\textbf{Rectangle}(x,y,w,h),(d_1,d_2)).$$

Figure 5.5 shows the application of this operation to **Rectangle**(x,y,w,h) with scalar values (1.5,2.0). The rectangle drawn in dotted lines represents the solid obtained from the application of the scaling operation to the rectangle drawn in solid lines.

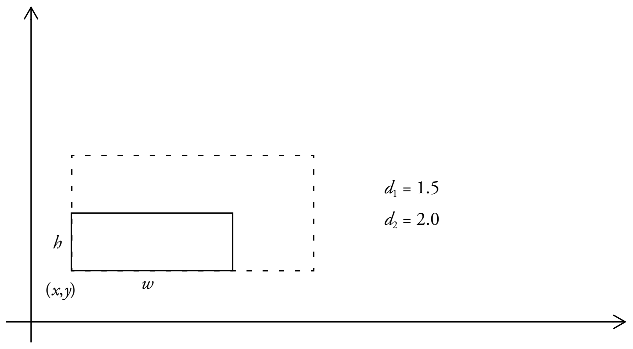


Figure 5.5: A scaling operation.

Lemma 5.24: If $\mathcal{D}$ is a design space, $\otimes = (((a_1,\ldots,a_n),q),(p_1\cdot\ldots\cdot p_n\cdot p),\mathcal{F},\mathcal{L},\mathrm{C})$ is an n-ary operation in $\mathcal{D}$, $\Phi_1,\ldots,\Phi_n$ are solids in $\mathcal{D}$, Ψ is the solid resulting from the application of $\otimes$ to $\Phi_1,\ldots,\Phi_n$ via interfaces $\phi_1,\ldots,\phi_n$, and $(x_i,y_i,z_i,\mathrm{C}_i,\Psi_i)$ is a factoring of Φ_i for each i $(1 \leq i \leq n)$, then $(x,y,z,\mathrm{X},\Delta)$ is a factoring of Ψ where x is $x_1\bullet\ldots\bullet x_n\cdot p$, y is $y_1\bullet\ldots\bullet y_n$, z is $z_1\bullet\ldots\bullet z_n\cdot q$, and

- $\mathrm{X} = \mathrm{C}_{p_1,\ldots,p_n}[\phi_1(x_1),\ldots,\phi_n(x_n)] \cup \mathrm{C}_1 \cup \ldots \cup \mathrm{C}_n$, and
- $\Delta(z) = \mathcal{F}((\Psi_1(z_1),\ldots,\Psi_n(z_n)),q)$.

Proof. We assume that for all i and j, where $j \neq i$, var(C_i) and var(C_j) are disjoint.

Let $t=|x|$, $m=|y|$, and $k=|z|$. If $u \in \mathbf{R}^t$, then u can be written as $u_1\cdot\ldots\cdot u_n\cdot b$ where $|u_i|=|x_i|$ for each i $(1 \leq i \leq n)$ and $|b|=|p|$. Similarly, elements of $\mathbf{R}^m$ can be written as $v_1\cdot\ldots\cdot v_n$ where $|v_i|=|y_i|$ for each i $(1 \leq i \leq n)$ and elements of $\mathbf{R}^k$ can be written as $w_1\cdot\ldots\cdot w_n\cdot d$ where $|w_i|=|z_i|$ for each i $(1 \leq i \leq n)$ and $|d|=|q|$.

(***a***): Since for each i $(1 \le i \le n)$, $(x_i,y_i,z_i,\mathbf{C}_i,\Psi_i)$ is a factoring of Φ_i, x_i, y_i, and z_i are disjoint sequences of distinct variables, therefore, because of the above assumption and the fact that p and q are also disjoint sequences of distinct variables, x, y, and z are disjoint sequences of distinct variables.

(***b***): Clearly $\mathrm{var}(\mathbf{X})=x\bullet y\bullet z$.

(***c***): Suppose $s \in z$, then $s \in z_i$ for some i $(1 \le i \le n)$ or $s \in q$. If $s \in z_i$, (*c*) holds by proof of Lemma 5.17. If $s \in q$, because of the condition on $\mathbf{C}$ in Definition 5.20, there exists a unique equality $\mathbf{E} \in \mathbf{C}$ such that $\mathrm{var}(\mathbf{E})=\{s\}$ or $\mathrm{var}(\mathbf{E})=\{s,l\}$ and $l \in p$. Therefore, each $s \in z$ occurs in a unique equality $\mathbf{E} \in \mathbf{X}$ such that $\mathrm{var}(\mathbf{E})=\{s\}$ or $\mathrm{var}(\mathbf{E})=\{s,l\}$ and $l \in x \cup y$.

(***d***): Δ is a partial function from $\mathbf{R}^k$ to $\mathcal{D}$. Suppose that for some $w \in \mathrm{scope}(\Delta)$, $(e,g),(e,h) \in \Delta(w)$. Then $w_i \in \mathrm{scope}(\Psi_i)$ for each i $(1 \le i \le n)$, $((\Psi_1(w_1),\ldots,\Psi_n(w_n)),w_{*|q|}) \in \mathrm{scope}(\mathcal{F})$ and $(e,g),(e,h) \in \mathcal{F}((\Psi_1(w_1),\ldots,\Psi_n(w_n)),w_{*|q|})$ so by the condition on $\mathcal{F}$ imposed by the definition of operation, $g=h$. Hence, Δ is a solid in $\mathcal{D}$.

It remains to show that the validity, conformity and identity conditions are satisfied.

(***Validity***): Suppose that $u \in \mathbf{R}^t$ and $u \in \mathrm{scope}(\Psi)$, then according to Definition 5.21, $\mathbf{C}_{p_1,\ldots,p_n,p,q}[\phi_1(u_1),\ldots,\phi_n(u_n),u_{*|p|},d]$ is true, and according to Lemma 5.22, $u_i \in \mathrm{scope}(\Phi_i)$ for each i $(1 \le i \le n)$. Since $(x_i,y_i,z_i,\mathbf{C}_i,\Psi_i)$ is a factoring of Φ_i, and $u_i \in \mathrm{scope}(\Phi_i)$, by the validity condition we can conclude that there exist v_i and w_i such that $(\mathbf{C}_i)_{x_i,y_i,z_i}[u_i,v_i,w_i]$ is true for each i $(1 \le i \le n)$. Therefore, $\mathbf{X}_{x,y,z}[u,v,w]$ is true where v is $v_1\bullet\ldots\bullet v_n$, and w is $w_1\bullet\ldots\bullet w_n\bullet d$, proving that the validity condition holds.

(***Conformity***): Suppose that $u \in \mathbf{R}^t$, $v \in \mathbf{R}^m, w \in \mathbf{R}^k$, and $\mathbf{X}_{x,y,z}[u,v,w]$ is true.

If $u \in \mathrm{scope}(\Psi)$, then according to Lemma 5.22, $u_i \in \mathrm{scope}(\Phi_i)$ for each i $(1 \le i \le n)$, and $((\Phi_1(u_1),\ldots,\Phi_n(u_n)),d) \in \mathrm{scope}(\mathcal{F})$. Since $\mathbf{X}_{x,y,z}[u,v,w]$ is true, then $(\mathbf{C}_i)_{x_i,y_i,z_i}[u_i,v_i,w_i]$ is true for each i $(1 \le i \le n)$, and since $u_i \in \mathrm{scope}(\Phi_i)$, by the conformity condition, $w_i \in \mathrm{scope}(\Psi_i)$ and by the identity condition $\Phi_i(u_i)=\Psi_i(w_i)$, for each i $(1 \le i \le n)$. Since $w_i \in \mathrm{scope}(\Psi_i)$ for each i $(1 \le i \le n)$, and $((\Psi_1(w_1),\ldots,\Psi_n(w_n)),d) \in \mathrm{scope}(\mathcal{F})$, $w \in \mathrm{scope}(\Delta)$.

If $w \in \text{scope}(\Delta)$, then $w_i \in \text{scope}(\Psi_i)$ for each i $(1 \leq i \leq n)$, and $((\Psi_1(w_1),\ldots,\Psi_n(w_n)),d) \in \text{scope}(\mathcal{F})$. Since $(\mathrm{C}_i)_{x_i,y_i,z_i}[u_i,v_i,w_i]$ is true and $w_i \in \text{scope}(\Psi_i)$ for each i $(1 \leq i \leq n)$, by the conformity condition on the factoring $(x_i,y_i,z_i,\mathrm{C}_i,\Psi_i)$ of Φ_i, $u_i \in \text{scope}(\Phi_i)$, and by the identity condition $\Phi_i(u_i)=\Psi_i(w_i)$, for each i $(1 \leq i \leq n)$. Therefore, $((\Phi_1(u_1),\ldots,\Phi_n(u_n)),d) \in \text{scope}(\mathcal{F})$. Since $u_i \in \text{scope}(\Phi_i)$ for each i $(1 \leq i \leq n)$, and $((\Phi_1(u_1),\ldots,\Phi_n(u_n)),d) \in \text{scope}(\mathcal{F})$, u is in the scope of the partial function defined by the expression $\mathcal{F}((\Phi_1(x_1),\ldots,\Phi_n(x_n)),q)$. Since $\mathrm{C}_{p_1,\ldots,p_n,p,q}[\phi_1(u_1),\ldots,\phi_n(u_n),u_{*|p|},d]$ is true, and u is in the scope of the partial function defined by the expression $\mathcal{F}((\Phi_1(x_1),\ldots,\Phi_n(x_n)),q)$, $u \in \text{scope}(\Psi)$, establishing the conformity condition.

(***Identity***): Suppose that $u \in \mathbf{R}^t$, $v \in \mathbf{R}^m$, $w \in \mathbf{R}^k$, and $u \in \text{scope}(\Psi)$ and $\mathrm{X}_{x,y,z}[u,v,w]$ is true. Since $u \in \text{scope}(\Psi)$, then u is in the scope of the partial function defined by the expression $\mathcal{F}((\Phi_1(x_1),\ldots,\Phi_n(x_n)),q)$. Therefore, $u_i \in \text{scope}(\Phi_i)$ for each i $(1 \leq i \leq n)$. Since $\mathrm{X}_{x\bullet y\bullet z}[u\bullet v\bullet w]$ is true, $(\mathrm{C}_i)_{x_i,y_i,z_i}[u_i,v_i,w_i]$ is true for each i $(1 \leq i \leq n)$, and since $(x_i,y_i,z_i,\mathrm{C}_i,\Psi_i)$ is a factoring of Φ_i, and $u_i \in \text{scope}(\Phi_i)$, by the conformity condition $w_i \in \text{scope}(\Psi_i)$ and by the identity condition $\Phi_i(u_i)=\Psi_i(w_i)$ for each i $(1 \leq i \leq n)$. Therefore,

$$\begin{aligned}\Psi(u) &= \mathcal{F}((\Phi_1(u_1),\ldots,\Phi_n(u_n)),d)\\ &= \mathcal{F}((\Psi_1(w_1),\ldots,\Psi_n(w_n)),d)\\ &= \Delta(w)\end{aligned}$$

establishing the identity condition. □

Lemma 5.25: If in Lemma 5.24, $(x_i,y_i,z_i,\mathrm{C}_i,\Psi_i)$ is a perfect factoring of Φ_i for each i $(1 \leq i \leq n)$, and $\mathrm{C}_{p_1,\ldots,p_n,p,q}[\phi_1(u_1),\ldots,\phi_n(u_n),u_{*|p|},d]$ implies that $((\Phi_1(u_1),\ldots,\Phi_n(u_n)),d) \in \text{scope}(\mathcal{F})$, then $(x,y,z,\mathrm{X},\Delta)$ is a perfect factoring of Ψ.

Proof. Since $\mathrm{X}_{x,y,z}[u,v,w]$ is true $(\mathrm{C}_i)_{x_i,y_i,z_i}[u_i,v_i,w_i]$ is true for each i $(1 \leq i \leq n)$, so for each i, $u_i \in \text{scope}(\Phi_i)$ since $(x_i,y_i,z_i,\mathrm{C}_i,\Psi_i)$ is a perfect factoring of Φ_i, and $\Phi_i(u_i)=\Psi_i(w_i)$ by the identity condition. Also, since $\mathrm{X}_{x,y,z}[u,v,w]$ is true, $\mathrm{C}_{p_1,\ldots,p_n,p,q}[\phi_1(u_1),\ldots,\phi_n(u_n),u_{*|p|},d]$ is true.

Therefore, $((\Phi_1(u_1),\ldots,\Phi_n(u_n)),d) \in \text{scope}(\mathcal{F})$, so that $((\Psi_1(w_1),\ldots,\Psi_n(w_n)),d) \in \text{scope}(\mathcal{F})$. Hence, $w \in \text{scope}(\Delta)$ and since $X_{x,y,z}[u,v,w]$ is true, $u \in \text{scope}(\Psi)$ by the conformity condition.

□

The introduction of extra variables in the above definitions not only facilitates the high-level parameterisation of solids constructed during assembly, it is also essential for defining certain behaviours for solids as we will see later.

We identify two cases with respect to extra variables. In the first case, as illustrated in the previous example, the new extra variables must indeed appear in the argument list of the new solid so that the new solid is parameterised with respect to the extra variables. In the second case, the extra variables of a solid may be such that they are dependant on other variables that already appear in the argument list of the new solid and including them would seem redundant. In the next section, we discuss how we remove such arguments.

5.4 Reduction

As discussed earlier, two solids are equivalent if they have the same image. In fact, given a solid, an infinite number of equivalent solids can be defined by introducing additional, possibly irrelevant variables. The notion of equivalence of solids is important when applying an operation, since if a solid that is to be used as an operand does not expose the right interface, it can be replaced by an equivalent solid that does. That is to say, each member of a set of equivalent solids highlights certain features of the solid which make it acceptable as an operand to certain operations.

5.4.1 Background

In [37], it was discussed that although the information that an operation requires from an operand solid can be computed from the solid, it would be easier if the solid provided the required information directly, so that the interfaces can be trivial functions that select the required parameters from the solid's variables. This implies that the variables of a solid must include all the variables required by those operations that may be applied to the solid. A process for transforming a solid into an equivalent one that exposes required interfaces was presented in [37]. This was achieved by adding the new required parameters to the variables while adjusting the definition of the solid accordingly. Then it was observed that this mechanism could introduce repeated variables and a remedy was proposed which removed the equivalent variables while maintaining the structure of both the solid and all the interfaces that it exposed. The final observation was that in the computed set of variables, there might be some that are related to others and not used in any of the interfaces that the solid is required to expose. This problem was addressed by removing the unnecessary variables.

We make two observations about the definition of sufficient and parameter sets for a solid presented in [37]. First, the condition in the definition of a sufficient set does not take invalid solids into account. Second, the condition in the definition of a parameter set for not including distinct equivalent variables is redundant as it is ensured by the minimality of a parameter set. Having made these observations, we reconstruct the definition of sufficient and parameter sets for solids defined as partial functions as follows.

Definition 5.26: If Φ be a solid in n variables and $\mathbf{N}$ is a set of interfaces to Φ, a subset P of its variables is said to be *sufficient for* Φ *with* $\mathbf{N}$ iff P includes all the variables required by the interfaces in $\mathbf{N}$, and for all $y, z \in \text{scope}(\Phi)$, if $y_i = z_i$ for all $i \in P$, then $\Phi(y) = \Phi(z)$. P is called *a parameter set for* Φ *with* $\mathbf{N}$ iff P is sufficient for Φ with $\mathbf{N}$, and no proper subset of P is sufficient for Φ with $\mathbf{N}$.

In the next section we introduce the notion of reduced sets of variables for solids. We identify conditions under which a reduced set becomes a parameter set.

5.4.2 Reduction Formalism and Methodology

In this section, we will present a mechanism for reducing solids by exploiting the factoring notion, where by "reducing" we mean removing unnecessary variables. We identify two conditions under which reducing a solid may be desirable. First, if an e-component is the final product of the execution of an LSD query, there may no longer be a need for the corresponding solid to expose any interfaces. This is in effect reducing the solid with respect to an empty set of interfaces. Second, a solid produced by applying an operation may have a variable not required by any interface and determined by other required variables. Such a variable may be a new one, introduced by the operation, or may be a variable of one of the operand solids which has been bound to other variables by the constraint of the operation.

We will present the reduction procedure using the following example.

Example 5.27: Consider the solid **Arm** illustrated in Figure 5.6 defined in 9 variables *b,c,d,e,f,g,l,a,r*. As indicated in the figure, the two points (*b,c*) and (*d,e*) are the centres of half circles with radius *r* that create the rounded ends of the solid, *l* is the distance between these points and *a* is the angle between the line that connects these points and the *x*-axis of the design space.

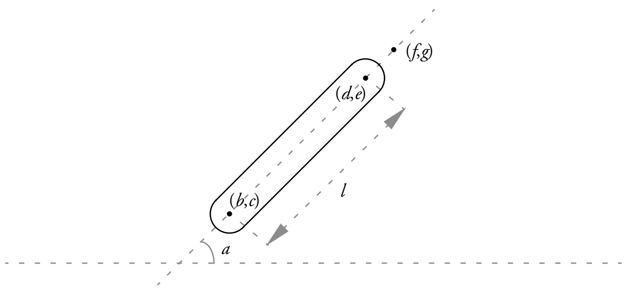


Figure 5.6: Solid **Arm**(b,c,d,e,l,a,r).

The point (*f,g*), called the *hinge*, lies on the line through (*b,c*) and (*d,e*) and is the centre of a circle with radius *r* tangent to the solid as shown. The point (*b,c*) is called the *pivot point*. The variables of the function **Arm** are constrained by the four equations

(I) $d-b = l.\cos(a)$ (II) $e-c = l.\sin(a)$

(III) $f = d+2r.\cos(a)$ (IV) $g = e+2r.\sin(a)$

Clearly, not all of the 9 variables of **Arm** are necessary. For example, the selection (b,c,d,e,r) is enough to fully define the solid, so there exists another solid in these variables, equivalent to **Arm**.

As we mentioned before, a procedure for reducing solids was discussed in [37]. We will present a reduction procedure that operates on the factorings of solids which is more specific than the process outlined in [37].

In solid modeling, constraints are frequently expressed in terms of equalities, for example, when a part has to be fixed at a certain distance from or tangent to another, or when a part must have a specific length. Our reduction algorithm considers only equalities in the factoring, however, the constraint of a factoring may consist of literals of any kind.

First we introduce some terms. If **C** is a set of constraints over n variables, an n-tuple of values for the n variables is said to be *a solution for* ***C*** iff it satisfies **C**. The set of all solutions for **C** is called *the solution set for* ***C***. We say two equalities are *distinct* iff they have different solution sets. A set of equalities **C** is called *connected* when members of any partition of **C** of size 2 share at least one variable. By *solving a constraint set* we mean finding a solution for the set or determining that no such solution exists.

If **C** is a non-empty set of m equality literals over n variables, we classify **C** as *under-constrained* or *over-constrained* if $n>m$ or $n<m$, respectively. **C** is called *well-constrained* when $n=m$ and no subset of **C** is over-constrained. An under-constrained set may have both well-constrained and over-constrained subsets. A set of equalities is *minimally well-constrained* if it is well-constrained and has no proper well-constrained subset. Although over-constrained sets of equalities may have solutions, they are frequently inconsistent.

We now prove some useful lemmas which later we will use in proving some important properties of sets of equalities.

Lemma 5.28: Each over-constrained set of equalities has at least one well-constrained subset.

Proof. Assume **C** is an over-constrained set of equalities. We use induction on $|\mathbf{C}|$.

Since **C** is over-constrained and has at least one variable, clearly, $|C|>1$.

Base case: If $|C|=2$, $|var(\mathbf{C})|=1$ and therefore each of the two equalities in **C** are well-constrained.

Assumption: Assume that each over-constrained set of equalities of size smaller than n has at least one well-constrained subset for some $n\geq 2$.

Induction: Let $|C|=n$, and $\mathbf{E} \in \mathbf{C}$. If {**E**} is well-constrained the lemma holds. If {**E**} is not well-constrained, then

If **C**–{**E**} is well-constrained, the lemma holds.

If **C**–{**E**} is over-constrained, since $|C–\{E\}|<n$ by the induction hypothesis **C**–{**E**} has at least one well-constrained subset. Since every subset of **C**–{**E**} is a subset of **C**, the lemma holds.

Now we prove that **C**–{**E**} cannot be under-constrained.

Clearly, $|var(\mathbf{C}–\{\mathbf{E}\})|\leq|var(\mathbf{C})|$ and since $|var(C)|<|C|$, $|var(C–\{E\})|<|C|$. If **C**–{**E**} is under-constrained, then $|C–\{E\}|<|var(\mathbf{C}–\{\mathbf{E}\})|$ and since $|C–\{E\}|=|C|–1$, $|C|–1<|var(C–\{E\})|<|C|$ which is impossible. Therefore, **C**–{**E**} cannot be under-constrained. □

Lemma 5.29: If **C** is a minimally well-constrained set of equalities, then every proper subset of **C** is under-constrained.

Proof. Since **C** is minimally well-constrained, no proper subset of **C** is well-constrained. If **X**, a proper subset of **C**, is over-constrained, then by Lemma 5.28, **X** has a well-constrained subset which is also a subset of **C**. Therefore, every proper subset of **C** is under-constrained. □

We make an important assumption about the constraint sets that our reduction algorithm is applied to: if the set is well-constrained then it is solvable. That is, either a solution can be found that satisfies all the constraints in the set or the set is inconsistent. For example, if an equality is over a single variable (or simplifies to such an equality as the result of substituting other variables with values), a constraint solver is capable of finding a value for the only variable to satisfy the equality or determine that no such value exists, regardless of the complexity of the equation.

In general, given a set of *n* equalities over *n* variables where the *n* equalities are independent, the substitution method can be used to solve the simultaneous equations. The difficulty of the substitution method strongly depends on the kind of expressions involved. There are also domain-specific methods for solving certain classes of equations. For example, for systems of simultaneous linear equations, Gauss-Jordan elimination [100] or Cholesky decomposition [88] can be used. General systems of simultaneous equations may be solved numerically using *n*-dimensional Newton's method [125].

Under the above assumption concerning the set of constraints, our reduction technique finds the smallest subset of the variables occurring in the set such that replacing them by values reduces the set of constraints to a well-constrained one.

Since our reduction algorithm is independent of the values of the constants in a set of constraints, in the following we assume that only one constant, denoted ε, occurs in the constraint set. Hence, if **C** is a set of constraints, and v is a variable, by $\mathbf{C}_v$ we mean $\mathbf{C}_v[\varepsilon]$.

Definition 5.30: If **C** and **D** are sets of equalities, then **D** is said to be a *trimming* of **C** iff:

- $\mathbf{D}=\mathbf{C}$ where **C** has no well-constrained subset.
- **D** is a trimming of $\mathbf{C}_{\mathrm{var}(\mathbf{X})}$ where $\mathbf{X} \subseteq \mathbf{C}$ and **X** is minimally well-constrained.

A set of equalities **C** is called *properly over-constrained* iff **C** is over-constrained and for each subset **V** of var(**C**) either $\mathbf{C}_\mathbf{V}=\varnothing$, or $\mathbf{C}_\mathbf{V}$ is over-constrained.

Lemma 5.31: If **C** is a properly over-constrained set of equalities, then for each subset **V** of var(**C**), either $\mathbf{C}_\mathbf{V}=\varnothing$, or $\mathbf{C}_\mathbf{V}$ is properly over-constrained.

Proof. Since **C** is properly over-constrained either $\mathbf{C}_\mathbf{V}=\varnothing$, or $\mathbf{C}_\mathbf{V}$ is over-constrained. In the latter case, for each subset **W** of var($\mathbf{C}_\mathbf{V}$), since $\mathbf{V}\cup\mathbf{W}$ is a subset of var(**C**), either $\mathbf{C}_{(\mathbf{V}\cup\mathbf{W})}=\varnothing$, or $C_{(V \cup W)}$ is over-constrained. But $C_{(V \cup W)}=(C_V)_W$, therefore, C_V is properly over-constrained. □

Lemma 5.32: If **C** is a minimally well-constrained set of equalities, then for each non-empty subset **V** of var(**C**), either $C_V=\varnothing$, or C_V is properly over-constrained.

Proof. If V=var(**C**), then $C_V=\varnothing$, and the result holds. Assume that V≠var(**C**) and let **X** be the subset of **C** such that **E** is in **X** iff var(**E**) is a subset of **V**. First, we point out some properties of **C**, C_V, and **X**.

$|C_V|=|C|-|X|$ (I)

$|var(C)|=|C|$ since **C** is minimally well-constrained (II)

$|var(C_V)|=|var(C)|-|V|$ (III)

By (I) and (II), $|var(C)|-|X|=|C_V|$ and by (III), $|var(C_V)|+|V|-|X|=|C_V|$. Since var(**C**)≠**V**, **X** is a proper subset of **C** so by Lemma 5.29, $|X|<|var(X)|$. But $|var(X)|\le|V|$ so that $|var(C_V)|<|C_V|$. Therefore, C_V is over-constrained.

Now let **W** be a non-empty subset of var(C_V). Since V∪W is a non-empty subset of var(**C**), either $C_{(V \cup W)}=\varnothing$, or $C_{(V \cup W)}$ is over-constrained, by the above reasoning. But $C_{(V \cup W)}=(C_V)_W$, therefore, C_V is properly over-constrained. □

Lemma 5.33: If **C** is a set of equalities and **X** a properly over-constrained subset of **C**, then **D** is a trimming of **C** iff **D** is a trimming of $C_{var(X)}$.

Proof. We prove this by induction on |**C**|. Clearly, since **C** has an over-constrained subset, |**C**|>1.

Base case: If |**C**|=2, since **C** has a properly over-constrained subset |var(**C**)|=1, **C**={**E**,**F**} such that var(**E**)=var(**F**)=var(**C**) and {**E**,**F**} is properly over-constrained. Since **E** and **F** are minimally well-constrained, ∅ is the only trimming of **C**. Also $C_{var(\{E,F\})}=\varnothing$, the only trimming of which is ∅.

Assumption: Assume that the result holds for all sets of fewer than n equalities, for some $n>2$.

Induction: Assume that $|\mathbf{C}|=n$. First we prove that if **D** is a trimming of **C**, then **D** is a trimming of $\mathbf{C}_{\text{var}(\mathbf{X})}$.

Since **D** is a trimming of **C**, there exists a minimally well-constrained subset **Y** of **C** such that **D** is a trimming of $\mathbf{C}_{\text{var}(\mathbf{Y})}$. If $\text{var}(\mathbf{X})\cap\text{var}(\mathbf{Y})=\varnothing$, then $\mathbf{X}\subseteq\mathbf{C}_{\text{var}(\mathbf{Y})}$. Since $|\mathbf{C}_{\text{var}(\mathbf{Y})}|<n$ and **X** is properly over-constrained in $\mathbf{C}_{\text{var}(\mathbf{Y})}$, by the induction hypothesis **D** is a trimming of $(\mathbf{C}_{\text{var}(\mathbf{Y})})_{\text{var}(\mathbf{X})}$. Since $(\mathbf{C}_{\text{var}(\mathbf{Y})})_{\text{var}(\mathbf{X})}=(\mathbf{C}_{\text{var}(\mathbf{X})})_{\text{var}(\mathbf{Y})}$, **D** is a trimming of $(\mathbf{C}_{\text{var}(\mathbf{X})})_{\text{var}(\mathbf{Y})}$. Therefore, **D** is a trimming of $\mathbf{C}_{\text{var}(\mathbf{X})}$.

If $\text{var}(\mathbf{X})\cap\text{var}(\mathbf{Y})\neq\varnothing$, since **X** is properly over-constrained and $\text{var}(\mathbf{X})\cap\text{var}(\mathbf{Y})$ is a subset of $\text{var}(\mathbf{X})$, either $\mathbf{X}_{\text{var}(\mathbf{X})\cap\text{var}(\mathbf{Y})}=\varnothing$, in which case $\text{var}(\mathbf{X})\subseteq\text{var}(\mathbf{Y})$ so that $\mathbf{C}_{\text{var}(\mathbf{Y})}=\mathbf{C}_{\text{var}(\mathbf{X}\cup\mathbf{Y})}$ and **D** is a trimming of $\mathbf{C}_{\text{var}(\mathbf{X}\cup\mathbf{Y})}$, or $\mathbf{X}_{\text{var}(\mathbf{X})\cap\text{var}(\mathbf{Y})}$ is properly over-constrained, by Lemma 5.31. But $\mathbf{X}_{\text{var}(\mathbf{X})\cap\text{var}(\mathbf{Y})}=\mathbf{X}_{\text{var}(\mathbf{Y})}$ so $\mathbf{X}_{\text{var}(\mathbf{Y})}$ is properly over-constrained and since $|\mathbf{C}_{\text{var}(\mathbf{Y})}|<n$, by the induction hypothesis, **D** is a trimming of $(\mathbf{C}_{\text{var}(\mathbf{Y})})_{\text{var}(\mathbf{X}_{\text{var}(\mathbf{Y})})}$. But $(\mathbf{C}_{\text{var}(\mathbf{Y})})_{\text{var}(\mathbf{X}_{\text{var}(\mathbf{Y})})}=\mathbf{C}_{\text{var}(\mathbf{X}\cup\mathbf{Y})}$, therefore **D** is a trimming of $\mathbf{C}_{\text{var}(\mathbf{X}\cup\mathbf{Y})}$.

Since **Y** is minimally well-constrained and $\text{var}(\mathbf{X})\cap\text{var}(\mathbf{Y})$ is a subset of $\text{var}(\mathbf{Y})$, by Lemma 5.32, either $\mathbf{Y}_{\text{var}(\mathbf{X})\cap\text{var}(\mathbf{Y})}=\varnothing$, in which case $\text{var}(\mathbf{Y})\subseteq\text{var}(\mathbf{X})$ so that $\mathbf{C}_{\text{var}(\mathbf{X})}=\mathbf{C}_{\text{var}(\mathbf{X}\cup\mathbf{Y})}$ and **D** is a trimming of $\mathbf{C}_{\text{var}(\mathbf{X})}$, or $\mathbf{Y}_{\text{var}(\mathbf{X})\cap\text{var}(\mathbf{Y})}$ is properly over-constrained. But $\mathbf{Y}_{\text{var}(\mathbf{X})\cap\text{var}(\mathbf{Y})}=\mathbf{Y}_{\text{var}(\mathbf{X})}$ so $\mathbf{Y}_{\text{var}(\mathbf{X})}$ is properly over-constrained, in which case, since $|\mathbf{C}_{\text{var}(\mathbf{X})}|<n$, by the induction hypothesis, a trimming of $(\mathbf{C}_{\text{var}(\mathbf{X})})_{\text{var}(\mathbf{Y}_{\text{var}(\mathbf{X})})}$ is a trimming of $\mathbf{C}_{\text{var}(\mathbf{X})}$. Since $\mathbf{C}_{\text{var}(\mathbf{X}\cup\mathbf{Y})}=(\mathbf{C}_{\text{var}(\mathbf{X})})_{\text{var}(\mathbf{Y}_{\text{var}(\mathbf{X})})}$ and **D** is a trimming of $\mathbf{C}_{\text{var}(\mathbf{X}\cup\mathbf{Y})}$, **D** is a trimming of $\mathbf{C}_{\text{var}(\mathbf{X})}$.

Now we prove that if **D** is a trimming of $\mathbf{C}_{\text{var}(\mathbf{X})}$, then **D** is a trimming of **C**.

Since **X** is over-constrained, by Lemma 5.28, there exists a minimally well-constrained subset **Y** of **X**. Since **X** is properly over-constrained and $\text{var}(\mathbf{Y})$ is a subset of $\text{var}(\mathbf{X})$, $\mathbf{X}_{\text{var}(\mathbf{Y})}$ is properly over-constrained, and since $|\mathbf{C}_{\text{var}(\mathbf{Y})}|<n$, by the induction hypothesis, a trimming of

$(\mathbf{C}_{\mathrm{var}(\mathbf{Y})})_{\mathrm{var}(\mathbf{X}_{\mathrm{var}(\mathbf{Y})})}$ is a trimming of $\mathbf{C}_{\mathrm{var}(\mathbf{Y})}$. But $(\mathbf{C}_{\mathrm{var}(\mathbf{Y})})_{\mathrm{var}(\mathbf{X}_{\mathrm{var}(\mathbf{Y})})}=\mathbf{C}_{\mathrm{var}(\mathbf{X})}$ and **D** is a trimming of $\mathbf{C}_{\mathrm{var}(\mathbf{X})}$ so that **D** is a trimming of $\mathbf{C}_{\mathrm{var}(\mathbf{Y})}$. Therefore, **D** is a trimming of **C**. □

If **C** is a set of equalities, we define #(**C**) to be 0 if **C** has no well-constrained subsets, and #(**C**)=|**C**| otherwise.

Lemma 5.34: If **C** is a set of equalities and **D** is a trimming of **C**, then either **D**=**C**, or for every minimally well-constrained subset **X** of **C**, **D** is a trimming of $\mathbf{C}_{\mathrm{var}(\mathbf{X})}$.

Proof. We prove this by induction on #(**C**).

Base case: If #(**C**)=0, **C** has no well-constrained subsets, therefore, **D**=**C**.

Assumption: Assume that for some $n>0$ the result holds for any set of equalities **B** such that #(**B**)<n.

Induction: Let #(**C**)=n and **X** be some minimally well-constrained subset of **C**. Since **D** is a trimming of **C**, there exists a minimally well-constrained subset **Y** of **C** such that **D** is a trimming of $\mathbf{C}_{\mathrm{var}(\mathbf{Y})}$. Now either **X**=**Y**, in which case the result is proved, or **X**≠**Y**. In the latter case, if $\mathrm{var}(\mathbf{X})\cap\mathrm{var}(\mathbf{Y})=\varnothing$, then $\mathbf{X}\subseteq\mathbf{C}_{\mathrm{var}(\mathbf{Y})}$. Since $\#(\mathbf{C}_{\mathrm{var}(\mathbf{Y})})<n$ and **D** is a trimming of $\mathbf{C}_{\mathrm{var}(\mathbf{Y})}$, by the induction hypothesis **D** is a trimming of $(\mathbf{C}_{\mathrm{var}(\mathbf{Y})})_{\mathrm{var}(\mathbf{X})}$. Since $(\mathbf{C}_{\mathrm{var}(\mathbf{Y})})_{\mathrm{var}(\mathbf{X})}=(\mathbf{C}_{\mathrm{var}(\mathbf{X})})_{\mathrm{var}(\mathbf{Y})}$, **D** is a trimming of $(\mathbf{C}_{\mathrm{var}(\mathbf{X})})_{\mathrm{var}(\mathbf{Y})}$. Therefore, **D** is a trimming of $\mathbf{C}_{\mathrm{var}(\mathbf{X})}$.

If $\mathrm{var}(\mathbf{X})\cap\mathrm{var}(\mathbf{Y})\neq\varnothing$, since **X** is minimally well-constrained, by Lemma 5.32 either $\mathbf{X}_{\mathrm{var}(\mathbf{X})\cap\mathrm{var}(\mathbf{Y})}=\varnothing$, in which case $\mathrm{var}(\mathbf{X})\subseteq\mathrm{var}(\mathbf{Y})$ so that $\mathbf{C}_{\mathrm{var}(\mathbf{Y})}=\mathbf{C}_{\mathrm{var}(\mathbf{X}\cup\mathbf{Y})}$ and **D** is a trimming of $\mathbf{C}_{\mathrm{var}(\mathbf{X}\cup\mathbf{Y})}$, or $\mathbf{X}_{\mathrm{var}(\mathbf{X})\cap\mathrm{var}(\mathbf{Y})}$ is properly over-constrained. But $\mathbf{X}_{\mathrm{var}(\mathbf{X})\cap\mathrm{var}(\mathbf{Y})}=\mathbf{X}_{\mathrm{var}(\mathbf{Y})}$ so $\mathbf{X}_{\mathrm{var}(\mathbf{Y})}$ is properly over-constrained, in which case, since **D** is a trimming of $\mathbf{C}_{\mathrm{var}(\mathbf{Y})}$, by Lemma 5.33 **D** is a trimming of $(\mathbf{C}_{\mathrm{var}(\mathbf{Y})})_{\mathrm{var}(\mathbf{X}_{\mathrm{var}(\mathbf{Y})})}=\mathbf{C}_{\mathrm{var}(\mathbf{X}\cup\mathbf{Y})}$. Also by Lemma 5.32, since **Y** is minimally well-constrained either $\mathbf{Y}_{\mathrm{var}(\mathbf{X})\cap\mathrm{var}(\mathbf{Y})}=\varnothing$, in which case $\mathrm{var}(\mathbf{Y})\subseteq\mathrm{var}(\mathbf{X})$ so that **D** is a trimming of $\mathbf{C}_{\mathrm{var}(\mathbf{X})}$ because $\mathbf{C}_{\mathrm{var}(\mathbf{X})}=\mathbf{C}_{\mathrm{var}(\mathbf{X}\cup\mathbf{Y})}$, or $\mathbf{Y}_{\mathrm{var}(\mathbf{X})\cap\mathrm{var}(\mathbf{Y})}$ is properly over-constrained. But $\mathbf{Y}_{\mathrm{var}(\mathbf{X})\cap\mathrm{var}(\mathbf{Y})}=\mathbf{Y}_{\mathrm{var}(\mathbf{X})}$ so $\mathbf{Y}_{\mathrm{var}(\mathbf{X})}$ is properly over-constrained, in which case, by Lemma 5.33, a trimming of $(\mathbf{C}_{\mathrm{var}(\mathbf{X})})_{\mathrm{var}(\mathbf{Y}_{\mathrm{var}(\mathbf{X})})}=\mathbf{C}_{\mathrm{var}(\mathbf{X}\cup\mathbf{Y})}$ is a trimming of $\mathbf{C}_{\mathrm{var}(\mathbf{X})}$. Since **D** is a trimming of $\mathbf{C}_{\mathrm{var}(\mathbf{X}\cup\mathbf{Y})}$, **D** is a trimming of $\mathbf{C}_{\mathrm{var}(\mathbf{X})}$. □

Lemma 5.35: If **C** is a set of equalities, and **D** and **E** are trimmings of **C**, then **D**=**E**.

Proof. We use induction on #(**C**).

Base case: If #(**C**)=0, **C** has no well-constrained subsets so that **D**=**E**=**C**.

Assumption: Assume that for some n>0, the result holds for any set of equalities **B** such that #(**B**)<n.

Induction: Let #(**C**)=n. Since **D** is a trimming of **C** and **C** has at least one well-constrained subset, there exists a minimally well-constrained subset **X** of **C** such that **D** is a trimming of $\mathbf{C}_{\text{var}(\mathbf{X})}$. But **E** is a trimming of **C**, so by Lemma 5.34 **E** is a trimming of $\mathbf{C}_{\text{var}(\mathbf{X})}$. Since #($\mathbf{C}_{\text{var}(\mathbf{X})}$)<$n$, by the induction hypothesis, **D**=**E**. □

If **C** is a set of equalities, since the trimming of **C** is unique, we denote it by ⊥**C**. Note that since ⊥**C** has no well-constrained subset, ⊥(⊥**C**)=⊥**C**.

Lemma 5.36: If **C** is a set of equalities and u ∈var(**C**), then u occurs in a well-constrained subset of **C** iff u ∉var(⊥**C**).

Proof. First we prove that if u is in a well-constrained subset of **C**, then u∉var(⊥**C**). We prove this by induction on #(**C**).

Base case: If #(**C**)=0, **C** has no well-constrained subset so the result holds vacuously.

Assumption: Assume that for some n>0 the result holds for any set of equalities **B** such that #(**B**)<n

Induction: Assume #(**C**)=n, u is in a well-constrained subset **Y** of **C**, and **X** is a minimally well-constrained subset of **C**. If u ∈var(**X**), then u ∉var(⊥($\mathbf{C}_{\text{var}(\mathbf{X})}$)). But by Lemma 5.34 and Lemma 5.35, ⊥($\mathbf{C}_{\text{var}(\mathbf{X})}$)=⊥**C**, therefore, u ∉var(⊥**C**).

If u ∉var(**X**), then u ∈var($\mathbf{Y}_{\text{var}(\mathbf{X})}$). Clearly, $\mathbf{Y}_{\text{var}(\mathbf{X})}$ is well-constrained in $\mathbf{C}_{\text{var}(\mathbf{X})}$, and since #($\mathbf{C}_{\text{var}(\mathbf{X})}$)<$n$, by the induction hypothesis, u ∉var(⊥($\mathbf{C}_{\text{var}(\mathbf{X})}$)). But by Lemma 5.34 and Lemma 5.35, ⊥($\mathbf{C}_{\text{var}(\mathbf{X})}$)=⊥**C**, therefore, u ∉var(⊥**C**).

Now we prove that if $u \notin \mathrm{var}(\perp\mathbf{C})$, then u is in a well-constrained subset of $\mathbf{C}$ using induction on $\#(\mathbf{C})$.

Base case: If $\#(\mathbf{C})=0$, $\mathbf{C}$ has no well-constrained subset so that for all $u \in \mathrm{var}(\mathbf{C})$, $u \in \mathrm{var}(\perp\mathbf{C})$ and the result holds vacuously.

Assumption: Assume that for some $n>0$ the result holds for any set of equalities $\mathbf{B}$ such that $\#(\mathbf{B})<n$.

Induction: Assume $\#(\mathbf{C})=n$, $u \notin \mathrm{var}(\perp\mathbf{C})$, and $\mathbf{X}$ is a minimally well-constrained subset of $\mathbf{C}$. If $u \in \mathrm{var}(\mathbf{X})$, the result holds.

Assume that $u \notin \mathrm{var}(\mathbf{X})$. By Lemmas 5.34 and 5.35, $\perp(\mathbf{C}_{\mathrm{var}(\mathbf{X})})=\perp\mathbf{C}$ so that $\mathrm{var}(\perp(\mathbf{C}_{\mathrm{var}(\mathbf{X})}))=\mathrm{var}(\perp\mathbf{C})$. Therefore, $u \notin \mathrm{var}(\perp(\mathbf{C}_{\mathrm{var}(\mathbf{X})}))$ and since $\#(\mathbf{C}_{\mathrm{var}(\mathbf{X})})<n$, by the induction hypothesis, u is in a well-constrained subset $\mathbf{Y}_{\mathrm{var}(\mathbf{X})}$ of $\mathbf{C}_{\mathrm{var}(\mathbf{X})}$. Let $\mathbf{Z}=\{\mathbf{E} \mid \mathbf{E} \in \mathbf{Y}, \mathbf{E} \notin \mathbf{X}$, and $\mathrm{var}(\mathbf{E}) \subseteq \mathrm{var}(\mathbf{X})\}$, then

$$
\begin{aligned}
|\mathbf{X}\cup(\mathbf{Y}-\mathbf{Z})| &= |\mathbf{X}|+|\mathbf{Y}-\mathbf{Z}| && \text{since } \mathbf{X}\cap(\mathbf{Y}-\mathbf{Z})=\varnothing \\
&= |\mathrm{var}(\mathbf{X})|+|\mathbf{Y}-\mathbf{Z}| && \text{since } \mathbf{X} \text{ is minimally well-constrained} \\
&= |\mathrm{var}(\mathbf{X})|+|\mathbf{Y}_{\mathrm{var}(\mathbf{X})})| && \text{since } \mathrm{var}(\mathbf{Z}) \subseteq \mathrm{var}(\mathbf{X}) \\
&= |\mathrm{var}(\mathbf{X})|+|\mathrm{var}(\mathbf{Y}_{\mathrm{var}(\mathbf{X})})| && \text{since } \mathbf{Y}_{\mathrm{var}(\mathbf{X})} \text{ is well-constrained} \\
&= |\mathrm{var}(\mathbf{X})|+|\mathrm{var}(\mathbf{Y})|-|\mathrm{var}(\mathbf{X})\cap\mathrm{var}(\mathbf{Y})| \\
&= |\mathrm{var}(\mathbf{X}\cup(\mathbf{Y}-\mathbf{Z}))| && \text{since } \mathrm{var}(\mathbf{Z}) \subseteq \mathrm{var}(\mathbf{X})
\end{aligned}
$$

Therefore, $\mathbf{X}\cup(\mathbf{Y}-\mathbf{Z})$ is well constrained in $\mathbf{C}$ and because $u \in \mathrm{var}(\mathbf{X}\cup(\mathbf{Y}-\mathbf{Z}))$, u is in a well-constrained subset of $\mathbf{C}$. □

Definition 5.37: If $\mathbf{C}$ is a set of equalities, then a subset s of $\mathrm{var}(\perp\mathbf{C})$ is said to be a *reduced set for C* iff

- $s=\varnothing$ and $\perp\mathbf{C}=\varnothing$, or
- $s\neq\varnothing$ and $\perp\mathbf{C}\neq\varnothing$ and for each $v \in s$, $s-\{v\}$ is a reduced set for $(\perp\mathbf{C})_v$.

In the above definition, since $s\subseteq\text{var}(\perp\mathbf{C})$, $s-\{v\}\subseteq\text{var}((\perp\mathbf{C})_v)$.

The intuition behind the above definition is that the set of variables of a set of equalities can be divided, not necessarily uniquely, into two sets, a reduced set and the others. Substituting all the variables in the reduced set with values will simplify the set of equalities to a well-constrained set which a constraint solver could be employed to solve. The constraint solver would either generate a solution for the well-constrained set, or show that the constraint is unsatisfiable. Furthermore, given the original set of equalities, the constraint solver would not be able to compute values for the variables in any reduced set. This latter feature is an important property of reduced sets. To clarify the above definition, we present a few simple examples chosen to highlight some of the motivations.

Example 5.38: Let $\mathbf{C}=\{\mathbf{E}\}$ where $\mathbf{E}$ is an equality such that $\text{var}(\mathbf{E})=\{a\}$, then since $\{\mathbf{E}\}$ is minimally well-constrained, $\perp\mathbf{C}=\varnothing$ and $\varnothing$ is a reduced set for $\mathbf{C}$.

Example 5.39: Let $\mathbf{C}=\{\mathbf{E}_1,\mathbf{E}_2\}$ where $\mathbf{E}_1$ and $\mathbf{E}_2$ are equalities such that $\text{var}(\mathbf{E}_1)=\text{var}(\mathbf{E}_2)=\{a,b\}$, then $\{\mathbf{E}_1,\mathbf{E}_2\}$ is minimally well-constrained and $\perp\mathbf{C}=\mathbf{C}_{\text{var}(\{\mathbf{E}_1,\mathbf{E}_2\})}=\varnothing$ and therefore $\varnothing$ is a reduced set for $\mathbf{C}$.

Example 5.40: Let $\mathbf{C}=\{\mathbf{E}_1,\mathbf{E}_2\}$ where $\mathbf{E}_1$ and $\mathbf{E}_2$ are equalities such that $\text{var}(\mathbf{E}_1)=\{a,b,c\}$ and $\text{var}(\mathbf{E}_2)=\{a,b\}$. Because $\mathbf{C}$ has no well-constrained subset $\perp\mathbf{C}=\{\mathbf{E}_1,\mathbf{E}_2\}$, and since $(\perp\mathbf{C})_c=\varnothing$, as in Example 5.39, $\varnothing$ is a reduced set for $(\perp\mathbf{C})_c$. Hence, $\{c\}$ is a reduced set for $\mathbf{C}$.

Example 5.41: Let $\mathbf{C}=\{\mathbf{E}_1,\mathbf{E}_2,\mathbf{E}_3\}$ where $\mathbf{E}_1$, $\mathbf{E}_2$, and $\mathbf{E}_3$ are equalities such that $\text{var}(\mathbf{E}_1)=\{a,b,c\}$, $\text{var}(\mathbf{E}_2)=\{a,b\}$, and $\text{var}(\mathbf{E}_3)=\{c,e\}$. Since $\perp\mathbf{C}=\mathbf{C}\neq\varnothing$, $(\perp\mathbf{C})_e=\{\mathbf{E}_1,\mathbf{E}_2\}$ ($\{\mathbf{E}_3\}_e$ is minimally well-constrained in $\mathbf{C}_e$), and $\varnothing$ is a reduced set for $(\perp\mathbf{C})_e$ (Example 5.40), $\{e\}$ is a reduced set for $\mathbf{C}$.

Example 5.42: Let $\mathbf{C}=\{\mathbf{E}_1,\mathbf{E}_2,\mathbf{E}_3,\mathbf{E}_4\}$ where $\mathbf{E}_1$, $\mathbf{E}_2$, $\mathbf{E}_3$, and $\mathbf{E}_4$ are equalities such that $\text{var}(\mathbf{E}_1)=\{a,b,c\}$, $\text{var}(\mathbf{E}_2)=\{a,b,d\}$, $\text{var}(\mathbf{E}_3)=\{c,e\}$, $\text{var}(\mathbf{E}_4)=\{d,f\}$. Since $\perp\mathbf{C}=\mathbf{C}\neq\varnothing$, $(\mathbf{C})_f=\{\mathbf{E}_1,\mathbf{E}_2,\mathbf{E}_3\}$ ($\{\mathbf{E}_4\}_f$ is minimally well-constrained in $\mathbf{C}_f$), and $\{e\}$ is a reduced set for $(\perp\mathbf{C})_f$ (Example 5.41), and $\{f\}$ is a reduced set for $(\perp\mathbf{C})_e$ (similar to Example 5.41), $\{e,f\}$ is a reduced set for $\mathbf{C}$. Note that $\{e,f\}$ is not the only reduced set for **C**. There are 12 others, namely $\{a,b\}$, $\{a,c\}$, $\{a,d\}$, $\{a,e\}$, $\{a,f\}$, $\{b,c\}$, $\{b,d\}$, $\{b,e\}$, $\{b,f\}$, $\{c,d\}$, $\{c,f\}$, and $\{d,e\}$.

Example 5.43: Let **C** be as in Example 5.42 except that var(E_4)=$\{a,d,f\}$. Then s=$\{a,b\}$ is a reduced set for **C**. In addition to $\{d,f\}$, which is another reduced set for this example, every reduced set for Example 5.42 is also a reduced set for this example.

Example 5.44: Let **C**=$\{E_1,E_2,E_3,E_4,E_5,E_6\}$ where E_1, E_2, E_3, E_4, E_5, and E_6 are equalities such that var(E_1)=$\{a,b\}$, var(E_2)=$\{a,b\}$, var(E_3)=$\{c,d\}$, var(E_4)=$\{c,d\}$, var(E_5)=$\{e,f\}$, and var(E_6)=$\{e,f\}$. Since $\{E_1,E_2\}$, $\{E_3,E_4\}$, and $\{E_5,E_6\}$ are minimally well-constrained, $\perp$**C**=$\emptyset$. Therefore, $\emptyset$ is the only reduced set for **C**.

Example 5.45: Let **C**=$\{E_1,E_2,E_3,E_4,E_5,E_6\}$ where E_1, E_2, E_3, E_4, E_5, and E_6 are equalities such that var(E_1)=$\{a,b,c,d\}$, var(E_2)=$\{d,g,e,j\}$, var(E_3)=$\{e,f\}$, var(E_4)=$\{f,e\}$, var(E_5)=$\{i,h\}$, and var(E_6)=$\{i,g,h\}$. s=$\{a,b,c,j\}$ is a reduced set for **C**. $\{b,c,i,j\}$ is another reduced set for **C**.

Example 5.46: Let **C**=$\{E_1,E_2,E_3,E_4,E_5,E_6,E_7\}$ where E_1, E_2, E_3, E_4, E_5, E_6, and E_7 are equalities such that var(E_1)=$\{a,b,c,d\}$, var(E_2)=$\{d,g,e,j\}$, var(E_3)=$\{e,f\}$, var(E_4)=$\{e,f\}$, var(E_5)=$\{e,f\}$, var(E_6)=$\{i,h\}$, and var(E_7)=$\{i,g,h\}$. $\{a,b,c,j\}$ is a reduced set for **C**. $\{b,c,i,j\}$ is another reduced set for **C**.

Example 5.47: In Example 5.27, **C**=$\{E_1,E_2,E_3,E_4\}$ where E_1, E_2, E_3, and E_4 are tagged with (I), (II), (III), and (IV), respectively. $\{b,f,g,a,r\}$, $\{b,c,d,a,r\}$, and $\{b,c,f,g,a\}$ are three reduced sets for **C**.

We prove some useful lemmas which later we will use in proving some important properties of reduced sets.

Lemma 5.48: If **C** is a set of equalities, then s is a reduced set for **C** iff it is a reduced set for $\perp$**C**.

Proof. Suppose $s=\emptyset$. If s is a reduced set for **C**, then $\perp\mathbf{C}=\emptyset=s$. Since $\perp(\perp\mathbf{C})=\perp\mathbf{C}=\emptyset=s$, s is a reduced set for $\perp\mathbf{C}$. If s is a reduced set for $\perp\mathbf{C}$, then $\perp(\perp\mathbf{C})=\emptyset=s$. Since $\perp(\perp\mathbf{C})=\perp\mathbf{C}$, $\perp\mathbf{C}=\emptyset=s$. Hence, s is a reduced set for **C**.

Now suppose $s\neq\emptyset$. If s is a reduced set for **C**, $\mathbf{C}\neq\emptyset$ and for each $v\in s$, $s-\{v\}$ is a reduced set for $(\perp\mathbf{C})_v$. Since $\perp(\perp\mathbf{C})=\perp\mathbf{C}$, for each $v\in s$, $s-\{v\}$ is a reduced set for $(\perp(\perp\mathbf{C}))_v$. Hence, s is a reduced set for $\perp\mathbf{C}$. If s is a reduced set for $\perp\mathbf{C}$, $\perp\mathbf{C}\neq\emptyset$ and for each $v\in s$, $s-\{v\}$ is a reduced set for $(\perp(\perp\mathbf{C}))_v$. Since $\perp\mathbf{C}=\perp(\perp\mathbf{C})$, for each $v\in s$, $s-\{v\}$ is a reduced set for $(\perp\mathbf{C})_v$. Hence, s is a reduced set for **C**. □

Lemma 5.49: If **C** is a set of equalities and $z\in\mathrm{var}(\perp\mathbf{C})$, then $\perp((\perp\mathbf{C})_z)=\perp(\mathbf{C}_z)$.

Proof. We prove this by induction on $\#(\mathbf{C})$.

Base case: If $\#(\mathbf{C})=0$, $\perp\mathbf{C}=\mathbf{C}$, therefore, $\perp((\perp\mathbf{C})_z)=\perp(\mathbf{C}_z)$.

Assumption: Assume that for some $n>0$ the result holds for any set of equalities **B** such that $\#(\mathbf{B})<n$.

Induction: Assume $\#(\mathbf{C})=n$. Let **X** be a minimally well-constrained subset of **C**. Since $\#(\mathbf{C}_{\mathrm{var}(\mathbf{X})})<n$, and $z\in\mathrm{var}(\perp(\mathbf{C}_{\mathrm{var}(\mathbf{X})}))$, by the induction hypothesis, $\perp((\perp(\mathbf{C}_{\mathrm{var}(\mathbf{X})}))_z)=\perp((\mathbf{C}_{\mathrm{var}(\mathbf{X})})_z)$, and since $\perp\mathbf{C}=\perp(\mathbf{C}_{\mathrm{var}(\mathbf{X})})$ by Lemma 5.34, $\perp((\perp\mathbf{C})_z)=\perp((\mathbf{C}_z)_{\mathrm{var}(\mathbf{X})})$. Since $z\in\mathrm{var}(\perp\mathbf{C})$, by Lemma 5.36 $z\notin\mathrm{var}(\mathbf{X})$ so by Lemma 5.34, $\perp((\mathbf{C}_z)_{\mathrm{var}(\mathbf{X})})=\perp(\mathbf{C}_z)$, therefore $\perp((\perp\mathbf{C})_z)=\perp(\mathbf{C}_z)$. □

Lemma 5.50: If **C** is a set of equalities, s a reduced set for **C**, and y a subset of s, then $s-y$ is a reduced set for $\mathbf{C}_y$.

Proof. We prove this by induction on $|y|$.

Base case: If $|y|=0$, $s-y=s$ is a reduced set for $\mathbf{C}_\emptyset=\mathbf{C}$.

Assumption: Assume that for all subsets of s of size less than $n>1$ the result holds.

Induction: Suppose $|y|=n$ and $z\in y$. Since $|y-\{z\}|<n$ and s is a reduced set for **C**, by the induction hypothesis, $s-(y-\{z\})$ is a reduced set for $\mathbf{C}_{y-\{z\}}$. Since $z\in(s-(y-\{z\}))$ and $s-(y-\{z\})$ is

a reduced set for $\mathbf{C}_{y-\{z\}}$, $(s-(y-\{z\}))-\{z\}$ is a reduced set for $(\perp(\mathbf{C}_{y-\{z\}}))_z$ and, by Lemma 5.48, for $\perp((\perp(\mathbf{C}_{y-\{z\}}))_z)$. Since $\perp((\perp(\mathbf{C}_{y-\{z\}}))_z)=\perp((\mathbf{C}_{y-\{z\}})_z)$ (by Lemma 5.49, because $z\in(s-(y-\{z\}))$ and $s-(y-\{z\})$ is a reduced set for $\mathbf{C}_{y-\{z\}}$ so that $z\in\mathrm{var}(\perp(\mathbf{C}_{y-\{z\}}))$), $(s-(y-\{z\}))-\{z\}$ is a reduced set for $\perp((\mathbf{C}_{y-\{z\}})_z)$, and by Lemma 5.48, for $(\mathbf{C}_{y-\{z\}})_z$. But $(s-(y-\{z\}))-\{z\}=s-y$ and $(\mathbf{C}_{y-\{z\}})_z=\mathbf{C}_y$. Therefore, $s-y$ is a reduced set for $\mathbf{C}_y$. □

Definition 5.51: If **C** is a set of equalities and $v\subseteq\mathrm{var}(\mathbf{C})$, then v is said to be *strictly constrained in C* iff $v=\mathrm{var}(\mathbf{E})$ for some equality $\mathbf{E}\in\mathbf{C}$.

Lemma 5.52: If **C** is a set of equalities and s a reduced set for **C**, then no subset of s is strictly constrained in **C**.

Proof. Suppose y is a subset of s strictly constrained in **C**, then there exists an equality **E** in **C** such that $\mathrm{var}(\mathbf{E})=y$.

Since s is a reduced set for **C**, for each $z\in y$, $s-(y-\{z\})$ is a reduced set for $\mathbf{C}_{y-\{z\}}$ (Lemma 5.50). Since $|\{\mathbf{E}_{y-\{z\}}\}|=|\mathrm{var}(\{\mathbf{E}_{y-\{z\}}\})|=1$, $\{\mathbf{E}_{y-\{z\}}\}$ is minimally well-constrained so cannot be in $\perp(\mathbf{C}_{y-\{z\}})$. Hence, $z\notin\mathrm{var}(\mathbf{C}_{y-\{z\}})$ and cannot be in $s-(y-\{z\})$ contrary to our choice of z. Therefore, no subset of s is strictly constrained in **C**. □

An implication of this lemma is that if **C** is a set of equalities and s is a reduced set for **C**, then $|\mathbf{C}_s|=|\mathbf{C}|$. Suppose this is not the case, then there is some equality **E** in **C** such that $\mathrm{var}(\mathbf{E})\subseteq s$, so that $\mathrm{var}(\mathbf{E})$ is a strictly constrained subset of s, contrary to the lemma. Since by Lemma 5.48, s is a reduced set for $\perp\mathbf{C}$, $|(\perp\mathbf{C})_s|=|\perp\mathbf{C}|$.

Lemma 5.53: If **C** is a set of equalities and $v\in\mathrm{var}(\perp\mathbf{C})$, then $\mathrm{var}(\perp(\mathbf{C}_v))\subseteq\mathrm{var}(\perp\mathbf{C})$.

Proof. Since $v\in\mathrm{var}(\perp\mathbf{C})$, by Lemma 5.49 $\perp(\mathbf{C}_v)=\perp((\perp\mathbf{C})_v)$ so that $\mathrm{var}(\perp(\mathbf{C}_v))=\mathrm{var}(\perp((\perp\mathbf{C})_v))$. Clearly, $\mathrm{var}(\perp((\perp\mathbf{C})_v))\subseteq\mathrm{var}(\perp\mathbf{C})$, therefore, $\mathrm{var}(\perp(\mathbf{C}_v))\subseteq\mathrm{var}(\perp\mathbf{C})$. □

Lemma 5.54: If **C** is a set of equalities, $v\in\mathrm{var}(\perp\mathbf{C})$, s a reduced set for $\mathbf{C}_v$, and $u\in s$, then $v\in\mathrm{var}(\perp(\mathbf{C}_u))$.

Proof. Suppose $v \notin \text{var}(\perp(\mathbf{C}_u))$, then by Lemma 5.36, v occurs in a well-constrained set $\mathbf{X}_u \subseteq \mathbf{C}_u$, hence $u \in \text{var}(\mathbf{X})$ since otherwise $\mathbf{X}_u = \mathbf{X}$ is a well-constrained subset of $\mathbf{C}$ in which v occurs, so that $v \notin \text{var}(\perp\mathbf{C})$ by Lemma 5.36. Since $\mathbf{X}_u$ is well-constrained, $|\mathbf{X}_u| = |\text{var}(\mathbf{X}_u)|$. Note that neither $\{v\}$ nor $\{u\}$ is strictly constrained in $\mathbf{C}$, and $\mathbf{X}$ cannot include any equality $\mathbf{E}$ such that $\text{var}(\mathbf{E}) = \{v, u\}$ because in that case, $\mathbf{E}_v$ would be minimally well-constrained and u could not be in s. Therefore, $|\mathbf{X}_v| = |\mathbf{X}_u| = |\text{var}(\mathbf{X}_u)| = |\text{var}(\mathbf{X}_v)|$. Hence, $|\mathbf{X}_v| = |\text{var}(\mathbf{X}_v)|$ which means that $\mathbf{X}_v$ is a well-constrained subset of $\mathbf{C}_v$, and since $u \in \text{var}(\mathbf{X})$, u cannot be in s, a contradiction. Hence, $v \in \perp(\mathbf{C}_u)$. □

Lemma 5.55: If $\mathbf{C}$ is a set of equalities, $v \in \text{var}(\perp\mathbf{C})$, and s is a reduced set for $\mathbf{C}_v$, then $s \cup \{v\}$ is a reduced set for $\mathbf{C}$.

Proof. We prove this by induction on $|\text{var}(\mathbf{C})|$. First we note that since $\text{var}(\mathbf{C}) \neq \emptyset$, then $\text{var}(\mathbf{C}) \geq 2$, because if $|\text{var}(\mathbf{C})| = 1$, then $\perp\mathbf{C} = \emptyset$ and v could not be in $\text{var}(\perp\mathbf{C})$.

Base case: If $|\text{var}(\mathbf{C})| = 2$, $\text{var}(\mathbf{C}) = \{u, v\}$ for some u. Suppose $|\text{var}(\mathbf{E})| = 1$ for some $\mathbf{E} \in \mathbf{C}$; then $\{\mathbf{E}\}$ is minimally well-constrained, so that $\perp\mathbf{C} = \perp(\mathbf{C}_{\text{var}(\mathbf{E})})$. But $|\text{var}(\mathbf{E}')| = 1$ for every $\mathbf{E}' \in \mathbf{C}_{\text{var}(\mathbf{E})}$, so that $\perp(\mathbf{C}_{\text{var}(\mathbf{E})}) = \emptyset$ contrary to the fact that $v \in \text{var}(\perp\mathbf{C})$. Hence $|\text{var}(\mathbf{E})| = 2$ for every $\mathbf{E} \in \mathbf{C}$. Clearly $|\mathbf{C}| = 1$ since otherwise $\mathbf{C}$ has a minimally well-constrained subset $\mathbf{X}$ such that $|\text{var}(\mathbf{X})| = 2$ and $\perp\mathbf{C} = \emptyset$ contrary to the fact that $v \in \text{var}(\perp\mathbf{C})$. Hence $\mathbf{C}$ consists of a single equality, $\mathbf{C} = \perp\mathbf{C}$, $\emptyset$ is a reduced set for $(\perp\mathbf{C})_v$, and $\{v\}$ is a reduced set for $\mathbf{C}$.

Assumption: Assume that the result holds for any set of equalities containing less than n variables, where $n > 2$.

Induction: Assume $|\text{var}(\mathbf{C})| = n$. Since s is a reduced set for $\mathbf{C}_v$, by Lemma 5.50, for each $u \in s$, $s - \{u\}$ is a reduced set for $(\mathbf{C}_v)_u = (\mathbf{C}_u)_v$. Since $v \in \text{var}(\perp\mathbf{C})$, s is a reduced set for $\mathbf{C}_v$, and $u \in s$, by Lemma 5.54, $v \in \text{var}(\perp(\mathbf{C}_u))$. Since $|\text{var}(\mathbf{C}_u)| < n$, $v \in \text{var}(\perp(\mathbf{C}_u))$, and $s - \{u\}$ is a reduced set for $(\mathbf{C}_u)_v$, by the induction hypothesis $(s - \{u\}) \cup \{v\}$ is a reduced set for $\mathbf{C}_u$ and by Lemma 5.48 for $\perp(\mathbf{C}_u)$. Since $u \in s$ and $s \subseteq \text{var}(\perp(\mathbf{C}_v))$ (because s is a reduced set for $\mathbf{C}_v$), $u \in \text{var}(\perp(\mathbf{C}_v))$.

But $\mathrm{var}(\perp(\mathbf{C}_v))\subseteq\mathrm{var}(\perp\mathbf{C})$(by Lemma 5.53 since $v\in\mathrm{var}(\perp\mathbf{C})$), therefore, $u\in\mathrm{var}(\perp\mathbf{C})$. Since $u\in\mathrm{var}(\perp\mathbf{C})$, by Lemma 5.49, $\perp(\mathbf{C}_u)=\perp((\perp\mathbf{C})_u)$. Therefore, $(s-\{u\})\cup\{v\}$ is a reduced set for $\perp((\perp\mathbf{C})_u)$ and, by Lemma 5.48, for $(\perp\mathbf{C})_u$, establishing that$((s-\{u\})\cup\{v\})\cup\{u\}=s\cup\{v\}$ is a reduced set for **C**. □

Lemma 5.56: If **C** is a set of equalities then **C** has a reduced set.

Proof. We prove this by induction on $|\mathrm{var}(\mathbf{C})|$.

Base case: If $|\mathrm{var}(\mathbf{C})|=1$, then **C** has at least one minimally well-constrained subset and $\perp\mathbf{C}=\varnothing$. Therefore, $\varnothing$ is a reduced set for **C**.

Assumption: Assume that every set of equalities with less than $n>1$ variables has a reduced set.

Induction: Assume $|\mathrm{var}(\mathbf{C})|=n$. If $\perp\mathbf{C}=\varnothing$, then $\varnothing$ is a reduced set for **C**.

Now assume that $\perp\mathbf{C}\neq\varnothing$, then there exists a variable $v\in\mathrm{var}(\perp\mathbf{C})$ and because $|\mathrm{var}(\mathbf{C}_v)|<n$, by the induction hypothesis, there exists s a reduced set for $\mathbf{C}_v$. Since $v\in\mathrm{var}(\perp\mathbf{C})$ and s is a reduced set for $\mathbf{C}_v$, by Lemma 5.55, $s\cup\{v\}$ is a reduced set for **C**. □

Lemma 5.57: If **C** is a set of equalities, then a variable in $\mathrm{var}(\mathbf{C})$ is in a reduced set for **C** iff it does not occur in any well-constrained subset of **C**.

Proof. Let $v\in\mathrm{var}(\mathbf{C})$ and v be in a reduced set for **C**, then $v\in\mathrm{var}(\perp\mathbf{C})$ and v does not occur in any well-constrained subset of **C** since otherwise, by Lemma 5.36 v could not be in $\mathrm{var}(\perp\mathbf{C})$.

Now assume that $v\in\mathrm{var}(\mathbf{C})$ and v does not occur in any well-constrained subset of **C**. Then $v\in\mathrm{var}(\perp\mathbf{C})$ since otherwise, by Lemma 5.36 v would have been in a well-constrained subset of **C**. By Lemma 5.56 there exists a reduced set s for $(\perp\mathbf{C})_v$. Since $v\in\mathrm{var}(\perp\mathbf{C})=\mathrm{var}(\perp(\perp\mathbf{C}))$ and s is a reduced set for $(\perp\mathbf{C})_v$, $s\cup\{v\}$ is a reduced set for $\perp\mathbf{C}$ by Lemma 5.55, and for **C** by Lemma 5.48. Therefore, v is in a reduced set for **C**. □

Lemma 5.58: If **C** is a set of equalities with no over-constrained subset and $\perp$**C**=$\emptyset$, then **C** is well-constrained.

Proof. We prove this by induction on |**C**|.

Base case: If |**C**|=1, since $\perp$**C**=$\emptyset$, **C** has exactly one minimally well-constrained subset. Therefore, **C** is well-constrained.

Assumption: Assume that for every set of equalities with less than n>1 variables the result holds.

Induction: Assume |**C**|=n. Since $\perp$**C**=$\emptyset$, **C** has a minimally well-constrained subset **X** and by Lemma 5.34, $\perp$($\mathbf{C}_{\mathrm{var}(\mathbf{X})}$)=$\perp$**C**=$\emptyset$. Since $\perp$($\mathbf{C}_{\mathrm{var}(\mathbf{X})}$)=$\emptyset$ and |$\mathbf{C}_{\mathrm{var}(\mathbf{X})}$|<$n$, by the induction hypothesis, $\mathbf{C}_{\mathrm{var}(\mathbf{X})}$ is well-constrained. Since **X** and $\mathbf{C}_{\mathrm{var}(\mathbf{X})}$ are well-constrained, |var(**X**)|+|var($\mathbf{C}_{\mathrm{var}(\mathbf{X})}$)|=|**X**|+|$\mathbf{C}_{\mathrm{var}(\mathbf{X})}$|. But |var(**X**)|+|var($\mathbf{C}_{\mathrm{var}(\mathbf{X})}$)|=|var(**C**)| and |**X**|+|$\mathbf{C}_{\mathrm{var}(\mathbf{X})}$|=|**C**| so that |var(**C**)|=|**C**|, therefore, **C** is well-constrained. □

Lemma 5.59: If **C** is a set of equalities, then $\perp$**C** has no over-constrained subset.

Proof. If $\perp$**C** has an over-constrained subset, then by Lemma 5.28 $\perp$**C** must have at least one well-constrained subset, which it does not. □

Lemma 5.60: All reduced sets for a set of equalities are the same size.

Proof. Let **C** be a set of equalities. By Lemma 5.48, a reduced set for **C** is a reduced set for $\perp$**C**.

Suppose $\perp$**C**=$\emptyset$, then $\emptyset$ is the only reduced set for $\perp$**C**, therefore, the result holds.

Now suppose $\perp$**C**$\neq\emptyset$, since $\perp$($\perp$**C**)=$\perp$**C**$\neq\emptyset$, there exists a reduced set $s\neq\emptyset$ for $\perp$**C**, and by Lemma 5.50, $s-s=\emptyset$ is a reduced set for $(\perp\mathbf{C})_s$, therefore, $\perp((\perp\mathbf{C})_s)=\emptyset$. Since $\perp$**C** has no over-constrained subset (Lemma 5.59) and by Lemma 5.52 no subset of s is strictly constrained in $\perp$**C**, $(\perp\mathbf{C})_s$ has no over-constrained subset. So since $\perp((\perp\mathbf{C})_s)=\emptyset$, by Lemma 5.58 $(\perp\mathbf{C})_s$ is well-constrained, so that $|\mathrm{var}((\perp\mathbf{C})_s)|=|(\perp\mathbf{C})_s|$. Clearly, $|s|=|\mathrm{var}(\perp\mathbf{C})|-|\mathrm{var}((\perp\mathbf{C})_s)|$ and since no subset of s is strictly constrained in $\perp\mathbf{C}$ (Lemma 5.52) $|(\perp\mathbf{C})_s|=|\perp\mathbf{C}|$, therefore, $|s|=|\text{var}(\perp\mathbf{C})|-|\perp\mathbf{C}|$. Therefore, all reduced sets for **C** are the same size. □

The above lemma not only shows that any two reduced sets for a set of equalities are the same size, but also allows us to compute the size.

Now we present the most important property of reduced sets.

Corollary 5.61: If **C** is a set of equalities and s is a reduced set for **C**, then no subset of s except s can be a reduced set for **C**.

In the following, we extend the definition of a reduced set to apply to sets of constraints that include formulae that are not equalities.

Definition 5.62: If **C** is a set of constraints, not all of which are equalities, then s is said to be a *reduced set for C* iff $s = x \cup (\text{var}(\mathbf{C})-\text{var}(\mathbf{C}^{=}))$ and x is a reduced set for $\mathbf{C}^{=}$.

This definition ensures the minimality of $|s|$ since $\text{var}(\mathbf{C})-\text{var}(\mathbf{C}^{=})$ is constant, and $|x|$ is minimal.

Definition 5.63: If **C** is a set of constraints and $v \subseteq \text{var}(\mathbf{C})$, then s is said to be a *reduced set for C with respect to v* iff $s = v \cup u$ and u is a reduced set for $\mathbf{C}_v$.

Since $|u|$ is minimal and $|v|$ is constant, this definition also ensures the minimality of $|s|$.

Lemma 5.64: If **C** is a set of equalities and s is a reduced set for **C** with respect to v, then $s-v$ is a reduced set for $\mathbf{C}_v$.

Proof. Since s is a reduced set for **C** with respect to v, there exists u a reduced set for $\mathbf{C}_v$ such that $s=v \cup u$. Because u is a reduced set for $\mathbf{C}_v$, u is a subset of $\text{var}(\perp(\mathbf{C}_v))$ so that $v \cap u=\emptyset$ and $s-v=u$. Therefore, $s-v$ is a reduced set for $\mathbf{C}_v$. □

5.4.3 Reducing Solids

Having defined reduced sets and discussed some of their important properties, in this section we focus on applying the notion of reduced sets to solids. As suggested earlier, the definition of a reduced set for solids relies on the notion of factoring.

Definition 5.65: If Φ is a solid with a factoring $(x,y,z,\mathbf{C},\Psi)$, $\mathbf{N}$ is a set of interfaces exposed by Φ, and v is the set of variables required by the interfaces in $\mathbf{N}$, then Φ *is said to be reduced with respect to* $\mathbf{C}$ *and* N iff x is a reduced set for $\mathbf{C}$ with respect to v.

Lemma 5.66: If Φ is a solid with factoring $(x,y,z,\mathbf{C},\Psi)$ exposing an interface ϕ, and Δ is a solid equivalent to Φ with a factoring $(x',y',z,\mathbf{C},\Psi)$ such that $x'\bullet y'$ is a permutation of $x\bullet y$ and the set of variables required by ϕ is a subset of x', then Δ exposes an interface δ equivalent to ϕ.

Proof. Since $(x,y,z,\mathbf{C},\Psi)$ is a factoring of Φ, for each $u \in \text{scope}(\Phi)$ there exist sequences of values v and t such that $\mathbf{C}_{x,y,z}[u,v,t]$ is true and since Φ and Δ are equivalent, there exists $u' \in \text{scope}(\Delta)$ such that $\Phi(u)=\Delta(u')$. Let $s=x\bullet y$, $s'=x'\bullet y'$, and π be the permutation function such that $s'_i=s_{\pi(i)}$ for all i and let $u'\bullet v'=w'$, and $u\bullet v=w$ where $w'_i=w_{\pi(i)}$. Then, $\mathbf{C}_{x',y',z}[u',v',t]=\mathbf{C}_{x,y,z}[u,v,t]$ is true.

Since Φ exposes ϕ, there exists a sequence $p=p_1,\ldots,p_k$ the elements of which are seleceted from those in x such that for all $u \in \text{scope}(\Phi)$, $\phi(u)_i = u_{p_i}$ for each i $(1 \leq i \leq k)$. Let δ be the partial function defined by $\delta(u')_i = [u' \in \text{scope}(\Delta)]\ u'_{\pi(p_i)}$. Clearly, for each $u \in \text{scope}(\Phi)$ there exists $u' \in \text{scope}(\Delta)$ such that $\Phi(u) = \Delta(u')$ and $\delta(u')_i = \phi(u)_i$. Therefore, δ is an interface equivalent to ϕ exposed by Δ. □

We will make an observation about the list of variables identified as z in Definition 5.15. Every variable v in z has exactly one occurrence in a unique equality $\mathbf{E}$ in $\mathbf{C}$, such that $\text{var}(\mathbf{E})=\{v,u\}$ where v is a variable in $x\bullet y$. Equalities involving variables in z indicate that values passed to Ψ are selected from the list of values that satisfy $\mathbf{C}$. We can remove the equalities involving variables in z from $\mathbf{C}$ to make it simpler. The following lemma shows that removing from $\mathbf{C}$ those equalities involving variables in z will have no effect on a reduced set for a solid.

This lemma takes into consideration the fact that variables in z cannot appear in interfaces exposed by the solid.

Lemma 5.67: If $\mathbf{C}$ is a set of equalities, $z \subseteq \text{var}(\mathbf{C})$, $w \subseteq \text{var}(\mathbf{C})-z$, and for every $x \in z$ there exists $\mathbf{E} \in \mathbf{C}$ such that $x \in \text{var}(\mathbf{E})$, $x \notin \text{var}(\mathbf{E}')$ for all $\mathbf{E}' \in \mathbf{C}-\{\mathbf{E}\}$, $|\text{var}(\mathbf{E})| \leq 2$, and $\text{var}(\mathbf{E}) \cap z=\{x\}$, then there exists u, a reduced set for $\mathbf{C}$ with respect to w, such that $u \cap z = \varnothing$.

Proof. Let s be a reduced set for $\mathbf{C}$ with respect to w, $x \in z$, and $\mathbf{E}$ be the equality in $\mathbf{C}$ such that $\text{var}(\mathbf{E}) \cap z=\{x\}$.

Assume $\text{var}(\mathbf{E})=\{x\}$. Since $x \notin w$ and $\{\mathbf{E}\}_w$ is minimally well-constrained in $\mathbf{C}_w$, $x \notin s$.

Now assume $\text{var}(\mathbf{E})=\{x,y\}$. If $y \in w$, then $x \notin s$ because $\{\mathbf{E}\}_w$ is minimally well-constrained in $\mathbf{C}_w$. If $y \notin w$ and $x \in s$, since by Lemma 5.64 $s-w$ is a reduced set for $\mathbf{C}_w$, $(s-w)-\{x\}$ is a reduced set for $(\perp(\mathbf{C}_w))_x$, and by Lemma 5.48 for $\perp((\perp(\mathbf{C}_w))_x)$. Since $x \in \text{var}(\perp(\mathbf{C}_w))$ (otherwise x could not be in $s-w$), by Lemma 5.49, $\perp((\perp(\mathbf{C}_w))_x)=\perp((\mathbf{C}_w)_x)$ so that $(s-w)-\{x\}$ is a reduced set for $\perp((\mathbf{C}_w)_x)$. Since $\{\mathbf{E}\}_x$ is a minimally well-constrained subset of $(\mathbf{C}_w)_x$, by Lemma 5.34 and Lemma 5.35, $\perp((\mathbf{C}_w)_x)=\perp(((\mathbf{C}_w)_x)_{\text{var}(\{\mathbf{E}\}_x)})=\perp(((\mathbf{C}_w)_x)_y)$. Since $\{\mathbf{E}\}_y$ is a minimally well-constrained subset of $(\mathbf{C}_w)_y$, by Lemma 5.34 and Lemma 5.35, $\perp((\mathbf{C}_w)_y)=\perp(((\mathbf{C}_w)_y)_{\text{var}(\{\mathbf{E}\}_y)})=\perp(((\mathbf{C}_w)_y)_x)$. Hence, since $\perp(((\mathbf{C}_w)_x)_y)=\perp(((\mathbf{C}_w)_y)_x)$ we have $\perp((\mathbf{C}_w)_x)=\perp((\mathbf{C}_w)_y)$. Therefore, $(s-w)-\{x\}$ is a reduced set for $(\mathbf{C}_w)_y$. Because $x \in s-w$ and $s-w$ is a reduced set for $\mathbf{C}_w$, $x \in \text{var}(\perp(\mathbf{C}_w))$ and $\{\mathbf{E}\}_w$ is not minimally well-constrained in $\mathbf{C}_w$ so that $y \in \text{var}(\perp(\mathbf{C}_w))$. Since $(s-w)-\{x\}$ is a reduced set for $(\mathbf{C}_w)_y$ and $y \in \text{var}(\perp(\mathbf{C}_w))$, by Lemma 5.56, $((s-w)-\{x\}) \cup \{y\}$ is a reduced set for $\mathbf{C}_w$ and $u=(s-\{x\}) \cup \{y\}$ is a reduced set for $\mathbf{C}$. □

In the following, when we present a constraint $\mathbf{C}$, we might omit equalities involving variables in z and talk only in terms of constraints involving variables in x and y. Clearly, the practical implication of this simplification is that fewer equalities are subjected to our reduction algorithm without any changes to the outcome of the algorithm.

Lemma 5.68: If Φ is a solid in a design space $\mathcal{D}$ with factoring $(x,y,z,\mathbf{C},\Psi)$, and Φ exposes a set of interfaces $\mathbf{N}$, there exists a solid Δ equivalent to Φ which exposes a set of interfaces $\mathbf{M}$

equivalent to those in $\mathbf{N}$, and has a factoring $(u,v,t,\mathrm{X},\Omega)$ such that Δ is reduced with respect to X and $\mathbf{M}$.

Proof. We first define Δ in terms of Ψ, then show that there is a factoring $(u,v,z,\mathbf{C},\Psi)$ of Δ, prove that Φ and Δ are equivalent, and at the end show that Δ exposes a set of interfaces $\mathbf{M}$ equivalent to those in $\mathbf{N}$.

Let w be the set of variables required by the interfaces in $\mathbf{N}$, x' a reduced set for $\mathbf{C}^{=}$ with respect to w where $x'\cap z=\varnothing$ (Lemma 5.67), y'=var($\mathbf{C}$)–($x'\cup z$), and Δ the partial function defined by $\Delta(x')=[\mathrm{C}]\ \Psi(z)$. Let $n=|x|$, $m=|y|$, $t=|x'|$, $e=|y'|$, and $k=|z|$. Then Δ is a solid in $\mathcal{D}$ in t variables.

Now we will show that $(u,v,z,\mathbf{C},\Psi)$ is a factoring of Δ.

(***a***): $x'\cap z=\varnothing$ and y'=var($\mathbf{C}$)–($x'\cup z$), therefore x', y', and z are disjoint.

(***b***): Clearly var($\mathbf{C}$)=$x'\bullet y'\bullet z$.

(***c***): Since $(x,y,z,\mathbf{C},\Psi)$ is a factoring of Φ then for each $p\in z$, p occurs in a unique equality $\mathrm{E}\in\mathbf{C}$ such that var(E)=$\{p\}$ or var(E)=$\{p,q\}$ and $q\in x\cup y$. Since $x'\cup y'=x\cup y$, for each $p\in z$, p occurs in a unique equality $\mathrm{E}\in\mathbf{C}$ such that var(E)=$\{p\}$ or var(E)=$\{p,q\}$ and $q\in x'\cup y'$.

(***d***): Ψ is a solid in $\mathcal{D}$.

It remains to show that the validity, conformity and identity conditions are satisfied.

(***Validity***): $\forall a\in\mathbf{R}^t\ a\in$ scope(Δ), then by the definition of Δ, there exist $b\in\mathbf{R}^e$ and $c\in\mathbf{R}^k$ such that $\mathbf{C}_{x',y',z}[a,b,c]$ is true.

(***Conformity***): $\forall a\in\mathbf{R}^t$, $\forall b\in\mathbf{R}^e$,$\forall c\in\mathbf{R}^k$ if $\mathbf{C}_{x',y',z}[a,b,c]$ is true and $a\in$scope(Δ), then by the definition of Δ, $c\in$ scope(Ψ).

$\forall a\in\mathbf{R}^t$, $\forall b\in\mathbf{R}^e$,$\forall c\in\mathbf{R}^k$ if $\mathbf{C}_{x',y',z}[a,b,c]$ is true and $c\in$ scope(Ψ), then by the definition of Δ, $a\in$ scope(Δ).

(***Identity***): $\forall a\in\mathbf{R}^t$, $\forall b\in\mathbf{R}^e$,$\forall c\in\mathbf{R}^k$ if $a\in$scope(Δ) and $\mathbf{C}_{x',y',z}[a,b,c]$ is true, by the conformity condition, $c\in$scope(Ψ), and by the definition of Δ, $\Delta(a)=\Psi(c)$.

Therefore, $(x',y',z,\mathbf{C},\Psi)$ is a factoring of Δ.

Now we show that Δ and Φ are equivalent.

$\forall a \in \mathbf{R}^n$, if $a \in \text{scope}(\Phi)$, since $(x,y,z,\mathbf{C},\Psi)$ is a factoring of Φ, by the validity condition there exist $b \in \mathbf{R}^m$, $c \in \mathbf{R}^k$ such that $\mathbf{C}_{x,y,z}[a,b,c]$ is true, and by the conformity and identity conditions $c \in \text{scope}(\Psi)$ and $\Phi(a)=\Psi(c)$. **(I)**

Since $\mathbf{C}_{x,y,z}[a,b,c]$ is true, there exist $a' \in \mathbf{R}^t$, $b' \in \mathbf{R}^e$ such that $\mathbf{C}_{x',y',z}[a',b',t]$ is true. Since $(x',y',z,\mathbf{C},\Psi)$ is a factoring of Δ, $\mathbf{C}_{x',y',z}[a',b',c]$ is true, and $c \in \text{scope}(\Psi)$, by the conformity and identity conditions, $a' \in \text{scope}(\Delta)$ and $\Delta(a')=\Psi(c)$. **(II)**

From **(I)** and **(II)** we conclude that $\forall a \in \mathbf{R}^n$, if $a \in \text{scope}(\Phi)$, there exists $a' \in \mathbf{R}^t$ such that $\Phi(a)=\Delta(a')$. **(A)**

$\forall a' \in \mathbf{R}^t$, if $a' \in \text{scope}(\Delta)$, since $(x',y',z,\mathbf{C},\Psi)$ is a factoring of Δ, by the validity condition there exist $b' \in \mathbf{R}^m$, $c \in \mathbf{R}^k$ such that $\mathbf{C}_{x',y',z}[a',b',c]$ is true, and by the conformity and identity conditions $c \in \text{scope}(\Psi)$ and $\Delta(a')=\Psi(c)$. **(III)**

Since $\mathbf{C}_{x',y',z}[a',b',c]$ is true, there exist $a \in \mathbf{R}^t$, $b \in \mathbf{R}^e$ such that $\mathbf{C}_{x,y,z}[a,b,t]$ is true. Since $(x,y,z,\mathbf{C},\Psi)$ is a factoring of Φ, $\mathbf{C}_{x,y,z}[a,b,c]$ is true, and $c \in \text{scope}(\Psi)$, by the conformity and identity conditions, $a \in \text{scope}(\Phi)$ and $\Phi(a)=\Psi(c)$. **(IV)**

From **(III)** and **(IV)** we conclude that $\forall a' \in \mathbf{R}^t$, if $a' \in \text{scope}(\Delta)$, there exist an $a \in \mathbf{R}^n$ such that $\Delta(a')=\Phi(a)$. **(B)**

(A) and **(B)** establish that Φ and Δ have identical images and therefore are equivalent.

Now it remains to show that Δ exposes a set of interfaces **M** equivalent to those in **N**.

Since Φ is a solid with a factoring $(x,y,z,\mathbf{C},\Psi)$ exposing interfaces in **N**, and Δ is equivalent to Φ with a factoring $(x',y',z,\mathbf{C},\Psi)$ such that x' includes all the variables required by the interfaces in **N**, Δ exposes a set of interfaces **M** equivalent to those in **N** (Lemma 5.66). □

Lemma 5.69: If $(x,y,z,\mathbf{C},\Psi)$ is a factoring of a solid Φ, **C** contains only equalities, **N** is a set of interfaces exposed by Φ, and Φ is reduced with respect to **C** and **N**, then x is a parameter set for Φ with **N**.

Proof. Since Φ is reduced with respect to **C** and **N**, then x is a reduced set for **C** with respect to v, where v is the set of variables required by the interfaces in **N**. Assume $x=v\cup s$, where s is a reduced set for $\mathbf{C}_v$. Since $\mathbf{C}_v$ contains only equalities, no subset of s can be a reduced set for $\mathbf{C}_v$ and every reduced set for **C** with respect to v requires the variables in v, therefore, no subset of x can be a reduced set for **C** with respect to v. Furthermore, x is a sufficient set for Φ, establishing that x is a parameter set for Φ. □

In the following examples, we use the term **arm** to refer to any solid equivalent to **Arm** from Example 5.27.

Example 5.70: Recall that **Arm** in Example 5.27 was defined over the 9 variables b, c, d, e, f, g, a, l, and r. These 9 variables constrained by **C** which is the conjunctions of the four equalities in Example 5.27 along with five other equalities $z_1=b$, $z_2=c$, $z_3=d$, $z_4=e$, $z_5=r$ can be used to build solids equivalent to **Arm** with different factorings. For example, $\mathbf{Arm_1}(b,f,g,a,r)$, $\mathbf{Arm_2}(b,c,d,a,r)$, and $\mathbf{Arm_3}(b,c,f,g,a)$ are three solids equivalent to **Arm**, each reduced with respect to **C** and $\varnothing$ with the factorings $((b,f,g,a,r),(c,d,e,l),(z_1,z_2,z_3,z_4,z_5),\mathbf{C},\Phi)$, $((b,c,d,a,r),(e,f,g,l),(z_1,z_2,z_3,z_4,z_5),\mathbf{C},\Phi)$, and $((b,c,f,g,a),(d,e,l,r),(z_1,z_2,z_3,z_4,z_5),\mathbf{C},\Phi)$, respectively. Note that $\{b,f,g,a,r\}$, $\{b,c,d,a,r\}$, and $\{b,c,f,g,a\}$ are reduced sets for **C**. Later in this chapter, we will present an algorithm that generates all reduced sets for a given solid.

The constraint set in the factoring of **Arm** in Example 5.70 contained only equalities and the set of required interfaces was $\varnothing$. However, as we mentioned earlier, sometimes the application of an operation to its operands also provides an opportunity for reducing the resulting solid. In such cases, the solid may still be required to expose certain interfaces.

Example 5.71: Let **Joint** be the operation $((o_1,o_2),(p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8,\beta),\mathrm{F}_1\cup\mathrm{F}_2,\{\mathbf{hinge}, \mathbf{pivot}\},\mathbf{align})$, where $\mathbf{align}=\{\, p_1=p_5\,,\, p_2=p_6\,,\, p_4=p_8\,,\, \beta=p_7-p_3\,,\, -90\leq\beta\leq 90\,\}$, $\mathbf{hinge}(o_1,p_1,p_2,p_3,p_4)$

is true iff o_1 has a rotative junction with radius p_4 and hinge at (p_1,p_2), and **pivot**(o_2,p_5,p_6,p_7,p_8) is true iff o_2 has a rotative junction with radius p_8 and pivot point at (p_5,p_6). Note that the two **arm** operands of **Joint** are required to expose two different interfaces, so an appropriate choice for the first operand is **Arm₁** which exposes its hinge, required by the selector **hinge**, while an appropriate choice for the second operand is **Arm₂** which exposes its pivot point, required by the selector **pivot**. **Joint** constrains the two **arms** so that the hinge of the first coincides with the pivot point of the second as illustrated in Figure 5.7. The product solid named **Partial-Robot** resulting from the application of **Joint** to **Arm₁**(b,f,g,a,r) with factoring $((b_1,f_1,g_1,a_1,r_1),(c_1,d_1,e_1,l_1),(z_1,z_2,z_3,z_4,z_5),\mathbf{C}_{b,c,d,e,f,g,l,a,r}[b_1,c_1,d_1,e_1,f_1,g_1,l_1,a_1,r_1],\Phi)$ and **Arm₂**(b_2,c_2,d_2,a_2,r_2) with factoring $((b_2,c_2,d_2,a_2,r_2),(e_2,f_2,g_2,l_2),(z_6,z_7,z_8,z_9,z_{10}),\mathbf{C}_{b,c,d,e,f,g,l,a,r,z_1,z_2,z_3,z_4,z_5}[b_2,c_2,d_2,e_2,f_2,g_2,l_2,a_2,r_2,z_6,z_7,z_8,z_9,z_{10}],\Phi)$, has the factoring $(x,y,z,\mathbf{X},\mathbf{G})$ where

$$x = (b_1,f_1,g_1,a_1,r_1,b_2,c_2,d_2,a_2,r_2,\beta)$$

$$y = (c_1,d_1,e_1,l_1,e_2,f_2,g_2,l_2)$$

$$z = (z_1,z_2,z_3,z_4,z_5,z_6,z_7,z_8,z_9,z_{10})$$

and

$$\begin{aligned}\mathbf{X} = {} & \mathbf{C}_{b,c,d,e,f,g,l,a,r}[b_1,c_1,d_1,e_1,f_1,g_1,l_1,a_1,r_1] \cup \\ & \mathbf{C}_{b,c,d,e,f,g,l,a,r,z_1,z_2,z_3,z_4,z_5}[b_2,c_2,d_2,e_2,f_2,g_2,l_2,a_2,r_2,z_6,z_7,z_8,z_9,z_{10}] \cup \\ & \mathbf{align}_{p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8}[f_1,g_1,a_1,r_1,b_2,c_2,a_2,r_2]\end{aligned}$$

and $\mathbf{G}(z_1,z_2,z_3,z_4,z_5,z_6,z_7,z_8,z_9,z_{10})=\Phi(z_1,z_2,z_3,z_4,z_5)\cup\Phi(z_6,z_7,z_8,z_9,z_{10})$.

After the substitutions, X consists of the following literals

(I)	$d_1-b_1 = l_1.\cos(a_1)$	(II)	$e_1-c_1 = l_1.\sin(a_1)$
(III)	$f_1 = d_1+2r_1.\cos(a_1)$	(VI)	$g_1 = e_1+2r_1.\sin(a_1)$
(V)	$d_2-b_2 = l_2.\cos(a_2)$	(IV)	$e_2-c_2 = l_2.\sin(a_2)$
(VII)	$f_2 = d_2+2r_2.\cos(a_2)$	(VIII)	$g_2 = e_2+2r_2.\sin(a_2)$

(IX) $f_1 = b_2$ (X) $g_1 = c_2$

(XI) $\beta = a_2 - a_1$ (XII) $r_1 = r_2$

(XIII) $-90 \leq \beta \leq 90$

together with

$$z_1 = b_1\ ,\ z_2 = c_1\ ,\ z_3 = d_1\ ,\ z_4 = e_1\ ,\ z_5 = r_1\ ,\ z_6 = b_2\ ,\ z_7 = c_2\ ,\ z_8 = d_2\ ,\ z_9 = e_2\ ,\ z_{10} = r_2$$

Suppose we want to reduce the solid corresponding to this factoring in such a way that it exposes three interfaces, the pivot point of the first arm, the hinge of the second arm, and an interface including the angles a_1, a_2, and β. The set of variables required by these interfaces is $\{b_1,c_1,f_2,g_2,a_1,a_2,\beta\}$. **Reduced-Partial-Robot**$(b_1,c_1,a_1,r_1,a_2,f_2,g_2,\beta)$ is equivalent to **Partial-Robot** reduced with respect to X and $\{b_1,c_1,f_2,g_2,a_1,a_2,\beta\}$ and has the factoring

$$((b_1,c_1,a_1,r_1,f_2,g_2,a_2,\beta),(d_1,e_1,f_1,g_1,l_1,b_2,c_2,d_2,e_2,l_2,r_2),z),\mathbf{X},\mathbf{G}).$$

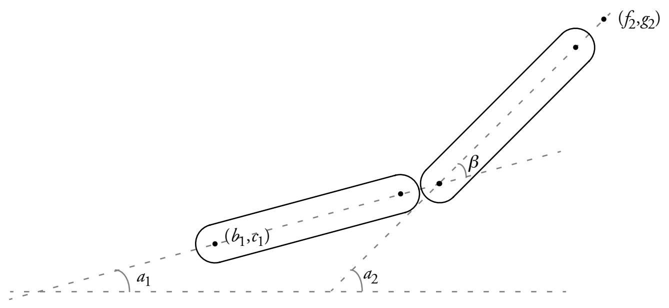


Figure 5.7: A **Partial-Robot** solid

5.4.4 Reduction Algorithm

Our reduction algorithm is an implementation of Definitions 5.62 and 5.63, and operates on an *equality constraint graph*, composed of *equality nodes*, each corresponding to an equality literal and represented by a circle with zero or more *terminals* around its perimeter. Each terminal represents the occurrence of a variable in the corresponding equality. Arcs connect all occurrences of a variable. A network of terminals interconnected by arcs is called an *occur graph*. Figure 5.8 illustrates the equality constraint graph of the solid **Arm** in Example 5.27. The four arcs forming a square along with the four terminals at its corners constitute the occur graph of variable *a*.

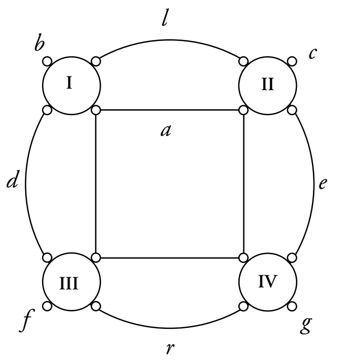


Figure 5.8: An equality constraint graph.

Since the reduction algorithm has been implemented, we will explain it by presenting and discussing the Prolog code of that implementation. A complete listing can be found in Appendix C.

The core of the algorithm is realised by the `reduce` predicate, which computes a reduced set for a constraint **C** composed only of equalities. This predicate has three arguments, a list representing var(**C**), the equality constraint graph for **C**, and a reduced set for **C** with respect to the empty set of interfaces. The equality constraint graph is represented as a list of nodes, each consisting of a list of the variables occurring in the corresponding equality.

```
reduce(VarC,C,[]):-
          trim(VarC,_,C,[]),!.
reduce(VarC,C,[V|Reduced]):-
          trim(VarC,VarTrimmedC,C,TrimmedC),
          next(V,VarTrimmedC,Rest),
          removeOccurGraph(V,TrimmedC,TrimmedCsubV),
          compress(Rest,VarPrunedC,TrimmedCsubV,PrunedC),
          reduce(VarPrunedC,PrunedC,Reduced).
```

The first clause of `reduce` realises the base case. It identifies the equality constraint graph as one which has an empty trimming, for which the only reduced set is the empty set. The second clause computes the trimming of the set of equalities by removing the occur graphs of all variables in minimally well-constrained sets, then picking a variable from those remaining as explained below, and removing its occur graph. If this succeeds, the equalities all variables of which have been identified as being in a minimally well-constrained set or in the reduced sets, are removed from the graph by the `compress` literal. Finally, the process is repeated on the pruned graph.

The trimming of a set of equalities is implemented by `trim` predicate which searches for a minimally well-constrained set in a given set of equalities, removes it from the set and repeats the process on the pruned set. This predicate could also use the result of the following lemma to improve the search for a minimally well-constrained set.

If **C** is a set of equalities, a variable is called *a solitary variable in* **C** when it occurs in exactly one equality in **C**.

Lemma 5.72: If **C** is a minimally well-constrained set of equalities and $|\mathbf{C}|>1$, then no variable in var(**C**) is a solitary variable in **C**.

Proof. Assume that there exists a variable $v\in\text{var}(\mathbf{C})$ such that v is solitary and occurs in some equality $\mathbf{E}\in\mathbf{C}$. Since **C** is minimally well-constrained $|\text{var}(\mathbf{C})|=|\mathbf{C}|$, and because $|\mathbf{C}|=|\mathbf{C}-\{\mathbf{E}\}|+1$ and $|\text{var}(\mathbf{C}-\{\mathbf{E}\})|<|\text{var}(\mathbf{C})|$ (since v occurs only in **E**), $|\text{var}(\mathbf{C}-\{\mathbf{E}\})|<|\mathbf{C}-\{\mathbf{E}\}|+1$ so that $|\text{var}(\mathbf{C}-\{\mathbf{E}\})|\leq|\mathbf{C}-\{\mathbf{E}\}|$. Therefore, since $|\mathbf{C}|>1$, $\mathbf{C}-\{\mathbf{E}\}$ is either well-constrained or over-constrained. In the former case, since $\mathbf{C}-\{\mathbf{E}\}$ is a subset of **C**, then **C** cannot be minimally well-constrained, contrary to our choice of **C**. In the latter case, since $\mathbf{C}-\{\mathbf{E}\}$ is over-constrained, by Lemma 5.28, $\mathbf{C}-\{\mathbf{E}\}$ has a well-constrained subset, contradicting the fact that **C** is minimally well-constrained. Therefore, no variable in var(**C**) is a solitary in **C**. □

Using the result of this lemma, `trim` could ignore any equality **E** with a solitary variable v unless $\text{var}(\mathbf{E})=\{v\}$.

The predicate `next` picks a variable from the list of variables and along with the selected element, returns the longest postfix of the list that trails the selected element. This stops the program from reporting permutations of a reduced set when we wish to find all reduced sets through forced backtracking.

The reader has probably noticed that the `reduce` predicate is in fact an implementation of the proof of Lemma 5.56 and therefore generates a reduced set for the set of equalities represented by the given equality graph. By means of forced backtracking, all possible reduced sets can be obtained. For example, the program returns 98 distinct reduced sets for the solid **Arm** in Example 5.27. Noticing that the equality constraint graph in Figure 5.8 has no well-constrained subset and using Lemma 5.60, we know that the reduced sets for this example must all have 9–4=5 variables. Out of 126 possible selections of 5 out of 9 variables, we know that 20 cannot be reduced sets because each has a subset which is strictly constrained in the example set, violating Lemma 5.52, for instance $\{b,d,a,l,r\}$ and $\{d,a,f,g,r\}$. There are also another 8 selections that violate Lemma 5.50 namely, $\{b,c,d,e,l\}$, $\{b,c,d,e,a\}$, $\{b,d,f,l,r\}$, $\{b,f,l,a,r\}$, $\{c,e,g,l,r\}$, $\{c,g,l,a,r\}$, $\{d,e,f,g,a\}$, and $\{d,e,f,g,r\}$. For example, $\varnothing$ is not a reduced set for the set of equalities obtained from the example set by substituting the variables in $\{b,c,d,e,l\}$ with values. This brings the total number of reduced sets for this example to 98 as reported by the program.

Having the core `reduce` predicate, as suggested earlier, the reduction algorithm uses the set of variables in the given interfaces to modify the equality constraint graph before passing it to `reduce` by removing every occur graph that corresponds to a variable which is in the set of interfaces and must therefore be included in the reduced set. For example, Figure 5.9 shows an equality constraint graph for solid **Partial-Robot** of Example 5.71 after it is modified with respect to the set of three required interfaces as explained in Example 5.71.

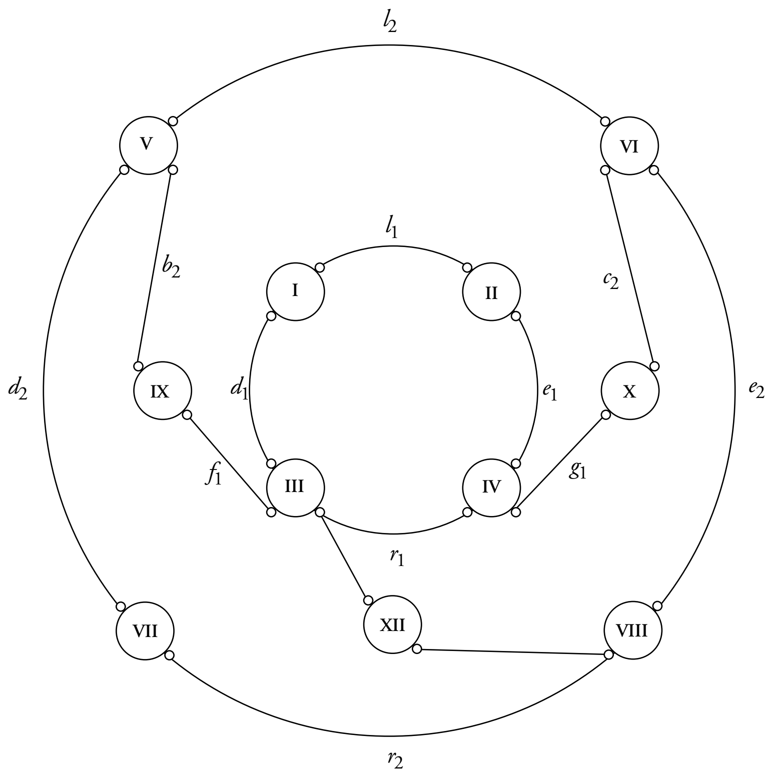


Figure 5.9: A simplified equality constraint graph.

The reduction algorithm is realised by the `reduceInterface` predicate the arguments to which are the lists of input and internal variables of the solid, a list of variables required by the interfaces, and an equality constraint graph represented by a list of nodes, each consisting of a list of the variables occurring in the equality corresponding to the node. The algorithm generates two new lists. The first corresponds to a reduced set for the solid with respect to the given interfaces and the other corresponds to the new set of internal variables of the reduced solid.

```
reduceInterface(X,Y,Required,C,XReduced,YExpanded):-
     removeRequired(Required,C,VarPrunedC,PrunedC),
     reduce(VarPrunedC,PrunedC,Reduced),
     reArrange(X,Y,Required,C,Reduced,XReduced,YExpanded).
```

To establish correctness, we observe that termination is assured because the size of the equality constraint graph is initially finite and is reduced through each iteration. Partial correctness follows from Lemma 5.56.

Example 5.73: In Example 5.71 we showed a reduced solid that exposed the pivot point and the hinge of **Partial-Robot** as well as three angles a_1, a_2, and β. The reduction algorithm returns 12 reduced sets. Each reduced set has 8 variables, 7 of them required by the interfaces described in Example 5.71, namely b_1, c_1, f_2, g_2, a_1, a_2, and β, while the eighth can be any of e_2, r_2, d_2, b_2, l_2, c_2, e_1, l_1, d_1, f_1, r_1, or g_1. This is not surprising as each of these 12 variables is a reduced set for the equality constraint graph shown in Figure 5.9 which includes 12 variables and 11 equality nodes, and contains no well-constrained subset (Lemma 5.60).

5.4.5 Reduction Algorithm Optimisation

The time complexity of the reduction algorithm is exponential with respect to the number of equality constraints in the set, which makes it less suitable in practice for sets with large numbers of equalities. The exponential complexity is due to the trimming at each recursion which in turn, in the worst case, calls for inspecting all subsets of the set of nodes in an equality constraint graph. Fortunately, some optimisation is possible, as the following lemma shows.

Lemma 5.74: If **C** is a set of equalities, **C** has no well-constrained subset, and $v \in \text{var}(\mathbf{C})$, then $\mathbf{C}_v$ has no over-constrained subset.

Proof. Assume $\mathbf{X} \subseteq \mathbf{C}$ and $\mathbf{X}_v$ is over-constrained, then $|\text{var}(\mathbf{X}_v)|<|\mathbf{X}_v|$. If $v \notin \text{var}(\mathbf{X})$, then $\mathbf{X}_v=\mathbf{X}$ which cannot be over-constrained because **C** has no well-constrained subset (Lemma 5.28). If $v \in \text{var}(\mathbf{X})$, then $|\text{var}(\mathbf{X})|=|\text{var}(\mathbf{X}_v)|+1$. Since **C** has no well-constrained subset, $\{v\}$ cannot be strictly constrained in **C**, therefore, $|\mathbf{X}|=|\mathbf{X}_v|$. Hence, $|\text{var}(\mathbf{X})|<|\mathbf{X}|+1$. But since **X** is not well-constrained, $|\mathbf{X}|<|\text{var}(\mathbf{X})|$ so that $|\mathbf{X}|<|\text{var}(\mathbf{X})|<|\mathbf{X}|+1$ which is impossible. Therefore, $\mathbf{X}_v$ cannot be over constrained. □

This lemma in conjunction with Lemma 5.60 is the cornerstone of our optimised reduction algorithm. Lemma 5.60 states that a reduced set for a trimming of a set of equalities is a reduced set for the original set of equalities and Lemma 5.74 indicates that the trimming of the set obtained from a set of equalities with no well-constrained subset by substituting the variables of one of its reduced sets with values will have no over-constrained subset.

To improve the complexity, the optimised algorithm eliminates the trimming step of the reduction algorithm. Instead, it relies on Lemma 5.74 as described below.

```
reduce(VarC,C,Original,[V|Reduced]):-
          next(V,VarC,Rest),
          removeOccurGraph(V,C,SimplifiedC),
          compress(VarC,VarPrunedC,SimplifiedC,PrunedC),
          intersect(Rest,VarPrunedC,Candidates),
          reduce(Candidates,PrunedC,Original,Reduced).
reduce(_,C,Original,[]):-
          notOverConstrained(C,Original).
```

The first clause of `reduce` selects a variable in the set as a candidate. The variable will be rejected if the removal of its occur graph would result in a set of equalities with an over-constrained subset violating the result of Lemma 5.74.

```
compress(_,_,C,_):-
          element([],C,_),!,fail.
compress(VarC,VarPrunedC,C,PrunedC):-
          element([V],C,Rest),!,
          removeOccurGraph(V,Rest,Rem),
          remove(V,VarC,Temp),
          compress(Temp,VarPrunedC,Rem,PrunedC),!.
compress(VarC,VarC,C,C).
```

The first clause of `compress` checks if removing a variable would leave an equality with no variables. Since the search for minimally well-constrained subsets of a set of equalities with single variables could be achieved by a single pass through the set, the second clause of `compress` finds and removes such minimally well-constrained subsets, and repeats the process on the simplified set. The last clause of `compress` returns the set of equalities when no over-constrained subset is detected and the set contains no minimally well-constrained subset with a single variable.

When the first clause of `reduce` fails to find any candidates, the second clause checks whether the set is a valid over-constrained set; that is, one that originally existed in the set rather than being produced by the removal of an occur graph. This is achieved by searching for over-constrained sets and comparing them against the original set. This is implemented by the `notOverConstrained` predicate as follows.

```
notOverConstrained([Eq|Rest],Original):-
          getGraph(Eq,Rest,C,Others),!,
          isValid([Eq|C],Original),
          notOverConstrained(Others,Original),!.
notOverConstrained([],_).


isValid(C,Original):-
          originalOverConstrainedSubSet(C,Original,Subset),
          simplify(Subset,C,Rest),!,
          isValid(Rest,Original).
isValid(C,_):-
          wellConstrained(C,_).
```

The best case occurs when a reduced set is found without iteration. This could happen when removing the occur graph of the first candidate variable results in an empty graph. There are many equality sets the structure of which could result in the best case. For example, if there are m equalities and $n=m+1$ variables, and the equality constraint graph forms a chain of length m with one variable shared between consecutive nodes in the chain. In this case, the time complexity of the algorithm is $O(m)$ which is the time required to propagate the removal of variables from the graph through the iteration of the `compress` predicate.

The worst case occurs when there are m equality constraints over n variables, every variable occurs in every constraint in the set, and $m > n$. This will cause the reduction algorithm to search the entire space for a combination of variables for a reduced set to no effect. At each level, each candidate for a reduced set will be tried, and only after the very last variable is tried will the algorithm find an inconsistency and backtrack. If we assume the algorithm has selected $k-1$ variables so far, for some $k < n$, and is about to choose the k^{th} candidate, removing the k^{th} variable will certainly neither instigate a failure nor remove any node from the graph, however removing the occur graph of a variable requires m iterations of the `removeOccurGraph` predicate.

A lower bound for the worst case is $\Omega(2^{n+1})$ since the algorithm must consider every size n subset of the variables, followed by a brute force search for over-constrained subsets, which in this case will take $\Omega(2^n)$ time. More precisely, the worst case complexity is characterised by the recurrence relation

$$T(n)=2T(n-1)+\mathbf{O}(m)$$

$$T(1)=\mathbf{O}(m)$$

therefore

$$T(n)=\mathbf{O}(m.2^n).$$

As stated earlier, the algorithm will be followed by a check for finding over-constrained subsets in $\mathbf{O}(2^n)$ time. Therefore the time complexity of the worst case is $\mathbf{O}(m.2^n)$.

Although the worst case complexity is still exponential, the optimisation becomes noticeable when the constraint set does not contain any over-constrained subset and the well-constrained subsets created as the result of selecting certain candidates contain small numbers of nodes compared to the total number of nodes in the set. The latter condition ensures that the simplified equality constraint graphs at each iteration will not invoke the worst case especially at the top level.

5.4.6 Incremental Reduction

In this section, we describe incremental reduction, a technique which makes the reduction of solids more practical.

The following lemmas provide the necessary foundations.

Lemma 5.75: If **C** and **D** are sets of equalities, and $\perp\mathbf{D}=\varnothing$, then $\perp(\mathbf{C}\cup\mathbf{D})=\perp(\mathbf{C}_{\mathrm{var}(\mathbf{D})})$.

Proof. We use induction on $|\mathbf{D}|$.

Base case: Assume $|\mathbf{D}|=0$, then $\mathbf{D}=\varnothing$ so the result holds vacuously.

Assumption: Assume that for some $n>0$ the result holds for any set of equalities **B** such that $|\mathbf{B}|<n$.

Induction: Suppose $|\mathbf{D}|=n$. Since $\mathbf{D}\neq\varnothing$ and $\perp\mathbf{D}=\varnothing$ there exists a subset **X** of **D** such that **X** is minimally well-constrained and $\perp(\mathbf{D}_{\mathrm{var}(\mathbf{X})})=\varnothing$. Since **X** is a subset of **D**, it is a subset of $\mathbf{C}\cup\mathbf{D}$ so that by Lemma 5.33, $\perp(\mathbf{C}\cup\mathbf{D})=\perp((\mathbf{C}\cup\mathbf{D})_{\mathrm{var}(\mathbf{X})})=\perp(\mathbf{C}_{\mathrm{var}(\mathbf{X})}\cup\mathbf{D}_{\mathrm{var}(\mathbf{X})})$. Since $\perp(\mathbf{D}_{\mathrm{var}(\mathbf{X})})=\varnothing$ and $|\mathbf{D}_{\mathrm{var}(\mathbf{X})}|<n$, by the induction hypothesis, $\perp(\mathbf{C}_{\mathrm{var}(\mathbf{X})}\cup\mathbf{D}_{\mathrm{var}(\mathbf{X})})=\perp((\mathbf{C}_{\mathrm{var}(\mathbf{X})})_{\mathrm{var}(\mathbf{D}_{\mathrm{var}(\mathbf{X})})})$. But $(\mathbf{C}_{\mathrm{var}(\mathbf{X})})_{\mathrm{var}(\mathbf{D}_{\mathrm{var}(\mathbf{X})})}=\mathbf{C}_{\mathrm{var}(\mathbf{D})}$. Therefore, $\perp(\mathbf{C}\cup\mathbf{D})=\perp(\mathbf{C}_{\mathrm{var}(\mathbf{D})})$. □

Lemma 5.76: If **C** is a set of equalities, s is a reduced set for **C**, y is a subset of s, and **Z** is a subset of **C** such that $\#(\mathbf{Z})=0$ and y is a subset of $\mathrm{var}(\mathbf{Z})$, then $|\mathrm{var}(\mathbf{Z})|-|\mathbf{Z}|\geq|y|$.

Proof. First we prove that $\mathbf{Z}_y$ is not over-constrained. Then use this result to show that $|\mathrm{var}(\mathbf{Z})|-|\mathbf{Z}|\geq|y|$. We prove by induction on $|y|$ that $\mathbf{Z}_y$ is not over-constrained.

Base case: Assume $|y|=0$, then because $\#(\mathbf{Z})=0$, $|\mathbf{Z}|<|\mathrm{var}(\mathbf{Z})|$ and $\mathbf{Z}_y=\mathbf{Z}$ is not over-constrained.

Assumption: Assume that for some $n>0$ the result holds for any subset z of s such that $|z|<n$.

Induction: Assume $|y|=n$. Suppose $\mathbf{Z}_y$ is over-constrained. Let $x\in y$. Since $y-\{x\}$ is a subset of s and $|y-\{x\}|<n$, by the induction hypothesis, $\mathbf{Z}_{y-\{x\}}$ is not over-constrained. Since $y-\{x\}\subseteq s$, by Lemma 5.50 $s-(y-\{x\})$ is a reduced set for $\mathbf{C}_{y-\{x\}}$ so that $x\in\mathrm{var}(\perp(\mathbf{C}_{y-\{x\}}))$, and by Lemma 5.36 $\mathbf{Z}_{y-\{x\}}$ is not well-constrained. Therefore, $\mathbf{Z}_{y-\{x\}}$ is under-constrained so that $|\mathbf{Z}_{y-\{x\}}|<|\mathrm{var}(\mathbf{Z}_{y-\{x\}})|$. Now $|\mathbf{Z}_{y-\{x\}}|=|\mathbf{Z}_y|$ since the subset $\{x\}$ of s cannot be strictly constrained in **C** by Lemma 5.52, $|\mathrm{var}(\mathbf{Z}_y)|<|\mathbf{Z}_y|$ since $\mathbf{Z}_y$ is over-constrained, and $|\mathrm{var}(\mathbf{Z}_{y-\{x\}})|=|\mathrm{var}(\mathbf{Z}_y)|+1$, so $|\mathrm{var}(\mathbf{Z}_y)|<|\mathbf{Z}_y|<|\mathrm{var}(\mathbf{Z}_y)|+1$ which is impossible. Therefore, $\mathbf{Z}_y$ is not over-constrained.

If $|\mathrm{var}(\mathbf{Z})|-|\mathbf{Z}|<|y|$, since $|\mathrm{var}(\mathbf{Z}_y)|=|\mathrm{var}(\mathbf{Z})|-|y|$, then $|\mathrm{var}(\mathbf{Z}_y)|<|\mathbf{Z}|$. Since no subset of s is strictly constrained in **C** (Lemma 5.52) no subset of y is strictly constrained in **Z** so that $|\mathbf{Z}|=|\mathbf{Z}_y|$ and as a result $|\mathrm{var}(\mathbf{Z}_y)|<|\mathbf{Z}_y|$ which is impossible because $\mathbf{Z}_y$ is not over-constrained. Therefore, $|\mathrm{var}(\mathbf{Z})|-|\mathbf{Z}|\geq|y|$. □

Lemma 5.77: If **C** and **D** are sets of equalities, s is a reduced set for **C**, v is a reduced set for **D**, and $\mathrm{var}(\mathbf{C})\cap\mathrm{var}(\mathbf{D})\subseteq s$, then for each $x\in v$, $x\in\mathrm{var}(\perp(\mathbf{C}\cup\mathbf{D}))$.

Proof. We use induction on $\#(\mathbf{C}\cup\mathbf{D})$.

Base case: If $\#(\mathbf{C}\cup\mathbf{D})=0$, then $\mathbf{C}\cup\mathbf{D}$ has no well-constrained subset so that, by Lemma 5.36, $x\in\mathrm{var}(\perp(\mathbf{C}\cup\mathbf{D}))$.

Assumption: Assume that for some $n>0$ the result holds for any sets of equalities **B** and **G** such that $\#(\mathbf{B}\cup\mathbf{G})<n$.

Induction: Assume $\#(\mathbf{C}\cup\mathbf{D})=n$, then there exists a minimally well-constrained subset **X** of $\mathbf{C}\cup\mathbf{D}$. Suppose $\mathbf{C}\cap\mathbf{D}\neq\varnothing$ then for each equality $\mathbf{E}\in\mathbf{C}\cap\mathbf{D}$, $\mathrm{var}(\mathbf{E})\subseteq\mathrm{var}(\mathbf{C})\cap\mathrm{var}(\mathbf{D})\subseteq s$ which violates the result of Lemma 5.52. Therefore, $\mathbf{C}\cap\mathbf{D}=\varnothing$. Now we show that if $\mathbf{X}=\mathbf{Y}\cup\mathbf{Z}$ such that $\mathbf{Y}\subseteq\mathbf{C}$ and $\mathbf{Z}\subseteq\mathbf{D}$, then either $\mathbf{Y}=\varnothing$ or $\mathbf{Z}=\varnothing$.

Since $\mathbf{Y}\subseteq\mathbf{C}$, s is a reduced set for **C**, $\mathrm{var}(\mathbf{Y})\cap\mathrm{var}(\mathbf{Z})\subseteq\mathrm{var}(\mathbf{Y})$, $\mathrm{var}(\mathbf{Y})\cap\mathrm{var}(\mathbf{Z})\subseteq s$ and $\#(\mathbf{Y})=0$ (otherwise **X** could not be minimally well-constrained), by Lemma 5.76 $|\mathrm{var}(\mathbf{Y})|-|\mathbf{Y}|\geq|\mathrm{var}(\mathbf{Y})\cap\mathrm{var}(\mathbf{Z})|$. Hence we have

$$
\begin{aligned}
|\mathbf{Z}| &\geq |\mathbf{X}|-|\mathbf{Y}| \\
&\geq |\mathbf{X}|+|\mathrm{var}(\mathbf{Y})\cap\mathrm{var}(\mathbf{Z})|-|\mathrm{var}(\mathbf{Y})| \text{ from the above} \\
&= |\mathrm{var}(\mathbf{X})|+|\mathrm{var}(\mathbf{Y})\cap\mathrm{var}(\mathbf{Z})|-|\mathrm{var}(\mathbf{Y})| \text{ since } \mathbf{X} \text{ is minimally well-constrained} \\
&= |\mathrm{var}(\mathbf{Y})|+|\mathrm{var}(\mathbf{Z})|-|\mathrm{var}(\mathbf{Y})\cap\mathrm{var}(\mathbf{Z})|+|\mathrm{var}(\mathbf{Y})\cap\mathrm{var}(\mathbf{Z})|-|\mathrm{var}(\mathbf{Y})| \\
&= |\mathrm{var}(\mathbf{Z})|
\end{aligned}
$$

so $\mathbf{Z}$ is not under-constrained, and therefore cannot be a proper subset of $\mathbf{X}$, since in that case $\mathbf{Z}$ would have to be under-constrained by Lemma 5.29. Therefore, either $\mathbf{Z}=\emptyset$ or $\mathbf{Z}=\mathbf{X}$, in which case $\mathbf{Y}=\emptyset$ since $\mathbf{Y}\cap\mathbf{Z}=\emptyset$ (because $\mathbf{C}\cap\mathbf{D}=\emptyset$).

If $\mathbf{Y}=\emptyset$, then $\mathbf{X}=\mathbf{Z}$ so that $\mathbf{X} \subseteq \mathbf{D}$ and $x \notin \text{var}(\mathbf{X})$ since otherwise x could not be in v by Lemma 5.57. Since v is a reduced set for $\mathbf{D}$, by Lemma 5.48, v is a reduced set for $\perp\mathbf{D}$ and, by Lemma 5.34, for $\perp(\mathbf{D}_{\text{var}(\mathbf{X})})$ and, by Lemma 5.48, for $\mathbf{D}_{\text{var}(\mathbf{X})}$. Since $\text{var}(\mathbf{C})\cap\text{var}(\mathbf{D})\subseteq s$, by Lemma 5.50 $s-(\text{var}(\mathbf{X})\cap(\text{var}(\mathbf{C})\cap\text{var}(\mathbf{D})))=s-(\text{var}(\mathbf{X}))$ is a reduced set for $\mathbf{C}_{\text{var}(\mathbf{X})\cap(\text{var}(\mathbf{C})\cap\text{var}(\mathbf{D}))}=\mathbf{C}_{\text{var}(\mathbf{X})}$. Also v is a reduced set for $\mathbf{D}_{\text{var}(\mathbf{X})}$, $\text{var}(\mathbf{C}_{\text{var}(\mathbf{X})})\cap\text{var}(\mathbf{D}_{\text{var}(\mathbf{X})})$ is a subset of $s-\text{var}(\mathbf{X})$, and $\#(\mathbf{C}_{\text{var}(\mathbf{X})}\cup\mathbf{D}_{\text{var}(\mathbf{X})})<n$. So by the induction hypothesis, $x \in\text{var}(\perp(\mathbf{C}_{\text{var}(\mathbf{X})}\cup\mathbf{D}_{\text{var}(\mathbf{X})}))$. But $\perp(\mathbf{C}_{\text{var}(\mathbf{X})}\cup\mathbf{D}_{\text{var}(\mathbf{X})})=\perp((\mathbf{C}\cup\mathbf{D})_{\text{var}(\mathbf{X})})$ and by Lemma 5.34, $\perp((\mathbf{C}\cup\mathbf{D})_{\text{var}(\mathbf{X})})=\perp(\mathbf{C}\cup\mathbf{D})$. Therefore, $x\in\text{var}(\perp(\mathbf{C}\cup\mathbf{D}))$.

If $\mathbf{Z}=\emptyset$, then $\mathbf{X}=\mathbf{Y}$, $\mathbf{X}\subseteq \mathbf{C}$, and $\text{var}(\mathbf{X})\cap\text{var}(\mathbf{D})=\text{var}(\mathbf{X})\cap(\text{var}(\mathbf{C})\cap\text{var}(\mathbf{D}))\subseteq\text{var}(\mathbf{X})\cap s=\emptyset$ by Lemma 5.36 (because each variable which is in $\text{var}(\mathbf{X})$ is not in $\text{var}(\perp\mathbf{C})$ and therefore cannot be in s), so that $x\notin\text{var}(\mathbf{X})$. Since s is a reduced set for $\mathbf{C}$, by Lemma 5.48, s is a reduced set for $\perp\mathbf{C}$ and, by Lemma 5.34, for $\perp(\mathbf{C}_{\text{var}(\mathbf{X})})$ and, by Lemma 5.48, for $\mathbf{C}_{\text{var}(\mathbf{X})}$. Since v is a reduced set for $\mathbf{D}=\mathbf{D}_{\text{var}(\mathbf{X})}$, s is a reduced set for $\mathbf{C}_{\text{var}(\mathbf{X})}$, $\text{var}(\mathbf{C}_{\text{var}(\mathbf{X})})\cap\text{var}(\mathbf{D}_{\text{var}(\mathbf{X})})=\text{var}(\mathbf{C})\cap\text{var}(\mathbf{D})$ is a subset of s, and $\#(\mathbf{C}_{\text{var}(\mathbf{X})}\cup\mathbf{D}_{\text{var}(\mathbf{X})})<n$, by the induction hypothesis, $x \in\text{var}(\perp(\mathbf{C}_{\text{var}(\mathbf{X})}\cup\mathbf{D}_{\text{var}(\mathbf{X})}))$. But $\perp(\mathbf{C}_{\text{var}(\mathbf{X})}\cup\mathbf{D}_{\text{var}(\mathbf{X})})=\perp((\mathbf{C}\cup\mathbf{D})_{\text{var}(\mathbf{X})})$ and $\perp((\mathbf{C}\cup\mathbf{D})_{\text{var}(\mathbf{X})})=\perp(\mathbf{C}\cup\mathbf{D})$ (Lemma 5.34). Therefore, $x\in\text{var}(\perp(\mathbf{C}\cup\mathbf{D}))$. □

Lemma 5.78: If $\mathbf{C}$ and $\mathbf{D}$ are sets of equalities, s is a reduced set for $\mathbf{C}$, v is a reduced set for $\mathbf{D}$, and $\text{var}(\mathbf{C})\cap\text{var}(\mathbf{D})$ is a subset of s, then $v \cup(s-\text{var}(\mathbf{D}))$ is a reduced set for $\mathbf{C}\cup\mathbf{D}$.

Proof. We use induction on $|v|$.

Base case: Assume $|v|=0$, then $v=\emptyset$ and $\perp\mathbf{D}=\emptyset$. Since $v=\emptyset$, $v\cup(s-\text{var}(\mathbf{D}))=s-(\text{var}(\mathbf{C})\cap\text{var}(\mathbf{D}))$.

Since $\text{var}(\mathbf{C})\cap\text{var}(\mathbf{D})\subseteq s$, by Lemma 5.50, $s-(\text{var}(\mathbf{C})\cap\text{var}(\mathbf{D}))=s-\text{var}(\mathbf{D})$ is a reduced set for $\mathbf{C}_{\text{var}(\mathbf{C})\cap\text{var}(\mathbf{D})}=\mathbf{C}_{\text{var}(\mathbf{D})}$ and, by Lemma 5.48, for $\perp(\mathbf{C}_{\text{var}(\mathbf{D})})$. Because $\perp\mathbf{D}=\emptyset$, by Lemma 5.75,

$\perp(\mathbf{C}\cup\mathbf{D})=\perp(\mathbf{C}_{\mathrm{var}(\mathbf{D})})$ so that s–var(**D**) is a reduced set for $\perp(\mathbf{C}\cup\mathbf{D})$ and, by Lemma 5.48, for $\mathbf{C}\cup\mathbf{D}$. Therefore, $v\cup(s-\mathrm{var}(\mathbf{D}))$ is a reduced set for $\mathbf{C}\cup\mathbf{D}$.

Assumption: Assume that for some $n>0$ the result holds for any set of equalities **B** such that the size of the reduced sets for **B** is less that n.

Induction: Assume $|v|=n$, and $x\in v$. Since $\{x\}$ is a subset of v, by Lemma 5.50, $v-\{x\}$ is a reduced set for $\mathbf{D}_{\{x\}}=\mathbf{D}_x$.

If $x\in\mathrm{var}(\mathbf{C})\cap\mathrm{var}(\mathbf{D})$, then $\{x\}$ is a subset of s so that by Lemma 5.50 $s-\{x\}$ is a reduced set for $\mathbf{C}_{\{x\}}=\mathbf{C}_x$. Since $\mathrm{var}(\mathbf{C}_x)\cap\mathrm{var}(\mathbf{D}_x)=(\mathrm{var}(\mathbf{C})\cap\mathrm{var}(\mathbf{D}))-\{x\}\subseteq s-\{x\}$ and $|v-\{x\}|<n$, by the induction hypothesis, $(v-\{x\})\cup((s-\{x\})-\mathrm{var}(\mathbf{D}_x))$ is a reduced set for $\mathbf{C}_x\cup\mathbf{D}_x=(\mathbf{C}\cup\mathbf{D})_x$ and, by Lemma 5.48, for $\perp((\mathbf{C}\cup\mathbf{D})_x)$. By Lemma 5.77, $x\in\mathrm{var}(\perp(\mathbf{C}\cup\mathbf{D}))$ so that by Lemma 5.49, $\perp((\mathbf{C}\cup\mathbf{D})_x)=\perp((\perp(\mathbf{C}\cup\mathbf{D}))_x)$. Therefore, $(v-\{x\})\cup((s-\{x\})-\mathrm{var}(\mathbf{D}_x))$ is a reduced set for $\perp((\perp(\mathbf{C}\cup\mathbf{D}))_x)$ and, by Lemma 5.48, for $(\perp(\mathbf{C}\cup\mathbf{D}))_x$. But $(v-\{x\})\cup((s-\{x\})-\mathrm{var}(\mathbf{D}_x))=(v\cup(s-\mathrm{var}(\mathbf{D})))-\{x\}$. Since $x\in\mathrm{var}(\perp(\mathbf{C}\cup\mathbf{D}))$ and $(v\cup(s-\mathrm{var}(\mathbf{D})))-\{x\}$ is a reduced set for $(\perp(\mathbf{C}\cup\mathbf{D}))_x$, $v\cup(s-\mathrm{var}(\mathbf{D}))$ is a reduced set for $\mathbf{C}\cup\mathbf{D}$.

If $x\notin\mathrm{var}(\mathbf{C})\cap\mathrm{var}(\mathbf{D})\subseteq\mathrm{var}(\mathbf{C})$, then $x\notin s$, $\mathbf{C}=\mathbf{C}_x$ and $\mathrm{var}(\mathbf{C}_x)=\mathrm{var}(\mathbf{C})$. Since s is a reduced set for $\mathbf{C}=\mathbf{C}_x$, $\mathrm{var}(\mathbf{C}_x)\cap\mathrm{var}(\mathbf{D}_x)=(\mathrm{var}(\mathbf{C})\cap\mathrm{var}(\mathbf{D}))-\{x\}\subseteq s$ and $|v-\{x\}|<n$, by the induction hypothesis, $(v-\{x\})\cup(s-\mathrm{var}(\mathbf{D}_x))$ is a reduced set for $\mathbf{C}_x\cup\mathbf{D}_x=(\mathbf{C}\cup\mathbf{D})_x$ and, by Lemma 5.48, for $\perp((\mathbf{C}\cup\mathbf{D})_x)$. By Lemma 5.77, $x\in\mathrm{var}(\perp(\mathbf{C}\cup\mathbf{D}))$ so that by Lemma 5.49, $\perp((\mathbf{C}\cup\mathbf{D})_x)=\perp(\perp((\mathbf{C}\cup\mathbf{D}))_x)$. Therefore, $(v-\{x\})\cup(s-\mathrm{var}(\mathbf{D}_x))$ is a reduced set for $\perp((\perp(\mathbf{C}\cup\mathbf{D}))_x)$ and, by Lemma 5.48, for $(\perp(\mathbf{C}\cup\mathbf{D}))_x$. But $(v-\{x\})\cup(s-\mathrm{var}(\mathbf{D}_x))=(v\cup(s-\mathrm{var}(\mathbf{D})))-\{x\}$. Since $x\in\mathrm{var}(\perp(\mathbf{C}\cup\mathbf{D}))$ and $(v\cup(s-\mathrm{var}(\mathbf{D})))-\{x\}$ is a reduced set for $(\perp(\mathbf{C}\cup\mathbf{D}))_x$, $v\cup(s-\mathrm{var}(\mathbf{D}))$ is a reduced set for $\mathbf{C}\cup\mathbf{D}$. □

Lemma 5.79: If for each i $(1\leq i\leq n)$ $\mathbf{C}_i$ is a set of equalities, $\mathrm{var}(\mathbf{C}_i)$ and $\mathrm{var}(\mathbf{C}_j)$ are disjoint for each $j\neq i$, s_i is a reduced set for $\mathbf{C}_i$, and **D** is a set of equalities such that for each i $(1\leq i\leq n)$

var($\mathbf{C}_i$)∩var($\mathbf{D}$) is a subset of s_i , and v is a reduced set for **D**, then $v \cup (\left(\bigcup_{i=1}^{n} s_i\right) - \mathrm{var}(\mathbf{D}))$ is a reduced set for $\mathbf{D} \cup \left(\bigcup_{i=1}^{n} \mathbf{C}_i\right)$.

Proof. We use induction on *n*.

Base case: If n=1, by Lemma 5.78, $v \cup (s - \mathrm{var}(\mathbf{D})$ is a reduced set for $\mathbf{D} \cup \mathbf{C}$.

Assumption: Assume that the result holds for every set of *k* sets of equalities, where $k<n$ and $n>1$.

Induction: By the induction hypothesis $v \cup (\left(\bigcup_{i=1}^{n-1} s_i\right) - \mathrm{var}(\mathbf{D}))$ is a reduced set for $\mathbf{S}=\mathbf{D} \cup \left(\bigcup_{i=1}^{n-1} \mathbf{C}_i\right)$, and by Lemma 5.78,

$(v \cup (\left(\bigcup_{i=1}^{n-1} s_i\right) - \mathrm{var}(\mathbf{D}))) \cup (s_n - \mathrm{var}(\mathbf{S}))$ is a reduced set for $\mathbf{S} \cup \mathbf{C}_n$. But

$$(v \cup (\left(\bigcup_{i=1}^{n-1} s_i\right) - \mathrm{var}(\mathbf{D}))) \cup (s_n - \mathrm{var}(\mathbf{S})) = v \cup ((\left(\bigcup_{i=1}^{n-1} s_i\right) - \mathrm{var}(\mathbf{D})) \cup (s_n - \mathrm{var}(\mathbf{S})))$$

$$= v \cup ((\left(\bigcup_{i=1}^{n-1} s_i\right) - \mathrm{var}(\mathbf{D})) \cup (s_n - (\mathrm{var}(\mathbf{D}) \cup \mathrm{var}(\left(\bigcup_{i=1}^{n-1} \mathbf{C}_i\right)))))$$

Since $\mathrm{var}(\left(\bigcup_{i=1}^{n-1} \mathbf{C}_i\right)) \cap s_n = \varnothing$

$$(v \cup (\left(\bigcup_{i=1}^{n-1} s_i\right) - \mathrm{var}(\mathbf{D}))) \cup (s_n - \mathrm{var}(\mathbf{S})) = v \cup ((\left(\bigcup_{i=1}^{n-1} s_i\right) - \mathrm{var}(\mathbf{D})) \cup (s_n - (\mathrm{var}(\mathbf{D}))$$

$$= v \cup (\left(\bigcup_{i=1}^{n} s_i\right) - \mathrm{var}(\mathbf{D}))$$

and $\mathbf{S} \cup \mathbf{C}_n = \mathbf{D} \cup \left(\bigcup_{i=1}^{n} \mathbf{C}_i\right)$. □

In the next lemma we will extend the above result to reduced sets with respect to a set of variables.

Lemma 5.80: If for each i ($1\le i\le n$) $\mathbf{C}_i$ is a set of equalities, var($\mathbf{C}_i$) and var($\mathbf{C}_j$) are disjoint for each $j\neq i$, s_i is a reduced set for $\mathbf{C}_i$ with respect to v_i , and $\mathbf{D}$ is a set of equalities such that for each i ($1 \le i \le n$) var($\mathbf{C}_i$)$\cap$var($\mathbf{D}$)$\subseteq s_i$, then $v \cup\left(\left(\bigcup_{i=1}^{n} s_i\right) - \text{var}(\mathbf{D})\right)$ is a reduced set for $\mathbf{D}\cup\left(\bigcup_{i=1}^{n} \mathbf{C}_i\right)$ with respect to $w=\left(\bigcup_{i=1}^{n} v_i\right)$, where v is a reduced set for $\mathbf{D}_w$.

Proof. Since for each i ($1\le i\le n$), s_i is a reduced set for $\mathbf{C}_i$ with respect to v_i , by Lemma 5.64, $s_i - v_i$ is a reduced set for $(\mathbf{C}_i)_{v_i}$. Since var($(\mathbf{C}_i)_{v_i}$) and var($(\mathbf{C}_j)_{v_j}$) are disjoint for each $j\neq i$,v is a reduced set for $\mathbf{D}_w$ and var($(\mathbf{C}_i)_{v_i}$)$\cap$var($\mathbf{D}_w$)$\subseteq$(s_i−v_i) for each i, by Lemma 5.79, $v\cup\left(\left(\bigcup_{i=1}^{n} (s_i - v_i)\right) - \text{var}(\mathbf{D}_w)\right)$ is a reduced set for $\mathbf{D}_w\cup\left(\bigcup_{i=1}^{n} (\mathbf{C}_i)_{v_i}\right)$, so that $v\cup\left(\left(\bigcup_{i=1}^{n} s_i\right) - \text{var}(\mathbf{D})\right)$ is a reduced set for $\mathbf{D}\cup\left(\bigcup_{i=1}^{n} \mathbf{C}_i\right)$ with respect to w. □

The result of this lemma could be used in computing a factoring of a solid created as the result of the application of an operation to a set of operand solids. Instead of applying the reduction algorithm to the entire set of equalities for the new solid, the algorithm would compute a reduced set for the simplified constraint of the operation and combine the result with the reduced sets for the operand solids. This could substantially improve the time complexity of the reduction process for solids. However, as the reader has probably noticed, incremental reduction requires that each operand distinguish between variables required by the exposed interfaces and the rest. However, this could be a reasonable price for what is gained by subjecting fewer equalities to the reduction algorithm, especially considering the exponential time complexity of the reduction algorithm in the worst case.

It must be mentioned that although the reduction algorithm considers only equalities, an operation and its operand solids may include constraints that are not equalities.

5.5 Behaviour

The word *behaviour* usually means actions performed by some entity that we can observe: that is, a behaviour is a sequence of the states an object goes through over some period of time. For example, as a wheel rolls along the ground we observe its changing position. In general, behaviour involves the change of state that results from a change in some variable.

A solid represents a family of objects in a design space. Our definition of behaviour exploits this fact by taking the view that a family of objects represents all the states of one object, that can be observed as the value of a parameter is varied.

Definition 5.81: Let $\mathcal{D}$ be a design space, Φ be a solid in $\mathcal{D}$ in n variables and p be a variable of Φ. The *behaviour of* Φ *with respect to* p is the partial function ϕ_p: $\mathbf{R} \rightarrow (\mathbf{R}^{n-1} \rightarrow \mathcal{D})$ such that $\forall y \in \mathbf{R}$, $z \in \mathbf{R}^{n-1}$, $\phi_p(y)(z)=[\ y=x_p\ ,\ z=\overline{x}_p\ ]\Phi(x)$. Here $(\mathbf{R}^{n-1} \rightarrow \mathcal{D})$ denotes the set of all partial functions from $\mathbf{R}^{n-1}$ to $\mathcal{D}$. p is called the *controlling variable* of the behaviour.

Just as the definition of solid is a generalisation of the ordinary notion of solid, providing for parameterisation, our definition of behaviour is a generalisation of the ordinary notion of behaviour discussed above in that it applies to parameterised solids. In fact, given a particular value for the controlling variable, the behaviour picks out the family of objects that represent the states of the whole family of objects corresponding to the chosen value for the controlling variable.

Example 5.82: Consider the **Disk** solid of Example 5.5. The behaviour of **Disk** with respect to its first variable is the function ϕ_1:$\mathbf{R} \rightarrow (\mathbf{R}^2 \rightarrow \mathcal{D})$ such that $\forall b \in \mathbf{R}$, $\phi_1(b)(c,r)=[\ 0 \leq r\ ]\{(x,y)\ |\ x,y \in \mathbf{R}$ and $(x-b)^2+(y-c)^2 \leq r^2\}\ \forall c,r \in \mathbf{R}$. The controlling variable of this behaviour is the horizontal position of the centre of the disk. As this value varies, the state of the disk, that is, the particular selection of points that it consists of, moves horizontally. It is, of course, difficult to reconcile this notion of behaviour with our normal understanding of the term because we are dealing with a parameterised object. If we fix the values of all the variables except the controlling one, then we see that the behaviour in this example corresponds to the set of all possible positions of a disk of fixed radius, constrained to lie on a fixed horizontal line.

In the real world the behaviour of a solid may be defined with respect to parameters that are apparently not part of the definition of the solid itself. However, as we have noted above, there may be many different but equivalent solids corresponding to a particular family of objects in a design space. Each such solid may have variables which are not directly related to its geometry or properties, but are used by the function to determine them. We illustrate this as follows.

Example 5.83: Suppose that we would like the horizontal position of the centre of our disk in Example 5.5 to be computed from time t, horizontal speed v, and starting position b_0, so $b=b_0+vt$. We can therefore represent our disk by the function **Disk'** where **Disk'**$(b,c,r,v,t,b_0)=[0\leq r\ ,\ 0\leq v\ ,\ b=b_0+vt]$ **Disk**(b,c,r). Note that **Disk** and **Disk'** are equivalent. Clearly, we can now define a behaviour for the disk with respect to time, the fifth variable of **Disk'**.

As this example illustrates, to allow the definition of a behaviour for a solid to be defined with respect to a variable other than those over which the solid is defined, we need to be able to inject new variables into a solid, such as time in the disk example. This can be done by applying an operation to a solid that introduces new variables together with appropriate constraints. It is important to notice that injecting extra parameters into a solid does not change the fundamental nature of the family of objects the solid defines. The extra parameters just allow us to define relationships between variables that can be exploited to define a new behaviour by constraining the family to one of its subsets.

There are two possible ways for injecting new variables into a solid, both of which rely on the concept of operation defined in [37]. The first is to apply to the solid a unary operation that provides the necessary extra variables and constraint. The second method is to apply a binary operation to the target solid and a special solid which provides the required new variables.

Although the definition of operation in [37] allows us to introduce new variables to a solid using the second method above by using special solids, it cannot be used to directly inject new variables into a solid, as required by the first method. However, as we mentioned in the previous section, our modified definition of an operation accommodates the direct injection of new variables.

Example 5.84: Let $\mathcal{D}$ and **Disk** be as defined in Example 5.5. An operation that defines a horizontal move behaviour for **Disk** can be defined by **H-Move**$=((a),(b,c,v,t,b_0),\mathbf{I},\mathbf{L},\mathbf{C})$ where **I** is the identity function, $\mathbf{L}(a,b,c)$ is true iff a is a solid with a reference point (b,c), and $\mathbf{C}=\{\ 0\leq v\ ,\ b=b_0+vt\}$. Here, v, t, and b_0 are the three extra variables of the operation representing horizontal speed, time and starting position respectively. By "reference point" we mean some point in a solid to which all other points can be related, such as the centre of a disk, or the left upper corner of a square. Applying **H-Move** to **Disk** creates a solid **Disk'** such that for all b,c,v,t,b_0, and r, **Disk'**$(b,c,r,v,t,b_0)=[\ 0\leq r\ ,\ 0\leq v\ ,\ b=b_0+vt\ ]$ **Disk**(b,c,r). As noted in Example 5.83, **Disk'** has a behaviour with respect to variable t. When non-controlling variables of **Disk'** are fixed, this behaviour with respect to t is the set of disks with a fixed radius r anchored on a horizontal line $y=c$. This can be pictured as a disk that moves with speed v on a horizontal line $y=c$ where b_0 is the initial horizontal position of the centre of the disk at time $t=0$. When t is fixed, the behaviour is a family of disks with various radii and speed, horizontally displaced from a reference point b_0.

An operation may be sufficiently generic to be applicable to more than one kind of solid. For example, the **H-Move** operation in Example 5.84 can be applied to any solid that has a reference point, as illustrated below.

Example 5.85: Let **Box** be the solid over the design space of Example 5.84 defined as **Box**(b,c,w,h)= $[\ 0\leq w\ ,\ 0\leq h\ ]\ \{(x,y)\mid b\leq x\leq b+w$ and $c\leq y\leq c+h\}$. Applying **H-Move** to **Box** creates the solid Ψ defined by $\Psi(b,c,w,h,v,t,b_0)$=$[\ 0\leq w\ ,\ 0\leq h\ ,\ 0\leq v\ ,\ b=b_0+vt\ ]$ **Box**(b,c,w,h). Ψ can be simplified to an equivalent solid defined by **Sliding-window**(c,w,h,v,t,b_0)=$[\ 0\leq w\ ,\ 0\leq h\ ,\ 0\leq v\ ,\ b=b_0+vt\ ]$ **Box**(b,c,w,h).

Although most behaviours of solids in the real world are with respect to time, the definition of a behaviour presented here is sufficiently general to capture non-temporal behaviours.

Example 5.86: Let **Cylinder**(b,c,d,v,h)=$[0<h\ ,\ 0\leq v]\ \{(x,y,z)\mid x,y,z\in\mathbf{R}$ and $(x-b)^2+(y-c)^2\leq(v/h\pi)$ and $d\leq z\leq d+h\}$, then **Cylinder** defines a vertical cylindrical solid in a 3D design space $\mathcal{D}$ over zero properties where (b,c,d) is the centre of the base of a cylinder with volume v and height h. The behaviour of this solid with respect to its volume v is the partial function ϕ_4:$\mathbf{R}\rightarrow(\mathbf{R}^4\rightarrow\mathcal{D})$ such that for all $0\leq v$, $\phi_4(v)(b,c,d,h)$= $[\ 0<h\ ]\ \{(x,y,z)\mid x,y,z\in\mathbf{R}$ and $(x-b)^2+(y-c)^2\leq(v/h\pi)$ and $d\leq z\leq d+h\}$. Note that changing the volume v will make a cylinder become thinner or thicker while its height and position remain intact.

As we mentioned earlier, there are two different ways for defining a new behaviour. We have presented one of them, namely, applying a unary operation that imposes new constraints on the parameters of a solid and possibly introduces new parameters if necessary.

In the second method, the new variables required for a behaviour, time for example, could be introduced by some special solid, and an associated special operation could extract the appropriate variables from the solid to which the behaviour is being attached and constrain them to variables supplied by the special solid. In fact, the special solid, rather than being just a technical device for accomplishing the result, could represent a physical controller, for example a slider like that in the QuickTime movie player, that varies time over an interval. The binary operation that modifies the target solid so that a required behaviour can be defined for it, provides the means for attaching such a slider to the target solid. The operation extracts an appropriate variable from the target solid, for example “angle to horizontal axis”, and constrains it to the time variable from the slider in an appropriate way. Hence applying the operation to the special solid and target solid is analogous to wiring a slider to an interface element, as is done in some graphical user interface builders.

Example 5.87: Let $\mathcal{D}$ be a design space in 2 dimensions over zero properties. For simplicity, we represent points in $\mathcal{D}$ in clockwise polar coordinates, where the angle component is measured in degrees. Let **Beam** be a solid in $\mathcal{D}$, a straight line at angle a from the vertical axis and length l with one end at the origin with a perfect factoring $((a,l),(),(z_1,z_2),\mathrm{C},\Phi)$, where $\mathrm{C}=\{0\leq l,\ a=z_1,\ l=z_2\}$ and $\Phi(z_1,z_2)=[\mathbf{true}]\ \{(\alpha,r)\mid \alpha,r\in\mathbf{R},\ \alpha=z_1 \text{ and } 0\leq r\leq z_2\}$. Let **Arc** be a solid in $\mathcal{D}$ defined, also in polar coordinates, with a perfect factoring $((d,t),(),(z_3,z_4),\mathrm{D},\Theta)$, where $\mathrm{D}=\{d=z_3,\ t=z_4\}$ and $\Theta(z_3,z_4)=[\mathbf{true}]\ \{(\alpha,1)\mid \alpha\in\mathbf{R} \text{ and } 0\leq\alpha\leq z_3z_4\}$. Note that **Arc** defines an arc on the unit circle centred at the origin. We use this second solid to inject time variable t and a coefficient variable d into other solids. Now let **Add-time** be the operation $((o_1,o_2),(a,d,t),\mathrm{F},\{\mathrm{L}_1,\mathrm{L}_2\},\{a=dt\})$, where $\mathrm{L}_1(o_1,a)$ is true iff o_1 is a beam with angle a (the first parameter of **Beam**) and $\mathrm{L}_2(o_2,d,t)$ is true iff o_2 is an **Arc**, and $\mathrm{F}(p_1,p_2)=p_1$. Applying **Add-time** to the two operands **Beam** and **Arc** results in a solid Ψ which has a perfect factoring $((a,l,d,t),(),(z_1,z_2,z_3,z_4),\mathrm{E},\Delta)$ where $\mathrm{E}=\{0\leq l,\ a=dt,\ a=z_1,\ l=z_2,\ d=z_3,\ t=z_4\}$ and $\Delta(z_1,z_2,z_3,z_4)=\mathrm{F}(\Phi(z_1,z_2),\Theta(z_3,z_4))=\Phi(z_1,z_2)$ (Lemma 5.18). Let **Hand**(l,d,t) be a solid equivalent to Ψ with a factoring $((l,d,t),(a),(z_1,z_2,z_3,z_4),\mathrm{E},\Phi)$. Since z_3 and z_4 have no occurrences in the definition of Δ, this latter factoring can be simplified to $((l,d,t),(a),(z_1,z_2),\mathrm{X},\Phi)$, where $\mathrm{X}=\{0\leq l,\ a=dt,\ a=z_1,\ l=z_2\}$. Applying **Add-time** to **Beam**(a_s,l_s) and **Arc**$(6,t_s)$, will create a solid **Hand$_1$**(l_s,t_s)=**Hand**$(l_s,6,t_s)$, representing a clock hand with l_s length. When t_s represents time in seconds, **Hand$_1$** exhibits a behaviour analogous to the behaviour of a clock's second hand. Similarly, applying **Add-time** to **Beam**(a_m,l_m) and **Arc**$(6,t_m)$, will create a solid **Hand$_2$**(l_m,t_m)=**Hand**$(l_m,6,t_m)$, representing a clock hand with l_m length. When t_m represents time in minutes, **Hand$_2$** exhibits a behaviour analogous to the behaviour of a clock's minute hand. An application of **Add-time** to **Beam**(a_h,l_h) and **Arc**$(30,t_h)$, will create a solid **Hand$_3$**(l_h,t_h)=**Hand**$(l_h,30,t_h)$, which behaves like a clock's hour hand when t_h is in hours.

In order to create a clock with three hands, let **Pin** be the operation $((o_s,o_m,o_h),(t_s,t_m,t_h),F_1\cup F_2\cup F_3, \{L_s,L_m,L_h\}, \{t_s=60t_m , t_m=60t_h\})$, where $L_s=L_{o,t}[o_s,t_s]$, $L_m=L_{o,t}[o_m,t_m]$, $L_h=L_{o,t}[o_h,t_h]$ and $L(o,t)$ is true iff o is a solid with a time variable t. Applying **Pin** to the solids **Hand$_1$**(l_s,t_s), **Hand$_2$**(l_m,t_m) and **Hand$_3$**(l_h,t_h) with $((l_s,t_s),(a_s),(z_1,z_2),X_1,\Phi)$, $((l_m,t_m),(a_m),(z_3,z_4),X_2,\Phi)$, and $((l_h,t_h),(a_h),(z_5,z_6),X_3,\Phi)$ factorings, respectively, corresponding to the second, minute, and hour hands of a clock respectively, will create a solid with a perfect factoring $((l_s,t_s,l_m,t_m,l_h,t_h),(a_s,a_m,a_h),(z_1,z_2,z_3,z_4,z_5,z_6),H,\Psi)$ where

$$
\begin{aligned}
H = \{\ & 0 \le l_s , a_s=6t_s , a_s=z_1 , l_s =z_2 \\
& 0 \le l_m , a_m=6t_m , a_m=z_3 , l_m =z_4 \\
& 0 \le l_h , a_h=30t_h , a_h=z_5 , l_s =z_6 \\
& t_s=60t_m, t_m=60t_h \}
\end{aligned}
$$

$$\Psi(z_1,z_2,z_3,z_4,z_5,z_6) = \Phi(z_1,z_2)\cup\Phi(z_3,z_4)\cup\Phi(z_5,z_6)$$

Clock(l_s,l_m,l_h,t_s) is a solid reduced with respect to H and $\emptyset$. **Clock**(l_s,l_m,l_h,t_s) has a perfect factoring $((l_s,l_m,l_h,t_s),(t_m,t_h,a_s,a_m,a_h),(z_1,z_2,z_3,z_4,z_5,z_6),W,\Psi)$ where Ψ is as in the above and

$$
\begin{aligned}
W= \{\ & 0 \le l_s , 0 \le l_m , 0 \le l_h , \\
& a_s=6t_s , a_m=6t_m , a_h=30t_h , \\
& t_s=60t_m , t_m=60t_h , \\
& a_s=z_1 , l_s=z_2 , a_m=z_3 , l_m=z_4 , a_h=z_5 , l_h=z_6 \}.
\end{aligned}
$$

Clock has a behaviour with respect to its fourth variable defined as follows

$$\phi_4(t_s)(l_s,l_m,l_h)=[\ W\ ]\ \Psi(z_1,z_2,z_3,z_4,z_5,z_6).$$

5.6 Multiple Behaviours

In the previous section, in order to introduce the notion of behaviour, we formally defined the behaviour of an object with respect to one parameter, however, there is no need to limit our definition in this way.

Definition 5.88: Let Φ be a solid in a design space $\mathcal{D}$. A *free set* for a solid Φ is any proper subset of a parameter set for Φ.

Definition 5.89: Let $\mathcal{D}$ be a design space, and let Φ be a solid in $\mathcal{D}$ in n variables and $p_1,\ldots,p_k$ denoted by p be an arbitrary but fixed ordering of a free set for Φ. The *behaviour of* Φ *with respect to variables* $p_1,\ldots,p_k$ is the partial function ϕ_p: $\mathbf{R}^k \rightarrow (\mathbf{R}^{n-k} \rightarrow \mathcal{D})$ such that $\forall y \in \mathbf{R}^k$, $z \in \mathbf{R}^{n-k}$, $\phi_p(y)(z) = [\ y = x_p\ ,\ z = \bar{x}_p\ ]\ \Phi(x)$. Here $(\mathbf{R}^{n-k} \rightarrow \mathcal{D})$ denotes the set of all partial functions from $\mathbf{R}^{n-k}$ to $\mathcal{D}$. $p_1,\ldots,p_k$ are called the *controlling variables* of the behaviour.

Definition 5.90: If ϕ_p is the behaviour of a solid Φ with respect to a free set p for Φ, then Φ is said to have $|p|$ *degrees of freedom* with respect to the behaviour ϕ_p.

Example 5.91: Consider **Disk'**$(b,c,r,v,t,b_0) = [\ 0 \leq r\ ,\ 0 \leq v\ ,\ b = b_0 + vt\ ]$ **Disk**(b,c,r) from Example 5.84. The behaviour of **Disk'** with respect to its third and fifth variables is the function $\phi_{3,5}$:$\mathbf{R}^2 \rightarrow (\mathbf{R}^4 \rightarrow \mathcal{D})$ such that $\forall r,t \in \mathbf{R}$, $\phi_{3,5}(r,t)(b,c,v,b_0) = [\ 0 \leq r\ ,\ 0 \leq v\ ,\ b = b_0 + vt\ ]\ \{(x,y) \mid x,y \in \mathbf{R}$ and $(x - b_0 - vt)^2 + (y - c)^2 \leq r^2\}\ \forall b,c,v,b_0 \in \mathbf{R}$. It is important to notice that the variables t and r are independent.

Example 5.92: To create a rolling disk from our disk in Example 5.5, as in Example 5.83, we first define an equivalent solid with appropriate variables including time t, angular speed ω, and starting position b_0, so $b = b_0 + \omega rt$. We can therefore represent our disk by the function **Roller** where **Roller**$(b,c,r,\omega,t,b_0) = [\ 0 \leq r\ ,\ 0 \leq \omega\ ,\ b = b_0 + \omega rt\ ]$ **Disk**(b,c,r). Clearly, we can now define the behaviour of the disk with respect to r and t. However, it is important to notice that this behaviour does not correspond to a rolling disk that inflates or deflates. The difference between these two is that for an inflating or deflating rolling disk, the ratio in which the radius of the disk changes is with respect to time. That would require the two variables which are both controlling variables in this example, to be dependant. Although **Roller** has a behaviour with respect to its radius and time, this behaviour is not analogous to what we think of an inflating or deflating rolling disk. However, if we were to define the radius to be a function of time, then **Roller** could have a behaviour with respect to time that characterises an inflating or deflating rolling disk.

5.7 Motion Modeling

When designing mechanical devices with moving parts, it is important to check whether a moving part of the design intersects itself or other objects in its operating domain. Similarly, in robot programming and motion planning, a robot must perform its task without bumping into other objects. A solid can have two types of motion. First, in a solid with moving parts, some of these parts may move causing changes in the internal configuration of the solid. Second, a solid may move in space without changing its internal configuration. In the latter case, motion planning can be viewed as choreography [92]. In this section, we restrict our attention to the first kind of motion, focussing on behaviours of solids with respect to *the internal configurational variables* of a parameterised solid. The internal configurational variables are those that determine the position and orientation of a rigid solid with respect to a reference coordinate frame attached to the solid. In motion modeling, the rigidity of a solid is maintained throughout the motion [92].

The maximum number of configurational variables of a solid with respect to which the solid has a motion behaviour, as defined in Definition 5.90, is known as the solid's *degree of freedom*. For example, a rigid 2 dimensional square in a plane has a maximum of 3 degrees of freedom where two degrees correspond to the position of the square in the plane with respect to the origin and the third degree of freedom determines the orientation of the square as, say, the angle between the base of the square and one of the axes of the coordinate frame as illustrated in Figure 5.1. In a 3 dimensional space, a free square has 6 degrees of freedom. Three degrees define the position of the square in the space and the other 3 define the yaw, pitch, and roll of the square. Note that both examples have zero degrees of freedom with respect to their internal configurational variables.

Aside from the free configurational variables of a solid, time also plays an important role in motion modeling. The controlling variables of a multiple behaviour are free and therefore could be expressed parametrically in terms of a time variable so that substituting a value for the time variable would result in values for the controlling variables. Therefore, a motion could be thought of as a parametric k-dimensional curve connecting a start point to an end point in a k-dimensional space the dimensions of which correspond to the controlling variables.

Definition 5.93: Let $\mathcal{D}$ be a design space, Φ a solid in $\mathcal{D}$, $p_1,\ldots,p_k$ denoted by p an arbitrary but fixed ordering of a free set for Φ, ϕ_p a multiple behaviour of Φ with respect to p, $[t_s,t_e]$ a closed interval in $\mathbf{R}$, and $\mathbf{f}$ a function from $\mathbf{R}^{2k+1}$ to $\mathbf{R}^k$ such that $\forall x,y \in \mathbf{R}^k$, $\mathbf{f}(x,y,t_s)=x$ and $\mathbf{f}(x,y,t_e)=y$. A *motion of* Φ *with respect to* ϕ_p *and* $\mathbf{f}$ *on the interval* $[t_s,t_e]$ is a partial function $\mathrm{M}:\mathbf{R}^k\times\mathbf{R}^k \to (\mathbf{R}\to(\mathbf{R}^{n-k}\to\mathcal{D}))$ such that $\forall (x,y)\in\mathbf{R}^k\times\mathbf{R}^k$, $\mathrm{M}(x,y)(t)=[t\in[t_s,t_e]\ ,\ z=\mathbf{f}(x,y,t)]\ \phi_p(z)$.

This definition generalises the usual concept of motion to parameterised solids. When all non-controlling variables are given values, fixing the initial and final configurations of the solid determines a motion as a function of time. Substituting the time variable of this function with different various values would result in different instantiations of the solid at the corresponding times. When the non-controlling variables of the solid are not given values, fixing the initial and final configurations would result in a function of time and the non-controlling variables. By substituting the time variable of this function we would get the configuration of the family of objects represented by the solid corresponding to the chosen time value. This latter interpretation, however, is difficult to reconcile with the usual understanding of motion.

Example 5.94: Let $\mathcal{D}$ be the same design space as in Example 5.87. Also let **Hands**(h,m,s,l_h,l_m,l_s) be defined as the following

$$\textbf{Hands}(h,m,s,l_h,l_m,l_s) = [\ s,m,h \in \mathbf{N},\ 0 \le s < 60\ ,\ a_s{=}6s\ ,\ 0 \le m < 60\ ,\ a_m{=}6m{+}s/10\ ,\ 0 < h \le 12\ ,\ a_h{=}30(h \bmod 12){+}m/2{+}s/120\ ]\ \textbf{Beam}(a_s,l_s) \cup \textbf{Beam}(a_m,l_m) \cup \textbf{Beam}(a_h,l_h)$$

Now let **Hands** have a multiple behaviour with controlling variables h,m,s defined by function $\phi_{1,2,3}$. Recall that % represents integer division. A motion of **Hands** with respect to behaviour $\phi_{1,2,3}$ is $\mathbf{G}((h_1,m_1,s_1),(h_2,m_2,s_2))$ such that

$$\mathbf{G}((h_1,m_1,s_1),(h_2,m_2,s_2))(t){=}[\ s_1,m_1,h_1,s_2,m_2,h_2 \in \mathbf{N},\ 0 \le s_1 < 60\ ,\quad 0 \le s_2 < 60\ ,\ 0 \le m_1 < 60\ ,\quad 0 \le m_2 < 60\ ,\ 0 < h_1 \le 12\ ,\quad 0 < h_2 \le 12\ ,\ s{=}(s_1{+}\Delta) \bmod 60\ ,\ m{=}(m_1{+}((s_1{+}\Delta)\ \%\ 60)) \bmod 60\ ,\ h{=}((h_1{+}((m_1{+}((s_1{+}\Delta)\ \%\ 60))\ \%\ 60){-}1) \bmod 12){+}1\ ]\ \phi_{1,2,3}(h,m,s)$$

where

$$\Delta = \mathbf{D}(3600(h_1 \bmod 12){+}60m_1{+}s_1\ ,\ 3600(h_2 \bmod 12){+}60m_2{+}s_2)(t{-}t_s)/(t_e{-}t_s)$$

and

$$\mathbf{D}(x,y){=}y{-}x \qquad \text{if } x \le y$$

$$\mathbf{D}(x,y){=}12{*}3600{-}(x{-}y) \qquad \text{if } x > y$$

Motion **G** from the starting configuration (h_1,m_1,s_1) to the ending configuration (h_2,m_2,s_2) during the interval $[t_s,t_e]$ corresponds to the movements of the hands of a clock. When the time represented by the interval $[t_s,t_e]$ is exactly T, where T is the period of time between h_1 hours m_1 minutes s_1 seconds and h_2 hours m_2 minutes s_2 seconds, the corresponding application of the motion simulates the normal function of a clock. If the interval $[t_s,t_e]$ is smaller than T, then the motion is similar to the behaviour of a clock in fast motion. If the interval $[t_s,t_e]$ is greater than T, then the motion is similar to the function of a normal clock in slow motion. Nevertheless, the motion simulates the actual relation between the hands of a clock when the hands move from the position indicating h_1 hours, m_1 minutes, and s_1 seconds to the one that indicates h_2 hours, m_2 minutes, and s_2 seconds. As the reader has certainly noticed, the motion of the behaviour was created such that the definition of the clock remained intact. This suggests that a motion can be defined for a solid without interfering with the internal structure of a solid. Also note that the motion behaviour is defined for a family of objects rather than for one instance of **Hands**.

Example 5.95: To further explain the definition of motion, we adapt an example discussed by Paoluzzi in [92]. Consider the articulated arm in Figure 5.12, obtained by applying **Joint** to **Reduced-Partial-Robot**$(b_1,c_1,a_1,r_1,a_2,f_2,g_2,\beta)$ and **Reduced-Partial-Robot**$(b_3,c_3,a_3,r_3,a_4,f_4,g_4,\delta)$ from Example 5.71. The product solid resulting from the application of **Joint** has a factoring $(x,y,z,\mathbf{Q},\mathbf{F})$ where

$$x=(b_1,c_1,a_1,r_1,a_2,f_2,g_2,\beta,b_3,c_3,a_3,r_3,a_4,f_4,g_4,\delta,\gamma)$$

$$y=(d_1,e_1,f_1,g_1,l_1,b_2,c_2,d_2,e_2,l_2,r_2,d_3,e_3,f_3,g_3,l_3,b_4,c_4,d_4,e_4,l_4,r_4)$$

$$z=(z_1,z_2,\ldots,z_{20})$$

and **Q** is the constraint consisting of the following literals

$d_1 - b_1 = l_1.\cos(a_1)$ $e_1 - c_1 = l_1.\sin(a_1)$

$f_1 = d_1 + 2r_1.\cos(a_1)$ $g_1 = e_1 + 2r_1.\sin(a_1)$

$d_2 - b_2 = l_2.\cos(a_2)$ $e_2 - c_2 = l_2.\sin(a_2)$

$f_2 = d_2 + 2r_2.\cos(a_2)$ $g_2 = e_2 + 2r_2.\sin(a_2)$

$f_1 = b_2$ $g_1 = c_2$

$\beta = a_2 - a_1$ $r_1 = r_2$

$$-90 \leq \beta \leq 90$$

$d_3 - b_3 = l_3.\cos(a_3)$ $e_3 - c_3 = l_3.\sin(a_3)$

$f_3 = d_3 + 2r_3.\cos(a_3)$ $g_3 = e_3 + 2r_3.\sin(a_3)$

$d_4 - b_4 = l_4.\cos(a_4)$ $e_4 - c_4 = l_4.\sin(a_4)$

$f_4 = d_4 + 2r_4.\cos(a_4)$ $g_4 = e_4 + 2r_4.\sin(a_4)$

$f_3 = b_4$ $g_3 = c_4$

$\delta = a_4 - a_3$ $r_3 = r_4$

$$-90 \leq \delta \leq 90$$

$f_2 = b_3$ $g_2 = c_3$

$\gamma = a_3 - a_2$ $r_2 = r_3$

$$-90 \leq \gamma \leq 90$$

together with

$$z_1 = b_1 \text{ , } z_2 = c_1 \text{ , } z_3 = d_1 \text{ , } z_4 = e_1 \text{ , } z_5 = r_1 \text{ , } z_6 = b_2 \text{ , } z_7 = c_2 \text{ , } z_8 = d_2 \text{ , } z_9 = e_2 \text{ , } z_{10} = r_2$$

$$z_{11} = b_3 \text{ , } z_{12} = c_3 \text{ , } z_{13} = d_3 \text{ , } z_{14} = e_3 \text{ , } z_{15} = r_3 \text{ , } z_{16} = b_4 \text{ , } z_{17} = c_4 \text{ , } z_{18} = d_4 \text{ , } z_{19} = e_4 \text{ , } z_{20} = r_4$$

and

$$\mathrm{F}(z_1, z_2, \ldots, z_{20}) = \Phi(z_1, z_2, \ldots, z_5) \cup \Phi(z_6, z_7, \ldots, z_{10}) \cup \Phi(z_{11}, z_{12}, \ldots, z_{15}) \cup \Phi(z_{16}, z_{17}, \ldots, z_{20}).$$

Let **Robot** be the solid reduced with respect to **Q** and a set of interfaces that includes variables representing the pivot point of the first arm segment (b_1,c_1), the hinge of the last arm segment (f_4,g_4), and angles a_1, β, γ, δ, as shown in Figure 5.10. **Robot** has four rigid arm segments attached to each other at rotative junctions. Our reduction algorithm determines that **Robot**(b_1,c_1,f_1,g_1,f_4,g_4,r_1,a_1,β,γ,δ) is reduced with respect to **Q** and the set of interfaces. **Robot** also happens to be reduced with respect to **Q** and $\varnothing$, indicating that the variables in β, γ, and δ are free variables of **Robot**. **Fold** is a behaviour of **Robot** with controlling variables β, γ, and δ defined by

Fold(β,γ,δ)(b_1,c_1,f_1,g_1,f_4,g_4,r_1,a_1)=[(b_1,c_1,f_1,g_1,f_4,g_4,r_1,a_1,β,γ,δ) $\in$ scope(**Robot**)]

Robot(b_1,c_1,f_1,g_1,f_4,g_4,r_1,a_1,β,γ,δ).

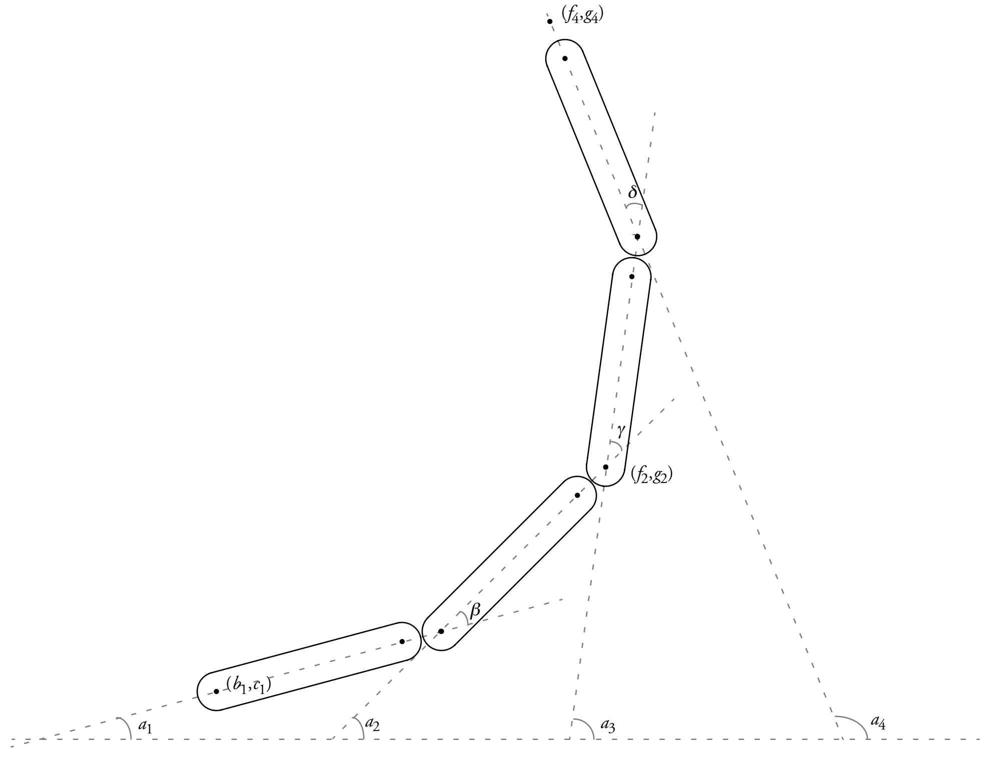


Figure 5.10: A **Robot** with articulated arm.

A motion of **Robot** with respect to **Fold** can be thought of as the continuous changes in the values of β, γ, and δ through time as the hinge of **Robot** (point (f_4,g_4)) moves from an initial position to a final position in a 2D space. Clearly, such a motion is not unique since many different transitions can have similar starting and ending configurations. Figure 5.11 shows two possible curves corresponding to two different motions with respect to **Fold**, a straight line between the two points representing the initial and the final configurations, and three line segments parallel to the axes. For simplicity, the origin coincides with the initial configuration.

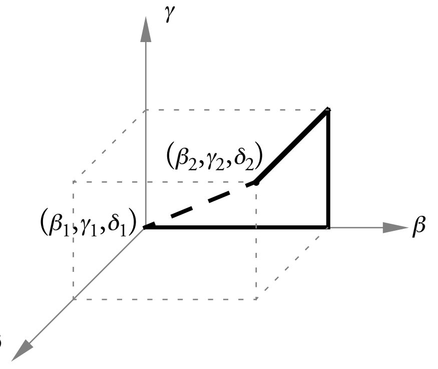


Figure 5.11: Two curves representing two motions for **Robot**

A motion for **Robot** with respect to **Fold** and the first curve in Figure 5.11 is defined by

$$
\begin{aligned}
M((\beta_1,\gamma_1,\delta_1),(\beta_2,\gamma_2,\delta_2))(t)=[\ & t_e \neq t_s\,,\ t_e \leq t \leq t_s\,, \\
& \beta=\beta_1+(\beta_2-\beta_1)(t-t_s)/(t_e-t_s), \\
& \gamma=\gamma_1+(\gamma_2-\gamma_1)(t-t_s)/(t_e-t_s), \\
& \delta=\delta_1+(\delta_2-\delta_1)(t-t_s)/(t_e-t_s)\ ]\ \textbf{Fold}(\beta,\gamma,\delta).
\end{aligned}
$$

The diagram at the top of Figure 5.12 shows a sequence of configurations of **Robot** that corresponds to this motion when **Robot** moves from the initial configuration corresponding to (0,0,0) to the final configuration corresponding to (90,90,90).

The lower diagram in Figure 5.12 corresponds to another motion for **Robot** with respect to **Fold** corresponding to the second curve in Figure 5.11.

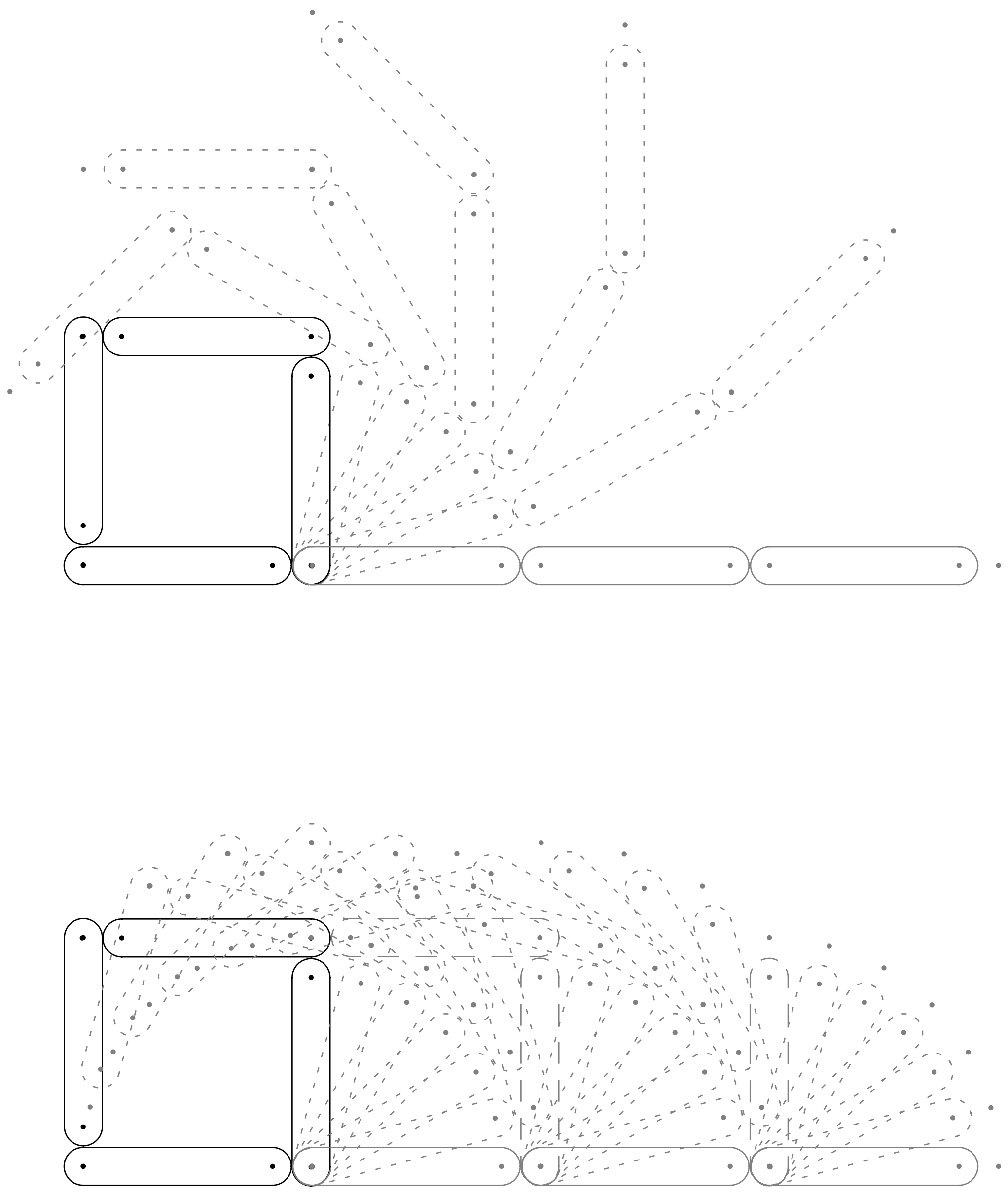


Figure 5.12: Two motions for Robot

5.8 Summary

In this chapter, we have extended the formal model for solids proposed in [37]. We first identified an ambiguity in the interpretation of the empty set in the previous definitions noting that the empty set could signify either an invalid solid or one which happens to include no points in the space. To rectify the problem, we defined a solid as a partial function.

We raised and investigated the issue of sample looks for solids in LSD programs and defined the notion of factoring of a solid, applying it to the automatic computation of a sample look for a partially free solid. The intuition behind factoring is to extract the constraints that define how a solid can be configured so that we can manipulate them in various ways, for example to generate sample values for the parameters of a solid. We noted that although this may be an ideal situation, it is difficult to achieve in practice. To make the notion of factoring more practical we introduced relaxed factoring, in which some, but not necessarily all, constraints are identified. In this case, selecting sample values that are consistent with the constraint of the solid is likely to generate a valid sample look for the object, but this is not guaranteed. We discussed how, in an implementation, the geometric kernel and the constraint solver might engage in a dialogue to iterate towards appropriate sample values.

We described what it means to reduce a solid and identified two conditions in which such reduction is useful. Instead of relying on any specific constraint solver, we made some general assumptions about constraints and constraint solvers and built our solution around those assumptions. The first assumption was to ensure that the constraint set did not include redundant constraints. The second assumption was that there exists a constraint solver that can successfully solve a system of n equations over n variables regardless of the complexity of the equations. The technique we introduced for the reduction algorithm exploits the equalities in the constraint set and does not work with other constraints; however, it does not exclude other constraints from the constraint set in a factoring of a solid.

Systems of simultaneous equations are known to be extremely difficult to solve. In our reduction algorithm, however, we have assumed that they are solvable regardless of their difficulty. An interesting extension to our reduction algorithm could be to investigate specific classes of systems that are more practical to solve.

Finally, we have formally defined the concept of behaviour of a solid and extended this definition to multiple behaviours. Then we identified a particular behaviour, motion, and investigated how it could be modeled in this formalism.

6 Solid Modeling for LSD

A successful assembly of a design specification in LSD, as described in [37], results in an anchor that corresponds to a solid in the employed solid modeler. However, the notion of factoring a solid formalised in Definition 5.15 and the modified definition of an operation according to Definition 5.20, suggest that in what remains from a successful assembly of a design specification could exist a set of constraint cells.

In this chapter, we show how the definitions of an explicit component and a link according to [37] are modified in order to capture in LSD what Definition 5.15 and Definition 5.20 capture in the solid modeling world. Then, we will give a brief recapitulation of various solid modeling schemes in just enough detail to discriminate between them and choose the most promising of them for our purpose. Finally, we will show how a graph resulting from a successful assembly of an LSD query corresponds to an expression in PLaSM, our choice of solid modeling kernel.

Earlier work on the correspondence between representation of solids in LSD and the PLaSM language was reported in [16], and relied on the definitions of solids and operations in [37]. In this chapter, we will revisit the correspondence using the notion of factoring as well as the modified definition of operation presented in Chapter 5.

6.1 Explicit Components and Operations in LSD

In [37], a formalisation of the LSD language was presented which, among other programming constructs, included explicit components and links, corresponding to objects and operations on them in a design space. In this section, we show how their LSD constructs are affected by the revised definitions of solid and operation, and the concept of factoring, introduced in Chapter 5.

Following [37], we assume the existence of a design space $\mathcal{D}$, a set $\mathcal{S}$ of solids in $\mathcal{D}$, and a set O of operations in $\mathcal{D}$. To each selector occurring in an operation in O, corresponds a unique function symbol with arity equal to the size of the selector called an *interface symbol.* To each solid in k variables corresponds a k-ary symbol called an *explicit symbol* which is either a function symbol or a predicate symbol. To each n-ary operation in O, corresponds a unique n-ary predicate symbol called a *link symbol.* The sets of explicit predicate symbols and link symbols are disjoint. Remaining predicate symbols are called *implicit symbols.* Similarly, the sets of explicit function symbols and interfaces are disjoint. Remaining function symbols are called *abstract.*

In the following if X is a selector, solid, or operation, we denote the corresponding symbol by {X}. Table A.1 in Appendix A summarises symbols in LSD and their relations to programming constructs in Lograph, and the design space.

6.1.1 Explicit Components

Explicit components in LSD are logical manifestations of parameterised objects provided by a solid modeling kernel. A solid exposes interfaces which are clues to the operations that can be applied to it, and may represent a primitive or an object assembled from primitives.

As discussed in detail in Section 5.2.1, the major motivation for factoring a solid is to extract the formula that constrains the variables of a solid and express it in LSD. Therefore the definition of an explicit component, the logical manifestation of a solid in LSD, needs to be adjusted to reflect Definition 5.15.

Definition 6.1: An *explicit component* (*e-component*) consists of

- a literal called the *anchor* of the component which is either of the form $r=\{\Phi\}(v_1,\ldots,v_n)$ (*primitive anchor*) or of the form $\{\Phi\}(r,v_1,\ldots,v_n)$ (*defined anchor*) where Φ is a solid in n variables for some $n \geq 0$, r is called *the root*, and $r,v_1,\ldots,v_n$ are distinct variables; and
- for each exposed interface ϕ of Φ, one literal of the form $w=\{\mathbf{L}\}(v_{i_1},\ldots,v_{i_k})$, called a *group*, where $\mathbf{L}$ is the selector corresponding to ϕ, k is the size of $\mathbf{L}$, and $i_1,\ldots,i_k$ is the sequence of variables of Φ required by ϕ. The variable w is called an *implicit group terminal of type* $\mathbf{L}$.

The above definition differs from the definition of an e-component in [37], in its description of the anchor of an e-component, by introducing the root, which, as we will see, will be used by the LSD representation of an operation (a knot) to explicitly tie the anchors of operands to the literal representing the operation. In [16], we solved the problem of finding the operands of an operation by tracking multiple occurrences of variables in the representation of an operation. The introduction of the root of an anchor not only simplifies the search for the operands of an operation, but is essential when the constraint and geometric function of an operation are separated in LSD in the case that no variable of an operand appears in the knot of the operation.

Definition 6.2: An *anchor-knot graph* is a connected graph of anchors, knots and constraint cells such that

- no root of an anchor or knot occurs in the terminals of a constraint cell.
- no root of an anchor occurs as the root of another anchor.

- each root of a knot occurs exactly once as a root of an anchor.

The above definition describes the structure of what remains from a successful assembly of an LSD query, when the query is executed without making calls to a geometric kernel. The conditions in the above definition correspond to those in Definition 5.21 and Lemma 5.24. The difference is that this definition is general enough to describe the structure of a successful assembly before and after executing the constraint cells in the graph. The conditions in this definition are more relaxed than those in Definition 5.21 so that in an efficient implementation of constraint cells of an operation, equalities involving variables in z could be replaced by direct connections indicating unification of the equated variables.

Definition 6.3: If **A** is a defined anchor, the *design of* **A** is a design **D** with the same signature as **A** such that for each case **S** of **D**, the body of **S** consists of an anchor-knot graph, such that the first terminal of the head of **S** occurs as the root of exactly one anchor in the body of **S** and each non-root terminal of an anchor that does not occur as a terminal of a constraint cell in the body of **S** occurs exactly once in the head of **S**, and each terminal of a constraint cell that does not occur as a non-root terminal of an anchor occurs exactly once in the head of **S**.

This definition allows explicit components created by execution to be added to an object library without making any call to a geometric kernel, provided that the LSD programming environment is equipped with a mechanism for creating a defined anchor from what remains of a successful execution of an LSD query. The definition also allows for factorings of a predefined explicit component. For example, a factoring $(x,y,z,\mathbf{C},\Psi)$ of a predefined e-component could be realised by one of the cases of the design of the explicit component's defined anchor as follows. The anchor-knot graph in the body of the case consists of exactly one anchor with name $\{\Psi\}$, root u_1, and non-root terminals w, where u is the head of the case, w is a sequence of variables disjoint from u, $|w|=|z|$ and $|x|=|u|-1$; and exactly one constraint cell of the form $\{\mathbf{C}\}(\overline{u_1}\bullet w)$ the specification of which implements **C**.

The nested structure of cases could be used to either tighten or relax the constraint of an assembled explicit component as needed. Note that regardless of the type of the anchor of an explicit component, a query containing an explicit component could be executed in animated or non-animated mode. The structure of a case for a defined anchor indicates how the corresponding anchor was created and not how it can be used.

6.1.2 Operations

A solid modeling kernel is only part of a solid modeling package and operations in a solid modeling package extend the operations supported by its kernel. The kinds of operations created from those supported by the kernel is what makes two solid modeling packages based on similar kernels different.

Extensions to a solid modeling package are motivated by what is needed by the client software which in our case is LSD. Hence, the generalised notion of an operation is our guideline for what we need to add to a solid modeling kernel. Operations in our solid modeler must also comply with Definition 5.20 which makes adding new parameters to a solid possible.

An operation consists of a list of selectors, a constraint which is a set of formulae and a function that deals with the geometry of the solid. Our first intuition for implementing operations in the solid modeler is to deal with the selectors of an operation in the logic provided by Lograph, deal with the constraints using the constraint logic programming capabilities of Lograph+C and rely on a geometric kernel to compute the geometric aspects of the product solid. This implies that the selection of the available geometric functions for an operation in LSD is limited to those provided by the employed solid modeling kernel.

Definition 6.4: A *link* for an *m*-ary operation $\otimes=(((a_1,\ldots,a_m),q),(z_1\bullet\ldots\bullet z_m\bullet p),\mathcal{F},\mathcal{L},\mathrm{C})$ consists of

- a literal, called a *knot*, of the form $\{\mathcal{F}\}(v_1\bullet\ldots\bullet v_m\bullet g)$ where v_i , called *a root*, is a variable for each i $(1 \leq i \leq m)$ and g is a sequence of variables the same length as q;

- one constraint cell of the form $\{\mathbf{C}\}(u_1\bullet\ldots\bullet u_m\bullet e\bullet g)$ the specification of which implements **C**, where e is a sequence of variables the same length as p, and for each i $(1 \leq i \leq m)$ u_i is a sequence of variables the same length as z_i ; and
- for each i $(1 \leq i \leq m)$ a literal $w_i=\{\mathbf{L}_i\}(u_i)$, also called a *group*, such that $\mathbf{L}_i$ is the i^{th} selector of ⊗. For each i $(1 \leq i \leq m)$ w_i is called an *implicit group terminal of type* $\mathbf{L}_i$.

This definition introduces a link as a generalised form of an operation in LSD. A knot in LSD is a special literal that is not subject to the LSD replacement rule, and does not have any associated definition. A knot is associated with a command or a sequence of commands that a solid modeler would use to assemble an object from a set of objects.

6.2 Common Solid Modeling Schemes

Solid modeling deals with the computation and manipulation of objects. The definition of solids and operations in [37] modified as in Chapter 5, provide just enough formalism to allow a solid modeler package for LSD to be built. In this section, first we will give a very brief overview of common solid modeling techniques and then discuss our approach to solid modeling for LSD.

Terminology and definitions that today are used in geometric modeling were first introduced by Requicha in his classic paper on solid modeling [108]. Requicha defined a *representation space* $\boldsymbol{R}$ to be the collection of all syntactically correct representations. A *syntactically correct representation* is a *finite symbol structure* constructed from a *symbol alphabet* according to *syntactical rules.* The semantics of the representations are defined by associating geometric entities in a *mathematical modeling space* $\boldsymbol{M}$ with representations. A *representation scheme* is formally defined as a mapping s: $\mathbf{M} \rightarrow \mathbf{R}$.

Requicha identifies two classes of properties for representation schemes: *formal* and *informal* [108]. Formal properties can be characterised as the properties of the mapping s and informal properties characterise the practical implications of a representation scheme. The following is a brief description of formal and informal properties of representation schemes according to Requicha, the complete and thorough description of which can be found in [108].

The formal properties of a representation scheme include domain, validity, uniqueness, and completeness. The *domain* of a representation scheme is the domain of the mapping s and characterises the descriptive power of the representation scheme. The range of the mapping s is called the *validity set* of the representation scheme. The *validity* of a representation scheme is of the utmost importance in ensuring that the entities in the range of a representation are not nonsense [62].

A representation scheme s is *unique* if it is an injection from the domain to range. That is, every model in the domain of s maps to a unique representation.

A representation scheme is *complete* or *unambiguous* if every representation in the range represents one single mathematical model in the domain of the scheme.

A complete and unique scheme provides a bijection between the mathematical models of solids in the domain of the scheme and their representations in the range. Such representation schemes are called *canonical*. In a canonical scheme, the identity check between two representations is reduced to a simple syntactic check which otherwise would be performed by checking whether the intersection and the union of the two representations are the same.

It must be noted that most solid modeling schemes are not unique. This is mostly due to the *permutational* and *positional* nature of the representations. For example, rigid translations result in non-unique representations. An example of non-uniqueness due to a permutational characteristic of a representation scheme is semantically identical representations with different symbolic arrangements.

Informal properties of a representation scheme are also of obvious importance. For example, *conciseness* corresponds to the size of a representation; that is, the number of symbols in a symbol structure that represents a geometric object. A concise representation requires less space to store, and therefore can be read from or written to the memory faster, and is also more easily to transmitted over a network. A representation which is detailed and requires too many symbols is called *verbose*. *Ease of use* is another informal property of a representation scheme and refers to the ease with which a user can create geometric objects in the representation scheme. Although the ease with which an object can be created is directly related to the complexity of the object, in a verbose scheme it is much more difficult to create objects. *Efficacy in the context of applications* is the degree to which a representation scheme provides support for correct, efficient, extensible, and robust algorithms for manipulating objects in the representation.

It should be mentioned that Rappoport in [105] points out some of the practical and theoretical drawbacks to the formalism provided by Requicha. However, we find them useful to give a general sense of each of the solid modeling schemes that we describe in the next section.

6.2.1 Classical Solid Modeling Schemes

Requicha in [108] discusses six classical unambiguous solid modeling schemes. Paoluzzi, too, has dedicated a full chapter in his recent book to classical solid modeling schemes [92]. In the following, we will briefly discuss some of the most important ones.

6.2.1.1 Constructive Solid Geometry

In Constructive Solid Geometry (CSG), a solid is represented by an ordered binary tree [31]. The leaves of a CSG tree are either primitive solids or arguments defining rigid motions [61]. The internal nodes of a CSG tree are either single regularized set operations including union, difference, and intersection, or rigid motions. CSG representation of an object is unambiguous but not unique.

6.2.1.2 Boundary Representation (B-Rep)

Boundary Representation (B-Rep) schema represents a solid as boundary specifications [61]. The topological description of a solid is specified in the form of connectivity of its vertices, edges, and faces. The geometric specifications are represented by equations defining the surfaces of the solid. Boundary representations are verbose but they are the most convenient of all the schemes for computer display and rendering. Because of their permutational nature, B-Rep representations are not unique.

6.2.1.3 Sweep Representation

In sweep representation, a solid is characterised by the volume that a moving object called a *generator* sweeps along a *trajectory*. For example, a rectangular cube is the volume covered by a planar rectangle swept along a line segment perpendicular to the plane, with one of its end-points on the plane. Such sweeping of 3D space is called *translational sweeping*. *Rotational sweeping* may be defined in an analogous way. If the generator of a sweep can change size, orientation, or shape, it is called a general sweep. Translational and rotational sweeping are commonly used in many CAD systems where a designer or a draftsman describes an object through a graphical interface. However, the system often translates the described object into a boundary representation.

6.2.2 Other Representations

In addition to the classical modeling systems there are other representations for modeling solids. In the following, we will discuss some of them.

In *feature based solid modeling* schemes, a solid is defined and manipulated through parameterised and user-modifiable definitions such as features, sketches and constraints [114]. Although there is no standard formal definition for features [114], it is widely agreed that feature based solid modeling is an effective high level approach. The main advantage of feature based modeling is that features capture higher level abstractions and therefore a design requires fewer steps with a more concise representation. Some of the problems that feature based parametric solid modeling schemes inherit from their classic modeling schemes are documented in [103,114], and a general approach for solving them through a topological framework for designing part families is proposed in [104].

Generative solid modeling is a relatively new approach to solid modeling originally introduced by Snyder [117,118]. *Generic Geometric Complex* is another modeling scheme for representing decomposed pointsets [105,106].

6.2.3 Commercial Solid Modelers

Although CSG technique takes a more intuitive approach for creating and representing solids, most commercial CAD products that support solid modeling take the B-Rep approach. This is mostly because of the limitations of the CSG technique for creating complex objects.

There are a limited number of commercial solid modeling packages that use proprietary solid modeling kernels such as thinkdesign [69]. Most of the other solid modeling packages are built around one of the two major solid modeling kernels, ACIS from Spatial Technologies [32,120] and Parasolid [113] from Unigraphics. For example, Bentley System's MicroStation [19] and SolidWorks [119] use Parasolid, while Autodesk Inventor [6] employs ACIS.

6.3 PLaSM

Programming Language for Symbolic Modeling (PLaSM) is a functional design language developed by the CAD group at the University of Rome "Le Sapienza". PLaSM is strongly influenced by FL [8,9,10], another functional programming language developed by the Functional Programming Group of IBM Research Division at Almaden, California.

One of the strengths of PLaSM is the closeness of the expressions of the mathematical model and the expressions of the language.

The syntactic validity of a geometric program is insufficient to conclude geometric validity. Usually, more expensive semantic checking is required. In PLaSM, however, it is sufficient for a geometric program to be created by a syntactically correct program, that is "The proper combination of well-formed expressions will always give a well-formed result" [93].

6.3.1 Non-geometric Operators

The non geometric subset of the PLaSM language is composed of functions, applications, primitive objects and sequences that constitute expressions. A sequence is an ordered set of expressions, separated by commas and embedded within a pair of angle brackets. For example, <x,y,z> is a sequence of length 3. An application expression, exp1:exp2, applies the function evaluated from exp1 to the argument evaluated from exp2. For instance, LEN:<x,y,z> = 3, where LEN is a primitive function that computes the length of a sequence. Binary composition of functions, indicated by ~, can be used to apply two functions to an expression in sequence. That is, (f~g):exp = f:(g:exp). Application binds more strongly than composition, therefore, f:g~h denotes (f:g)~h. There is also a notation for a consecutive binary composition of functions, namely COMP:<f_1,f_2,…,f_n>:exp = (f_1~f_2~…f_n):exp=(f_1~f_2~…f_{n-1}):(f_n:exp).

The construction function, denoted by CONS, is used to create a sequence of applications of a sequence of functions. CONS:<f_1,f_2,…,f_n>:exp=[f_1,f_2,…,f_n]:exp=<f_1:exp,f_2:exp,…,f_n:exp>

Conditions in PLASM are of the form of `IF:<p,f,g>:x`.

```
IF:<p,f,g>:x = f:x     if     p:x = TRUE
IF:<p,f,g>:x = g:x     if     p:x = FALSE
```

Built-in function K, called the *constant function*, when applied in combination form K:exp1:exp2 will always return exp2. Another built-in function ID, called the *identity function*, used in the form ID:exp will leave the argument exp unchanged.

There are also many other useful primitive functions in PLaSM, the complete description of which can be found in [92,93,94].

6.3.2 Geometric Operators

PLaSM also includes polyhedral expressions, evaluation of which results in polyhedral complexes. Every polyhedron in PLASM has a double decomposition. The lowest level of the decomposition consists of convex cells. A collection of convex cells constitutes a polyhedral cell. The collection of all polyhedral cells in a polyhedron constitute the polyhedron.

There are only two geometric operators in PLASM that require a programmer to take into account such double decomposition, namely MKPOL and UKPOL.

MKPOL is applicable to a sequence of three parameters. The first is a sequence of the vertices of the polyhedron. The second is a sequence of descriptions of the convex cells that comprise the polyhedron, each consisting of a list of the vertices of the corresponding cell. The third is a sequence of polyhedral cells, each a list of its constituent convex cells. For example,

```
    MKPOL:<<<1,1>,<2,1>,<1,2>>,<<1,2,3>>,<<1>>>
```

constructs a triangle composed of one polyhedral cell which in turn is composed of one convex cell. <1,1>, <2,1>, and <1,2> are the three vertices of the convex cell.

UKPOL can be thought of as the inverse of MKPOL. It decomposes a polyhedron into its building blocks. Although MKPOL~UKPOL is not necessarily equivalent to ID, the algebraic equality MKPOL~UKPOL~MKPOL=MKPOL is always true.

PLaSM also provides predefined functions for normal affine transformations including, translation, scaling, rotation, and shearing identified by T, S, R, H, respectively.

Translation is applied to sequences of specifiers and parameters with the same length. Specifiers identify the axes along which the translation should take place. The parameters specify the values used for the transformation in each direction. For instance, T:<1,3>:<1.5,2> is a translation along the first and third axes to the extent of 1.5 and 2 units, respectively.

`STRUCT` is another useful geometric operator in PLaSM and is used to generate hierarchical structures of objects. This operator is applicable to a sequence composed of polyhedra, affine transformations and other invocations of `STRUCT` operator.

6.3.3 PLaSM Syntax

A PLaSM program is a collection of function definitions. A function in PLaSM is either a *global* function or a *local* function. The definition of a global function consists of an identifier for the function preceded by the keyword DEF and followed by the function *specifiers* and *parameters* at the left hand side of an equality. On the right is a PLaSM expression. Each global function may contain local function definitions provided that they are embodied between the keywords WHERE and END. A global function definition may also contain a set of default values for the specifiers and/or parameters. The default values are indicated between the two keywords DEFAULT and END.

Example 6.5: A planar triangle can be thought of as a convex cell of three points corresponding to the three corners of the triangle, expressed in PLaSM by the following definition

```
DEF slice(x1,y1,x2,y2,x3,y3::IsRealPos) =
          MKPOL:<vertices,convexCells,polyhedralCells>
WHERE
     vertices = <<x1,y1>,<x2,y2>,<x3,y3>>,
     convexCells = <<1,2,3>>,
     polyhedralCells = <<1>>
END
```

6.4 LSD and PLaSM

Each solid modeling package is meant to fulfil a special need depending on its client software. In our case, we seek a solid modeling package that fulfils services requested by LSD. In Section 1.4 we argued in favour of a constraint logic based solid modeling engine based on Lograph+C. In the remainder of this chapter, we will discuss the foundations for such a solid modeler.

Once an LSD query involving several operand solids is successfully executed, the result is supposed to correspond to one or more objects in the geometric kernel. However, applying only replacement, merge and deletion rules produces one or more anchor-knot graphs which do not represent solids. Completing the process requires calls to the solid modeler to collapse each anchor-knot graph into a structure that represents a solid. Note that since constraint cells will be dealt with in Lograph+C, the process of collapsing an anchor-knot graph operates only on the anchors and knots and ignores the constraint cells, however, we will refer to the process as collapsing an anchor-knot graph.

We will use PLaSM as an example to show how the correspondence between explicit components in LSD and solids in a solid modeling kernel can be established. We have chosen PLaSM for our presentation because it

- supports parametric solid modeling;
- naturally can operate with the backtracking mechanism of LSD;
- its functional underpinnings are close to the logic underpinnings of LSD;
- its programming approach to solid modeling allows us to easily infer the constraints between the variables of a solid needed for its factoring. The constraints can be extracted from the syntax of the program defining the solid in PLaSM.

We show through some examples how an anchor-knot graph corresponds to a PLaSM definition.

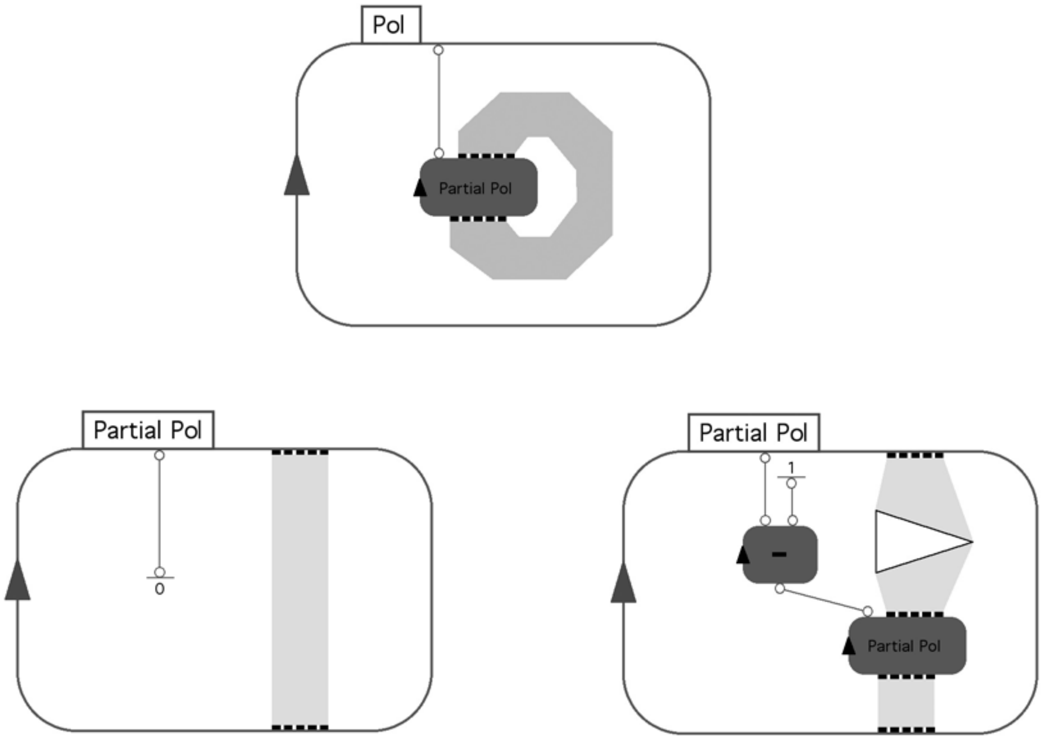


Figure 6.1: Designs **Pol** and **Partial Pol**

Let us start with an example for creating a two dimensional parameterised *n*-sided polygon. Figure 6.1 illustrates designs **Pol** and **Partial Pol** in LSD for creating such a polygon. The white triangle in the recursive case of **Partial Pol** is an explicit component, **Slice**, defined over 6 parameters as shown in Figure 6.2(a). Figure 6.2(b) shows the underlying representation of **Slice** in LSD consisting of a primitive anchor named **Slice** and two groups represented by two function cells **edge**. The underlying structure of the bonding operation contains two implicit group terminals named **edge**, a knot named **fuse**, and a constraint cell named **bond** as depicted in Figure 6.2(c).

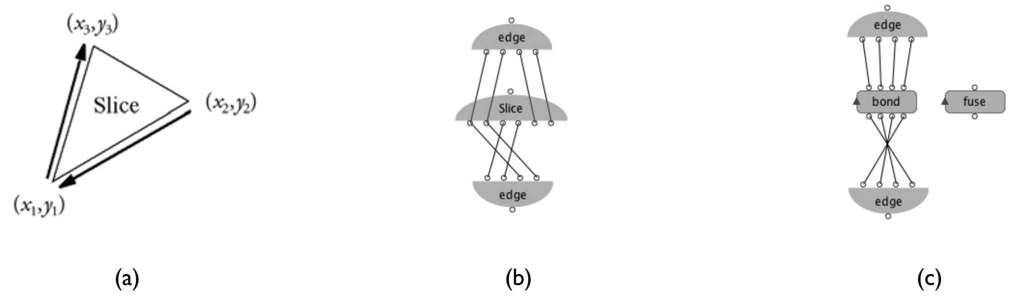
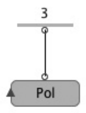
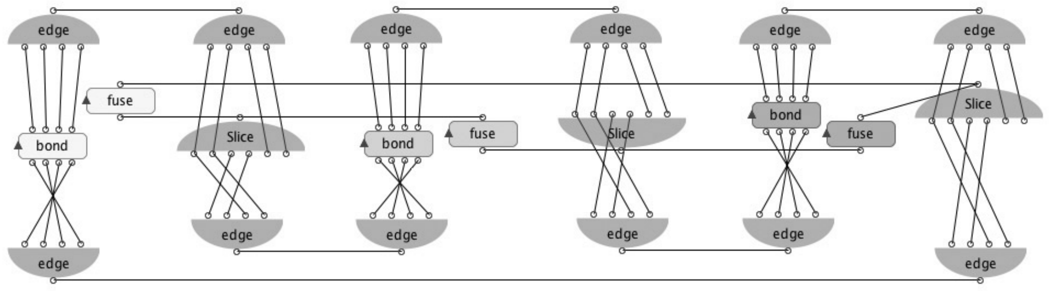


Figure 6.2: Slice solid, and LSD representations of **Slice** and the bond operation

The query in Figure 6.3(a) contains an invocation of design **Pol**. The graph in Figure 6.3(b) shows the product of the assembly before the group function cells are merged and deleted. The shade of colours in which the knots in Figure 6.3(b) are painted correspond to the order in which they were added to the graph. That is, the first knot that must be executed is the one that was added to the graph first.



(a)



(b)

Figure 6.3: A query (a) and its state before merge and deletion (b)

Merging and deleting the **edge** function cells in Figure 6.3(b) will result in an anchor-knot graph that contains three anchors, three knots, and three constraint cell as illustrated in Figure 6.4 after having removed the constraints from the graph as they are dealt with separately. Translating the anchor-knot graph of Figure 6.4 to a PLaSM program requires that every anchor in the graph be associated with a predefined global function in PLaSM. For example, the **Slice** anchors in Figure 6.4 can have the same definition as the one in Example 6.5. This follows from the fact that an anchor in LSD is an invocation of a function in the solid modeler.

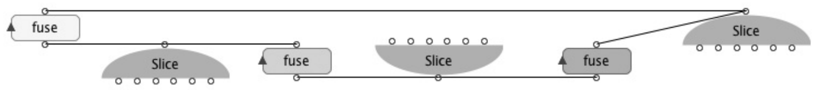


Figure 6.4: An anchor-knot graph

Translating knots to PLaSM expressions, however, is slightly different because an operation in LSD can be applied to a variety of solids if they expose proper interfaces. Here, instead of creating a direct mapping between LSD operations and expressions in PLaSM, we create a blueprint for each knot in LSD which can be used to produce a new anchor which substitutes the knot and its operand anchors. A generic blueprint for a knot in LSD has the following structure.

```
DEF newname(op1_parms,op2_parms,...,opn_parms,extra_parms
                                           ::IsReal) = product
WHERE
      op1 = f1:<op1_parms>,
      op2 = f2:<op2_parms>,
            .
            .
            .
      opn = fn:<opn_parms>,
      product = f:<op1,op2,...,opn,extra_parms>
END
```

The intuition behind the above generic blueprint is provided by Definition 5.21. The parameters of the blueprint are assumed to be of `IsReal` type since they are the sequence of parameters of the operand anchors and extra variables of the operation which all have type `IsReal`. Table 6-1 describes the relationship between the terms in the general blueprint and those of Definition 5.21.

Table 6-1: Relating the generic blueprint for a knot to Definition 5.21.

	Blueprint
Ψ	`newname`
Φ_i	`opi`
$\mathbf{C}$	–
p	–
q	`extra_parms`
$\mathcal{F}$	`f`
$(y_1\cdot\ldots\cdot y_n\cdot q)$	`(op1_parms,op2_parms,...,opn_parms,extra_parms)`

Following the correspondence in Table 6-1, a blueprint for the bonding operation in PLaSM can be expressed as follows

```
DEF newname(op1_parms,op2_parms::IsReal) = product
WHERE
     op1 = f1:<op1_parms>,
     op2 = f2:<op2_parms>,
     product = STRUCT:<op1,op2>
END
```

Note that the above blueprint does not constrain the application of the bonding operation to any particular type of solid and as we will see later, it may as well be applied to a single operand.

In order to show how the above blueprint for bonding can be used to translate an anchor-knot graph containing bond knots to a PLaSM program, we trace the algorithm for translating the anchor-knot graph of Figure 6.4.

Although knots in LSD are not subject to the LSD execution rules, we refer to the process of merging a knot and its operand anchors to create a new anchor as the "execution" of the knot.

Executing a knot in LSD is analogous to the application of an operation to its operand solids in the solid modeler. Therefore, executing a knot will remove the knot and its operand anchors from the anchor-knot graph and replace it with a new anchor. This process can be thought of as the reverse of the replacement rule in LSD, which later we will explain in more detail.

The collapsing of the anchor-knot graph in Figure 6.4 starts with executing the rightmost **fuse** knot, which will collapse the knot and its two operand anchors into a new anchor with the following PLaSM definition.

```
DEF solid1(x1,y1,x2,y2,x3,y3,x4,y4,x5,y5,x6,y6::IsReal)=
                                                  product
WHERE
     op1 = slice:<x1,y1,x2,y2,x3,y3>,
     op2 = slice:<x4,y4,x5,y5,x6,y6>,
     product = STRUCT:<op1,op2>
END
```

Clearly, a name is required for the new anchor and the associated function definition in PLaSM. Here, for simplicity, we have assumed that a newly generated solid is named *solidn* where *n* is the successor of the number of solids previously generated. Figure 6.5(a) shows the anchor-knot graph obtained from that of Figure 6.4 containing a new anchor named **solid1**.

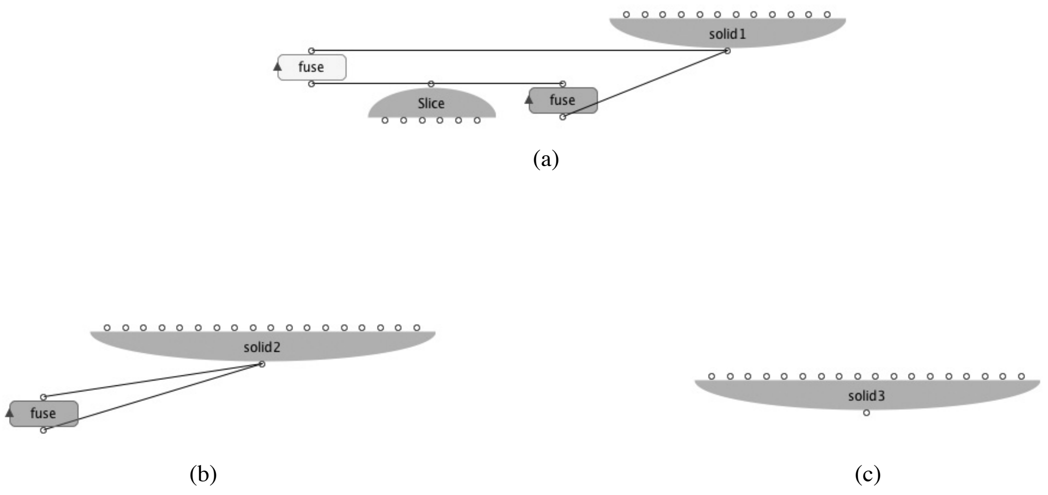


Figure 6.5: Reduced anchor-knot graphs

Executing the rightmost knot in Figure 6.5(a) will collapse it and its associated anchors into a new anchor with the following definition as illustrated in Figure 6.5(b).

```
DEF solid2(x1,y1,x2,y2,x3,y3,x4,y4,x5,y5,x6,y6,
                 x7,y7,x8,y8,x9,y9::IsReal)= product
WHERE
     op1 = solid1:<x1,y1,x2,y2,x3,y3,x4,y4,x5,y5,x6,y6>,
     op2 = slice:<x7,y7,x8,y8,x9,y9>,
     product = STRUCT:<op1,op2>
END
```

Finally, executing the only knot in the anchor-knot graph of Figure 6.5(b) will produce anchor **solid3** that represents the product solid as shown in Figure 6.5(c).

```
DEF solid3(x1,y1,x2,y2,x3,y3,x4,y4,x5,y5,x6,y6,
                x7,y7,x8,y8,x9,y9::IsReal)= product
WHERE
     op1 = solid2:<x1,y1,x2,y2,x3,y3,x4,y4,
                          x5,y5,x6,y6,x7,y7,x8,y8,x9,y9>,
     op2 = solid2:<x1,y1,x2,y2,x3,y3,x4,y4,
                          x5,y5,x6,y6,x7,y7,x8,y8,x9,y9>,
     product = STRUCT:<op1,op2>
END
```

Since `op1` and `op2` are identical, the union of `op1` and `op2`, implemented by `STRUCT:<op1,op2>`, would be identical to `op1` so that `solid3` could be optimised by eliminating `op2` and then replacing the statement

```
product = STRUCT:<op1,op2>
```

with

```
product = op1
```

Although the current version of the translation algorithm, listed in Appendix C, does not perform this kind of optimisation, it does identify repeated variables in the parameter list of the function and eliminates redundant ones. For example, if in the implementation of **bond** equalities involving variables in *z* were replaced by direct connections indicating unification of the equated variables and the translation followed the execution of **bond** cells, the translation would result in the following definitions.

```
DEF solid1(x1,y1,x2,y2,x3,y3,x6,y6::IsReal)= product
WHERE
     op1 = slice:<x1,y1,x2,y2,x3,y3>,
     op2 = slice:<x1,y1,x3,y3,x6,y6>,
     product = STRUCT:<op1,op2>
END

DEF solid2(x1,y1,x2,y2,x3,y3,x6,y6::IsReal)= product
WHERE
     op1 = solid1:<x1,y1,x2,y2,x3,y3,x6,y6>,
     op2 = slice:<x1,y1,x6,y6,x2,y2>,
     product = STRUCT:<op1,op2>
```

```
END

DEF solid3(x1,y1,x2,y2,x3,y3,x6,y6::IsReal)= product
WHERE
     op1 = solid2:<x1,y1,x2,y2,x3,y3,x6,y6>,
     op2 = solid2:<x1,y1,x2,y2,x3,y3,x6,y6>,
     product = STRUCT:<op1,op2>
END
```

6.5 Discussion

The ultimate goal of both LSD and PLaSM is to reconcile the programming world and the design world. PLaSM achieves this by bringing the world of solids into the textual programming world. LSD takes an opposite approach by bringing the programming world into the concrete world of solids. PLaSM deals with solids by providing geometric functions for primitive solids, where LSD does not; however, because of its basis in logic, LSD provides a natural integration of data and program, making simultaneous visual representation of solids and algorithms possible.

What has influenced our decision in choosing PLaSM instead of other solid modeling kernels is its functional approach. PLaSM supports parametric design by defining solids as functions. This is similar to the representation of solids in the formal model for design spaces proposed in [37] and extended here.

The generative power of PLaSM is considerably greater than that of typical variational geometry techniques, owing to the fact that parameters to PLaSM functions can be of any type, including polyhedron and function. We argue that the notion of operations on solids in our solid modeler is comparable to the notion of functions which accept polyhedra as their parameters. This provides a suitable platform to show that the functions generated as the result of translating anchor-knot graphs is a subset of functions that can directly be programmed in PLaSM, however they are just enough to enable a designer to directly interact with solid objects and apply operations to them in a tactile fashion.

Note that anchor-knot graphs are not meant to be a “visual PLaSM”, however they provide one possible visual shell to wrap around the PLaSM kernel in order to exploit the declarative geometric generative power of the PLaSM language.

We have argued earlier in favour of an animated execution environment for Lograph, allowing the user to step through the execution of a query as a debugging aid. If similar animated debugging were carried over to the LSD extension of Lograph, the process of translating anchor-knot graphs to PLaSM would have to be interleaved with Lograph execution in order to provide renderings of e-components as they are generated.

7 Conclusions

This project was motivated by the hypothesis that since both geometric objects and some of the operations used for combining them can have concrete visual representations, a visual design environment that provides programming capabilities via manipulation of such representations would be a powerful tool for creating parameterised objects. Such a design environment would be an attractive alternative to conventional CAD systems, where the algorithms used for the parameterised design and the objects on which they operate have dissimilar representations, one textual, the other pictorial.

This hypothesis, and the observation that a homogeneous integration of data and algorithms that operate on them is the forté of logic programming, led to the conjecture that visual logic programming would provide a reasonable basis for programming designs for parameterised geometric objects. This led to LSD [38], a programming language for parameterised design that provides a close mapping between the programming domain and the design space. LSD also allows designers to more easily solve certain problems within the design space, by employing its declarative problem solving capabilities [15].

The original LSD had just one operation. However, other operations are needed depending on the domain of application. In order to extend LSD, a general notion of design space, solids and operations on solids was needed. The proposed solution to the latter problem was a general characterisation of a design space, solids and operations [37] which then was used as a basis for generalising LSD.

The formalism in [37] defines the concept of design space, solids and operations on solids in general, without further considering the characteristics of a solid modeler to realise this formalism.

The work reported here was inspired by, but not limited to, the directions laid out in [37] and [38] for the integration of LSD in a design environment and its interface with a solid modeler.

7.1 Contributions

This thesis has contributed to the theory and design of LSD, extending the formal model for solids and design spaces by

- Defining the role of constraints by introducing the notion of factoring.
- Refining the formal model to explicitly define the role of constraints in the solid modeler.
- Extending the formal model to provide support for the sample look of a solid and its automatic generation.
- Defining the reduction of solids and designing an algorithm to achieve it.
- Defining solid behaviours.
- Extending the definition of an operation and related definitions to support high level parameterisation and transformations.
- Defining motion as a specific class of behaviours.

- Drawing parallels between the representation of solids in LSD and programs in a functional programming language for solid modeling (PLaSM).
- Designing an algorithm for translating LSD description of solids to PLaSM programs.

Our work has also contributed to certain practical aspects of LSD by

- Devising a deterministic execution mechanism for Lograph.
- Introducing constraints to Lograph to obtain Lograph+C,
- Designing and prototyping an execution engine for Lograph, which reveals unification and resolution while taking full advantage of backtracking of an industrial implementation of Prolog. The engine also facilitates certain desirable LSD debugging features such as single step execution of programs, undo and redo of execution steps, and just-in-time programming.
- Building a programming environment for Lograph including a debugger upon which LSD can be built.
- Designing an algorithm for automatic layout of queries in Lograph, which eventually can be used as a foundation for an automatic layout for an LSD debugger, once extended into 3D.

7.2 Future Work

In the process of addressing various questions about LSD, our work has brought to light many additional interesting and challenging problems. In this section, we discuss some of these and describe our initial intuitions about how to solve them.

7.2.1 Programming Environment

The design of the Lograph interpreter engine is aimed at providing support for many desirable LSD debugging features. Although the design of the Lograph engine realises many such features there are other considerations that for practical reasons now need attention, such as support for numerical computation, integration of constraint solvers, and provision of special operators such as cut. It would also be interesting to investigate other approaches to automatic layout of Lograph programs.

LSD is intended for creating 3D objects, therefore requiring a 3D editor environment. Navigation is considered to be one of the most difficult challenges in designing useful 3D environments since the user can easily become disoriented while navigating in 3D. In a design environment for structural and mechanical design, a 3D environment is of utmost importance. The challenge is how to design a 3D environment which allows the user to employ the debugging features of LSD with minimum cognitive overhead.

7.2.2 Animation of Operations

As suggested in [38] and discussed in Chapters 3 and 4, an implementation of LSD should provide animated execution to help the user understand and debug designs.

Animation of the application of an operation involves transforming the sample looks of the operand solids and visualisation of the operation. In [15], we presented a preliminary definition for the animation of an operation, the intuition behind which was that an animation for an operation could be characterised by the way the variables of the operation's constraint change from their original values before the application of the operation, to values that satisfy the operation's constraint. The animation was defined as a continuous function on the interval during which the change takes place. This continuous function over time defines the feel of the animation. For example, a bonding operation could be animated in a linear fashion or, by means of a second degree animation function, with a snap effect. However, this definition did not address the validity of the operand solids during the animation interval. A more general definition for animation of an operation is required, which also addresses validity. Our intuition is that an approach similar to that used for formalising motion could be used. For an n-ary operation, an animation could be defined by n multidimensional continuous curves. The starting point of each curve would represent a point in the solution set of the constraint of a factoring of the corresponding operand used for the computation of its sample look, and the end point would be the projection of a solution of the constraint of the factoring of the product solid used for the computation of its sample look. The execution of an operation would have a valid animation iff every curve corresponding to an operand was completely contained in the solution set of the constraint of its factoring.

7.2.3 Defining Behaviours by Demonstration

The definition of behaviours for solids in Chapter 5 describes the nature of behaviours, but not how they can be created in a design environment. Defining a new behaviour for a solid could be achieved in many ways. Three possibilities immediately come to mind, as follows

First a programming language, such as the language underlying the corresponding solid modeler, could be employed to directly define a behaviour for a family of solids. For example, PLaSM supports defining behaviours for objects [92].

A second approach is to apply a predefined operation to a solid to obtain a desired behaviour. In this case, the design environment would be equipped with, say, a set of icons, each representing a separate operation with its own selectors and constraint. Dragging and dropping such an icon on to a solid in the design space would trigger the application of the operation to the target solid, extracting the appropriate variables from the solid, adding necessary variables, and applying the operation's constraint. A variation of this would rely on the other means for adding variables, by using special solids to supply them as discussed in Section 5.5. In this case, the environment would provide sliders and other such controller solids which could be connected to the target solids by an appropriate wiring operation, and allow the user to vary the controlling variables of the behaviour.

In the third, more direct approach, the designer explicitly adds new variables to a solid, then creates new relationships between the variables of the solid. Creating these new relationships between variables might be achieved by either using predefined generic constraints, or directly defining constraints by demonstrational techniques.

Programming by Demonstration (PBD) is viewed by many as an elegant way to create agents with behaviour by direct manipulation of representations of domain entities. PBD entails the use of concrete examples to teach an agent how to behave under various situations when there is an analogy between the current circumstances and those of the examples. Different flavours of PBD techniques have been applied to a wide range of applications such as programming repetitive tasks in an operating system [40], building applications [75], animated interfaces [133], text recognition [72], and robot programming [12,73,84]. A PBD system records the actions of the programmer and uses various generalisation techniques to infer the behaviours of the agent [3,12,71,77,85].

In a design environment, two possible applications for PBD are creating new operations on solids and creating new behaviours.

One major difference between PBD for creating automated agents and creating behaviours for solids is that in the former, an agent responds to some environmental changes while in the latter case, a solid does not have the notion of perceiving its surrounding environment. The behaviour of a solid is defined with respect to some of its own variables.

In the case of agent behaviours, inferred behaviour is often represented as a set of rules, finite automata, or a grammar. In the case of solids, we aim to infer constraints that realise a behaviour. In [13], we demonstrated how this could be achieved for linear constraints in a design environment.

A tool that uses PBD to define constraints would employ an appropriate constraint solver, such as a linear equation solver in the case of linear constraints. In order to create a linear constraint that realises a behaviour, the PBD tool needs to know which variables of a solid are also the variables of the constraint. The variables it can choose from are those new variables introduced by the designer, and the current variables of the solid. To achieve this, in [13] we proposed a solid palette for accessing the existing variables of a solid, and adding new variables if needed. Then we proposed an inference mechanism and described how a tool implementing such a mechanism might operate.

This approach for inferring linear constraints can easily be extended to polynomial constraints of any degree given that the user selects the degree of the equation before the inferring process begins. Note that although the constraint itself is polynomial, the solutions to the constraint will provide a set of linear equations the solution to which gives the coefficients of the polynomial constraint.

Our work on using demonstrational techniques for defining behaviours is still preliminary, but it appears to be promising, and deserves further attention.

7.2.4 A Continuous Domain Constraint Solver

Constraints and constraint solving play an important role in LSD. They are essential for the computation of sample looks for solids, defining bahaviours, reduction of solids, and defining operations. Therefore, an “industrial-strength” constraint solver for LSD and its environment is critical.

Despite much progress in the field of constraint satisfaction in recent years, there is still no practical constraint solver which can truly realise and demonstrate the power of constraint programming. Due to the complexity of constraint solving problems, most solvers usually target a specific domain or class of constraints in order to achieve practicality by exploiting domain-specific characteristics. A major difficulty in constraint solvers is the exponential complexity of the algorithms involved, with respect to both time and space. Another contributing factor is that numerical computations needed for constraint solving in continuous domains are both processor-intensive and error prone because of the limitation of computers for representing reals.

For LSD, we need a constraint solver for continuous domains with no limitation on the type of relations and allowing a wide range of operators. A constraint solver which can be used for finding a solution as well as the solution set is required.

Two important properties of continuous domain constraint solvers are soundness and completeness. Soundness means that the solution set of a constraint contains no point that is not consistent in the constraint. Completeness means that all solutions are accounted for in the solution set. In practice, however, achieving both soundness and completeness is extremely difficult and almost impossible considering the approximate nature of the representation of numbers in computers. Instead, a relaxed soundness or completeness is normally sought. In the case of LSD, although it would be desirable to achieve both, soundness appears to be more important than completeness.

Our preliminary work on a constraint solver with the above requirements has resulted in a prototype implementation of a solver for a continuous domain named Progressive Interval Enumeration (PIE).

The key idea in PIE is to generate a symbolic representation of each formula in a set, allowing conjunctions of them to be performed using only logical operations which are normally less expensive than numerical ones. This idea is borrowed from consistency techniques for constraint solving [109,110].

The sequence of symbols representing a formula is a compressed enumeration of a k-dimensional space decomposed to equal-size k-dimensional hypercubes, where k is the number of variables in the formula. The compression technique in PIE is inspired by the Run Length Encoding (RLE) algorithm. In RLE each chain of identical symbols in a sequence is replaced by a positive integer followed by the symbol. The integer represents the number of times the symbol is repeated in the chain. For example, *aabbbbbbbccccc* is encoded to $2a7b5c$. Despite its simplicity, RLE has many applications in more complex compression algorithms such as JPEG. Another consideration in the compression technique in PIE exploits the continuous nature of the problem domain; that is, the symbolic representation of a formula can be compressed considering the continuity of the boundary of the shape corresponding to the solution of a formula.

The hypercube used for decomposing the space can have unequal size in different dimensions allowing for the approximation of the constraint with different precision in different dimensions. The inequalities are represented by progressive enumeration of the hypercubes parallel to the coordinate system axes. This approach is very similar to the solid modeling scheme known as Flat Spatial Occupancy Enumeration [108].

Every variable in a formula is associated with an initial interval used for the creation of the PIE sequence that represents the formula. A PIE sequence is a symbolic representation of a subregion of a *k*-dimensional space quantised to a specific resolution and confined in a convex hall defined by the boundary of intervals specified for each dimension.

Although promising, PIE suffers from a pitfall common to most constraint solvers, that is, the memory requirement is exponential with respect to the number of variables in the constraint set. We intend to equip PIE with a preprocessing rewrite rule technique that minimises the number of variables in the set before passing the constraints to PIE.

7.3 Final Notes

There are many challenging problems to be tackled on the way to a practical LSD, involving a variety of disciplines in computer science, including constraint solving, visual programming, design languages, constraint logic programming, user-interface design, solid modeling, programming by demonstration, and empirical studies. LSD provides a rich context for pursuing research in a variety of areas. The findings in each, although motivated by LSD, could potentially have wider impact.

There has been little or no empirical study of the effect of computer aided design tools and design languages on the performance of a designer. Design languages and CAD systems today are far from ideal and enable only part of the design process. There are still many opportunities for improving CAD systems.

The usefulness of a logic-based design language needs to be empirically verified. However, such studies cannot be performed in the absence of at least a prototype implementation. Although there exist some techniques for evaluating a user interfaces in the absence of a software implementation such as paper modeling, considering the complex 3D nature of CAD systems, it seems unlikely that such approaches would be very useful. We note that the 3D nature of CAD systems is considered to be one of the major contributing factors in the slow progress of CAD systems for mechanical design vis-à-vis those for digital design [134].

Bibliography

[1] Adobe Systems Inc., *Photoshop 6.0 User Guide*, 2000.

[2] J. Agusti, J. Puigegur, D. Robertson, A Visual Syntax for Logic and Logic Programming, *Journal of Visual Languages and Computing*, Volume 9, Number 4, pp. 392-427, 1998.

[3] R. St. Amant, H. Lieberman, R. Potter, and L. Zettlemoyer, Visual Generalization in Programming by Example, *Communications of the ACM*, Volume 43, Issue 3, ACM Press, New York, pp. 107-114, March 2000.

[4] E.K. Antonsson, The Potential for Mechanical Design Compilation, *Research in Engineering Design,* Volume 9, Number 4, pp 191-194, 1997.

[5] Autodesk Inc., *AutoCAD Mechanical 6 User's Guide*, 2001.

[6] Autodesk Inc., *Autodesk Inventor 5*, 2001.

[7] Autodesk Inc., *AutoLISP Release 12 Programmers Reference Manual*, 1992.

[8] J. Backus, Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, *Communications of the ACM (ACM Turing Award Lecture)*, Volume 21, Issue 8, ACM Press, New York, pp. 613-641, 1978.

[9] J. Backus, J. H. Williams, E. L. Wimmers, An Introduction to the programming language FL, *Research Topics in Functional Programming,* Chapter 9, pp. 219-247, D.A. Turner (Ed.), Addison-Wesley, Reading, MA, 1990.

[10] J. Backus, J. H. Williams, E. L. Wimmers, P. Lucas, A. Aiken, *FL Language manual, Parts 1 and 2*, Technical Report RJ 7100, IBM Almaden Research Center, Almaden, CA, October 1989.

[11] O. Banyasad, P. T. Cox, An Automatic Layout Algorithm for Lograph, *Proceedings of the 2004 IEEE Symposium on Visual Languages and Human-Centric Computing*, Rome, pp. 139-146, 2004.

[12] O. Banyasad, *A Visual Programming Environment for Autonomous Robots*, MCompSci thesis, Dalhousie University, 2000.

[13] O. Banyasad, P. T. Cox, Defining Behaviours for Solids in a Logic-Based Visual Design Environment, *Proceedings of the 2002 IEEE Symposia on Human-Centric Computing: End-User Programming*, Washington, pp. 93-95, 2002.

[14] O. Banyasad, P. T. Cox, Design and Implementation of an Interpreter Engine for a Visual Logic Programming Language, *Proceedings of the Second CologNet Workshop on Implementation Technology for Computational Logic Systems (ITCLS'03)*, September 9, Pisa, Italy, pp. 39-50, 2003.

[15] O. Banyasad, P. T. Cox, Integrating Design Synthesis and Assembly of Structured Objects in a Visual Design Language, *The Theory and Practice of Logic Programming,*, 5(6), Cambridge University Press, pp. 601-621, 2005.

[16] O. Banyasad, P. T. Cox, On Translating Geometric Solids to Functional Expressions, *Proceedings of The Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, Uppsala, Sweden, pp. 44-55, 2003.

[17] F. Benhamou, Interval constraint logic programming, *Constraint programming: basics and trends*, A. Podelski, editor, volume 910, LNCS, Springer-Verlag, pp. 1-21, 1995.

[18] F. Benhamou, F. Goualard, Universally Quantified Interval Constraints, *Proceedings of CP'2000*, pp. 67-82, 2000.

[19] Bentley Systems Inc., *MicroStation V8 2004 Edition User Guide*, 2004.

[20] A. F. Blackwell, K. N. Whitley, J. Good, M. Petre, Cognitive Factors in Programming with Diagrams, *Artificial Intelligence Review*, 15(1/2), pp. 95-114, 2001.

[21] P. Bouvier, Visual tools to debug Prolog IV programs, *Analysis and Visualization Tools for Constraint Programming: Constraint Debugging*, pp. 177-190, 2000.

[22] J. Branke, Dynamic Graph Drawing, *Drawing Graphs*, M. Kaufmann and D. Wagner, editors, volume 2025, LNCS, Springer-Verlag, pp. 228-246, 2001.

[23] D. C. Brown, B. Chandrasekaran, Expert Systems for a class of mechanical design activity, *Knowledge Engineering in Computer-Aided Design*, ed. J. S. Gero, Amsterdam, North-Holland, pp 259-282, 1985.

[24] M. Cameron, M Garcia de la Banda, K. Marriot, P. Moulder, ViMer: A Visual Debugger for Mercury, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practices of Declarative Programming (PPDP'03)*, Uppsala, Sweden, pp. 56-66, 2003.

[25] C. N. Carlson, *Grammatical Programming: an Algebraic Approach to the Description of Design Spaces*, PhD thesis, Department of Architecture, Carnegie Mellon University, 1993.

[26] S. C. Chase, Design Modeling with Shape Algebras andFormal Logic, *Proceedings of ACADIA (Association for Computer-Aided Design in Architecture) '96 Conference*, Tucson, AZ, October 31-November 3, 1996.

[27] J. C. Cleary, Logical Arithmetic, *Future Computing Systems*, 2 (2), pp. 125-149, 1987.

[28] R. F. Cohen, G. Di Battista, R. Tamassia, I. G. Tollis, and P. Bertolazzi, A Framework for Dynamic Graph Drawing, *Proceedings of the Eighth Annual Symposium on Computational Geometry*, Berlin, Germany, pp. 261-270, 1992.

[29] A. Colmerauer, An Introduction to Prolog-III. *Communications of the ACM*, Volume 33, Number 7, ACM Press, New York, pp. 69-90, July 1990.

[30] M. C. Cooper, An optimal k-consistency algorithm, *Artificial Intelligence*, 41(1), pp. 89-95, 1989.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts, 1997.

[32] J. Corney, *3D Modeling with the ACIS kernel and toolkit*, John Wiley & Sons Ltd., 1997.

[33] R. D. Coyne, M. A. Rosenman, A. D. Radford, J. S. Gero, Innovation and creativity in knowledge-based CAD, *Expert Systems in Computer-Aided Design*, ed. J. S. Gero, Amsterdam, North-Holland, pp 435-465, 1987.

[34] P.T. Cox, F.R. Giles, T. Pietrzykowski, Prograph: a step towards liberating programming from textual conditioning, *Proc. 1989 IEEE Workshop on Visual Programming*, Rome (Oct 1989), 150-156. Reprinted in *Visual Object-Oriented Programming: Concepts and Environments*, M. Burnett, A. Goldberg, & T.G. Lewis (Eds), Manning Publications, 1995.

[35] P.T. Cox, T. Pietrzykowski, Incorporating equality into logic programming via Surface Deduction, *Annals of Pure and Applied Logic 31*, North Holland, pp 177-189, 1986.

[36] P.T. Cox, T. Pietrzykowski (1985), LOGRAPH: a graphical logic programming language. *Proceedings IEEE COMPINT 85*, Montreal, pp 145-151.

[37] P.T. Cox, T. Smedley, A Formal Model for Parameterized Solids in a Visual Design Language, *Journal of Visual Languages and Computing*, Volume 11, Number 6, Academic Press, pp. 687-710, 2000.

[38] P.T. Cox, T. Smedley, LSD: A Logic Based Visual Language for Designing Structured Objects, *Journal of Visual Languages and Computing*, Volume9, Number 5, Academic Press, pp. 509-534, 1998.

[39] I. F. Cruz, A. Garg. Drawing Graphs by Example Efficiently: Trees and Planar Acyclic Digraphs, *Graph Drawing*, pp. 404-415, 1994.

[40] A. Cypher, EAGER: Programming Repetitive Tasks by Example, *Human Factors in Computing Systems*, New Orleans, LA, pp. 33-39, April1991.

[41] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, *Computational Geometry Algorithms and Applications*, second edition, Springer-Verlag 1998.

[42] S. Diehl, C. Gorg, A. Kerren, Preserving the Mental Map using Foresighted Layout, *Proceedings of the Joint Eurographics-IEEE TVCG Symposium on Visualization*, VisSym 2001, pp. 175-184, 2001.

[43] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, F. Berthier, The Constraint Logic Programming Language CHIP, *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, Tokyo, Japan, pp 693-702, December 1988.

[44] P. Eades, A heuristic for graph drawing, *Congressus Numerantium*, 42, pp. 149-160, 1984.

[45] C. Eastman, M. Henrion, GLIDE: A language for design information systems, *ACM Computer Graphics*, 11, 2, pp 24-33, 1977.

[46] M. Eisenstadt, M. Brayshaw, A fine-grained account of Prolog execution for teaching and debugging, *Instructional Science* , 19(4/5), pp. 407-436, 1990.

[47] M. Eisenstadt, M. Brayshaw, The Transparent Prolog Machine (TPM): An execution model and graphical debugger for logic programming, *Journal of Logic Programming*, 5(4), pp. 277-342, 1988.

[48] M. Engeli, V. Hrdliczka, *EUKLID-eine Einfiihrung*, Fides Rechenzentrum, Zurich, 1974.

[49] W. Espelage, E. Wanke, The Combinatorial complexity of masterkeying, *Mathematical Methods of Operations Research*, 52, pp. 325-348, 2000.

[50] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, *Computer Graphics: Principles and Practice*, 2nd ed. in C, Addison-Wesley, 1996.

[51] S. Fortune, A sweep line algorithm for Voronoi diagrams, *Algorithmica*, 2, pp. 153-174, 1987.

[52] E. C. Freuder, Synthesizing constraint expressions, *Communications of the ACM*, Volume 21, Issue 11, ACM Press, New York, pp. 958-966, 1978.

[53] T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy, M. Wallace, *Constraint Logic programming- An Informal Introduction*, Technical Report ECRC-93-5, European Computer-Industry Research Centre, 1993.

[54] J. S. Gero, Design Prototypes: A knowledge Representation Schema for Design, *AI Magazine*, 11(4), pp. 26-36, 1990.

[55] F. Glover and M. Laguna, Tabu Search, *Modern Heuristics for Combinatorial Problems*, C.R. Reeves, editor, John Wiley & Sons, Inc. New York, pp. 70-150, 1993.

[56] M. M. Godse, *A parametric design methodology for concurrent engineering*, University of Iowa, Ph.D. Thesis, 1991.

[57] T. R. G. Green, Building and Manipulating Complex Information Structures: Issues in Prolog Programming, In P. Brna, B. du Boulay and H. Pain (Eds.), *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*, Stamford CT: Ablex, Chapter 1, pp. 7-27, 1999.

[58] T. R. G. Green, M. Petre, Usability Analysis of Visual Programming Environments: A Cognitive Dimension's Framework, *Journal of Visual Languages and Computing*, Volume 7, pp.131-174, 1996.

[59] C. Han and C. Lee, Comments on Mohr and Henderson's Path Consistency Algorithm, *Artificial Intelligence*, 36, pp. 125-130, 1988.

[60] J. Heisserman, R. Woodbury, Generating Languages of Solid Models, *Proceedings of the Second ACM/IEEE Symposium on Solid Modeling and Applications*, pp. 103-112, 1993.

[61] C. M. Hoffmann, *Geometric and Solid Modeling: An Introduction*, Morgan Kaufmann, San Mateo, CA, 1989.

[62] D. A. Huffman, Impossible Objects as Nonsense Sentences, *Machine Intelligence* vol. 6, pp. 295-323, American Elsevier, New York, 1971.

[63] J. Jaffar, J. L. Lassez, Constraint Logic Programming, *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ACM Press, New York, pp. 111-119, 1987.

[64] K.M. Kahn, V.A. Saraswat, Complete Visualizations of Concurrent Programs and their Executions, *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pp. 7-15, 1990.

[65] T. Kamps, K. Reichenberger, Automatic Layout Based on Formal Semantics, *Proceedings of the 1994 Workshop on Advanced Visual Interfaces*, Bari, Italy, pp 231-233, 1994.

[66] E. Knill, P.T. Cox, T. Pietrzykowski, Equality and abductive residua for Horn clauses, *Theoretical Computer Science*, 120, pp. 1-44, 1993.

[67] R. A. Kowalski, *Logic for problem solving*, North-Holland, 1979.

[68] V. Kumar, Algorithms for constraint satisfaction problems: A survey, *AI Magazine*, 13(1), 1992.

[69] D. LaCourse, *thinkdesign and the Importance of Today's 3D Kernel Technology*, think3, Santa Clara, California, 2000.

[70] D. Ladret, M. Rueher, VLP: a Visual Programming Language, *Journal of Visual Languages and Computing*, Volume 2, Number 2, pp. 163-189, 1991.

[71] T. Lau. *Programming by Demonstration: a Machine Learning Approach*, PhD thesis, University of Washington, 2001.

[72] H. Lieberman, B. Nardi, and D. Wright, Training Agents to Recognize Text by Example, *Proceedings of the 1999 International Conference on Autonomous Agents (Agents'99)*, pp. 116-122, 1999.

[73] L. S. Lopes, L. M. Camarinha-Matos, Robot programming by demonstration in the assembly domain, *Proceedings of Mechatronics'96*, Portugal, pp.125-130, 1996.

[74] K. A. Lyons, H. Meijer, D. Rappaport, Algorithms for Cluster Busting in Anchored Graph Drawing, *Journal of Graph Algorithms and Applications*, vol. 2, no. 1, pp. 1-24, 1998.

[75] R. G. McDaniel and B. A. Myers, Building Applications Using Only Demonstration, *IUI'98: 1998 International Conference On Intelligent User Interfaces*, San Francisco, CA, pp. 109-116, January 6-9, 1998,

[76] P. Meseguer, Constraint satisfaction problem: an overview, *AI Communications*, 2(1), 1989.

[77] A. Michail. *Imitation: An Alternative to generalization in Programming by Demonstration Systems*, University of Washington, Technical Report UW-CSE-98-08-06, 1998.

[78] K. Misue, P. Eades, W. Lai, and K. Sugiyama, Layout Adjustment and the Mental Map, *Journal of Visual Languages and Computing*, Volume 6, Number 2, pp. 183-210, 1995.

[79] R. Mohr and T. C. Henderson, Arc and Path consistency revisited, *Artificial Intelligence*, 25, pp. 65-74, 1986.

[80] U. Montanari, Networks of constraints: fundamental properties and applications to picture processing, *Information Sciences*, 7, pp. 95-132, 1974.

[81] R. E. Moore, *Interval analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1966.

[82] R. E. Moore, *Methods and Applications of Interval Analysis*, SIAM Studies in Applied mathematics, 1979.

[83] P. Mulholland, Incorporating Software Visualization into Prolog teaching: a challenge, a restriction, and an opportunity, *Proceedings of ICLP'97 Postconference Workshop on Logic Programming Environment*, pp. 33-42, 1997.

[84] S. Munch, J. Kreuziger, M. Kaiser, R. Dillmann, Robot Programming by Demonstration (RPD) - Using Machine Learning and User Interaction Methods for the Development of Easy and Comfortable Robot Programming Systems, *Proceedings of the 24th International Symposium on Industrial Robots*, pp. 685–693, 1994.

[85] B. Myers and R. McDaniel, Demonstrational Interfaces: Sometimes You Need a Little Intelligence; Sometimes You Need a Lot, *Your Wish is My Command*, Henry Lieberman, Ed. San Francisco: Morgan Kaufmann, pp. 45-60, 2001.

[86] L. R. Nackman, M. A. Lavin , R. H. Taylor , W. C. Dietrich, Jr. , D. D. Grossman, AML/X: A Programming Language for Design and Manufacturing, *Proceedings of 1986 Fall Joint Computer Conference*, pp. 145-159, 1986.

[87] M. A. Najork, S.M. Kaplan, The CUBE Language, *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pp. 218-224, 1991.

[88] J. C. Nash, *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation, 2nd ed.*, Bristol, England: Adam Hilger, 1990.

[89] E. Neufeld, A. J. Kusalik, M. Dobrohoczki, Visual Methaphors for Understanding Logic Program Execution, *Proceedings of Graphics Interface 1997*, pp. 114-120, 1997.

[90] W. Older and A. Vellino, Extending Prolog with Constraints Arithmetic on Real Intervals, *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, 1990.

[91] W. Older and A. Vellino, Constraint Arithmetic on Real Intervals, *Constraint Logic Programming: Selected Papers*, F. Benhamou & A. Colmerauer eds., The MIT Press, Cambridge, MA, 1993.

[92] A. Paoluzzi, *Geometric Programming for Computer-Aided Design*, John Wiley & Sons Ltd, England, 2003.

[93] A. Paoluzzi, V. Pascucci, and M. Vicentino, Geometric Programming: A Programming Approach to Geometric Design, *ACM Transactions on Graphics*, Vol. 14, No. 3, pp. 266-306, 1995.

[94] A. Paoluzzi, C. Sansoni, Programming language for solid variational geometry, *Computer Aided Design*, 24, 7, pp. 349-366, 1992.

[95] M. J. Patel, B. du Boulay, C. Taylor, Comparison of contrasting Prolog trace output formats, *International Journal of Human-Computer Studies*, 47, pp. 289-322, 1997.

[96] L.F. Pau, H. Olason, Visual Logic Programming, *Journal of Visual Languages and Computing*, pp. 3-15, 1991.

[97] M. Petre, A. Blackwell, T. Green, Cognitive Questions in Software Visualization, In J. Stasko, J. Domingue, M. Brown, B. Price (eds.) *Software Visualization: programming as a Multi-media Experience*,Cambridge, MA, MIT Press, pp. 453-480, 1998.

[98] Pictorius Incorporated, *Prograph CPX User's Guide*, 1993.

[99] R. J. Popplestone, A. P. Ambler, I. M. Bellos, An Interpreter for a Language for Describing Assemblies, *Artificial Intelligence*, volume 14, pp. 79-107, 1980.

[100] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in FORTRAN: The Art of Scientific Computing, 2nd ed.*, Cambridge, England: Cambridge University Press, 1992.

[101] B. A. Price, I. S. Small, and R. M. Baecker, A Principled Taxonomy of Software Visualization, *Journal of Visual Languages and Computing,* Volume 4, Number 3, pp. 211-266, 1993.

[102] J. Puigsegur, W.M. Schorlemmer, J. Agusti , From Queries to Answers in Visual Logic Programming, *Proceedings of the 1997 IEEE Symposium on Visual Languages*, pp. 102-109, 1997.

[103] S. Raghothama, V. Shapiro, Boundry representation deformation in parametric solid modeling, *ACM Transactions on Graphics*, 17(4), pp 259-286, 1998.

[104] S. Raghothama, V. Shapiro, Topological framework for part families, *Proceedings of the seventh ACM symposium on Solid modeling and applications*, pp 1-12, June 17-21, Saarbrucken, Germany, 2002.

[105] A. Rappoport, Geometric Modeling: a New Fundamental Framework and its Practical Implications, *Proceedings of the Third ACM/Siggraph Symposium on Solid Modeling and*

Applications (Solid Modeling '95), May 1995, Salt Lake City, ACM Press, pp. 31-42, 1995.

[106] A. Rappoport, The Generic Geometric Complex (GGC): a Modeling Scheme for Families of Decomposed Poinsets, *Proceedings of the Fourth ACM/Siggraph Symposium on Solid Modeling and Applications* (Solid Modeling '97), ACM Press, Atlanta, pp. 19-30, May 1997.

[107] A. Rau-Chaplin, B. MacKay-Lyons, and P. Spierenburg, The LaHave House Project: Towards and Automated Architectural Design Service, *Proceedings of the International Conference on Computer Aided Design (*CAD*EX'96)*, IEEE Computer Society Press, pp. 25-31, 1996.

[108] A. A. G. Requicha, Representation for rigid solids: Theory, methods and systems, *ACM Computer Survey*, 12, 4, 437-464, 1980.

[109] J. Sam, *Constraint Consistency Techniques for Continuous Domains*, PhD thesis, Ecole polytechnique federale de Lausanne, 1995.

[110] J. Sam-Haroud, B. V. Faltings, Consistency techniques for continuous constraints, *Constraints*, pp. 85-118, 1996.

[111] C. Schulte, Oz Explorer: A Visual Constraint Programming Tool, *Proceedings of the Fourteenth International Conference on Logic Programming* , The MIT Press, Leuven, Belgium, pp. 286-300, July 1997.

[112] H. Senay, S. G. Lazzeri, Graphical representation of logic programs and their behaviour, *Proceedings of the 1991 Workshop on Visual Languages*, pp 25-31, 1991.

[113] Shape Data Limited and Electronic Data Systems Corporation, *Parasolid v5.0 Programming Reference Manual*, Cambridge, England,1992.

[114] V. Shapiro , D. L. Vossler, What is a parametric family of solids?, *Proceedings of the third ACM symposium on Solid modeling and applications*, Salt Lake City, Utah, United States, pp. 43-54, May 17-19, 1995.

[115] B. Shneiderman, R. Mayer, D. McKay, P. Heller, Experimental Investigations of the Utility of Detailed Flowcharts in Programming, *Communications of the ACM*, Volume 20, Issue 6, ACM Press, New York, pp. 373-381, 1977.

[116] T.J. Smedley, P.T. Cox, Visual Languages for the Design and Development of Structured Objects, *Journal of Visual Languages and Computing,* Volume 8, Number 1, Academic Press, pp. 57-84, 1997.

[117] J. M. Snyder, *Generative modeling for computer graphics and CAD*, Academic Press, London, 1992.

[118] J. M. Snyder, J.T. Kajiya, Generative Modeling: A Symbolic System for Geometric Modeling, *ACM Siggraph*, pp. 369-378,1992.

[119] SolidWorks Corporation, *SolidWorks 2006 What's New*, 2006.

[120] Spatial Technology Inc., *ACIS Geometric Modeler*, Version 4.0, 1997.

[121] L.L. Spratt, A.L. Ambler, A Visual Logic Programming Language Based on Sets and Partitioning Constraints, *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pp. 204-208, 1993.

[122] *Standard VHDL Language Reference Manual*-Std 1076-1987. IEEE, 1988.

[123] M. -A. D. Storey, F. D. Fracchia, H. A. Muller. Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration, *Proceedings of the 5th International Workshop on Program Comprehension*, pp. 17–28, Dearborn, Michigan, May 1997.

[124] K. Sugiyama, K. Misue, Graph drawing by magnetic-spring model, *Journal of Visual Languages and Computing (special issue on Graph Visualisation)*, edited by I. F. Cruz and P. Eades, Volume 6, Number 3, 1995.

[125] E. Süli, D. Mayers, *An Introduction to Numerical Analysis*, Cambridge University Press, 2003.

[126] Swedish Institute of Computer Science, *SICStus Prolog User's Manual* release 3.12.2, 2005.

[127] R. Tamassia. Graph Drawing, In *Handbook of Discrete and Computational Geometry* (J. E. Goodman and J. O'Rourke, ed.), CRC Press, pp. 815-832, 1997.

[128] C. Taylor, J. B. H. du Boulay, M. J. Patel, Textual Tree Trace Notation for Prolog: an Overview, *Proceedings of Post-Conference Workshop on Logic Programming and Education, International Conference on Logic Programming*, Santa Margherita Ligure, Italy, pp. 73-80, 1994.

[129] E. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, London, 1995.

[130] K.J. Waldron, Drafting a New Plan for Design, *Mechanical Engineering, Design Supplement*, pp. 37-38, November 1999.

[131] D. L. Waltz, Understanding line drawings of scenes with shadows, *The Psychology of Computer Vision*, ed. P. W. Winston, McGraw-Hill, 1975.

[132] D. E. Whitney, Why mechanical design cannot be like VLSI design, *Research in Engineering Design,* Volume 8, Number 3, pp. 139-150, 1996.

[133] D. Wolber. Pavlov: an interface builder for designing animated interfaces, *ACM Transactions on Computer-Human Interaction (TOCHI)*, 4(4), pp. 347-386, 1997.

[134] J. Woodwark, *private communication*, October 2002.

Appendix A: LSD Semantics

In this Appendix, we will present the semantics of LSD. Each definition is marked with a tag O, M, or N, indicating whether the definition is reproduced from [37], is a modified version of an earlier definition according to [37], or is an entirely new definition introduced in this thesis, respectively. In addition, each item tagged M or N is numbered in the same way as in the body of the thesis.

We assume the existence of a design space $\mathcal{D}$, a set S of solids in $\mathcal{D}$, and a set O of operations in $\mathcal{D}$. To each selector occurring in an operation in O, corresponds a unique function symbol with arity equal to the size of the selector called an *interface symbol.* To each solid in k variables corresponds a k-ary symbol called an *explicit symbol* which is either a function symbol or a predicate symbol. To each n-ary operation in O, corresponds a unique n-ary predicate symbol called a *link symbol.* The sets of explicit predicate symbols and link symbols are disjoint. Remaining predicate symbols are called *implicit symbols.* Similarly, the sets of explicit function symbols and interfaces are disjoint. Remaining function symbols are called *abstract.*

In the following if X is a selector, solid, or operation, we denote the corresponding symbol by {X}. Table A.1 summarises symbols in LSD and their relations to programming constructs in Lograph, and the design space.

(O) Definition: A *function cell* is a literal of the form $x = \mathbf{f}(y_1,\ldots,y_k)$ where **f** is some *k*-ary function symbol, $k \geq 0$, x is a variable, and y_i is a variable for each i $(0 \leq i \leq k)$. Each y_i is called a *terminal* of the cell, and x is called the *root*.

(O) Definition: An *implicit component* (*i-component*) is a literal of the form $\mathbf{p}(v_1,\ldots,v_k)$ where **p** is some *k*-ary implicit symbol, $k \geq 0$, and for each i $(0 \leq i \leq k)$, v_i is a variable called a *terminal* of the component. Each terminal of an i-compoenent is classified as a *simple terminal* or as an *explicit group terminal of type* **L** where **L** is a selector of some operation, and the signature of an i-component $\mathbf{p}(v_1,\ldots,v_k)$ is a list $(F_1,\ldots,F_k)$ where for each i $(0 \leq i \leq k)$, $F_i = \mathbf{s}$ if v_i is a simple terminal, and $F_i = \mathbf{L}$ if v_i is a group terminal of type **L**.

(O) Definition: A *component design* (or simply *design*) consists of a set of cases with no variables in common, such that the heads have the same implicit symbol and signature. A *case* is a flat clause the head of which is a literal of the same form as an implicit component, with simple terminals, link terminals and signature defined analogously. The *body* of a case is a set of function cells, components and links, satisfying the following conditions:

- No variable occurring in an e-component or link occurs anywhare else in the case, with the exception of the implicit group terminals of the component or link.
- Any variable in the case which occurs as a group terminal or implicit group terminal has exactly two occurrences which must both be of the same type. If one of these occurrences is in a component, the other must be in the head or a link, otherwise both occurrences must be in the head.

(M) Definition 6.1: An *explicit component* (*e-component*) consists of

- a literal called the *anchor* of the component which is either of the form r={**Φ**}$(v_1,\ldots,v_n)$ (*primitive anchor*) or of the form {**Φ**}$(r,v_1,\ldots,v_n)$ (*defined anchor*) where **Φ** is a solid in n variables for some $n \geq 0$, r is called *the root*, and $r,v_1,\ldots,v_n$ are distinct variables; and
- for each exposed interface φ of **Φ**, one literal of the form w={**L**}$(v_{i_1},\ldots,v_{i_k})$, called a *group*, where **L** is the selector corresponding to φ, k is the size of **L**, and $i_1,\ldots,i_k$ is the sequence of variables of **Φ** required by φ. The variable w is called an *implicit group terminal of type* **L**.

(N) **Definition 6.2:** An *anchor-knot graph* is a connected graph of anchors, knots and constraint cells such that

- no root of an anchor or knot occurs in the terminals of a constraint cell.
- no root of an anchor occurs as the root of another anchor.
- each root of a knot occurs exactly once as a root of an anchor.

(N) **Definition 6.3:** If **A** is a defined anchor, the *design of* **A** is a design **D** with the same signature as **A** such that for each case **S** of **D**, the body of **S** consists of an anchor-knot graph, such that the first terminal of the head of **S** occurs as the root of exactly one anchor in the body of **S** and each non-root terminal of an anchor that does not occur as a terminal of a constraint cell in the body of **S** occurs exactly once in the head of **S**, and each terminal of a constraint cell that does not occur as a non-root terminal of an anchor occurs exactly once in the head of **S**.

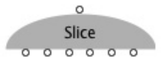
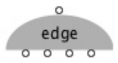
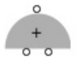
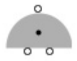
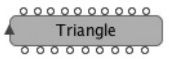
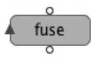
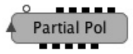
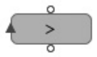
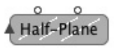
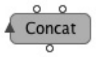
(M) **Definition 6.4:** A *link* for an m-ary operation $\otimes=(((a_1,\ldots,a_m),q),(z_1\bullet\ldots\bullet z_m\bullet p),\mathcal{F},\mathcal{L},\mathbf{C})$ consists of

- a literal, called a *knot*, of the form $\{\mathcal{F}\}(v_1\bullet\ldots\bullet v_m\bullet g)$ where v_i, called *a root*, is a variable for each i $(1 \leq i \leq m)$ and g is a sequence of variables the same length as q;
- one constraint cell of the form {**C**}$(u_1\bullet\ldots\bullet u_m\bullet e\bullet g)$ the specification of which implements **C**, where e is a sequence of variables the same length as p, and for each i $(1 \leq i \leq m)$ u_i is a sequence of variables the same length as z_i; and

- for each i ($1 \leq i \leq m$) a literal w_i={$\mathbf{L}_i$}(u_i), also called a *group*, such that $\mathbf{L}_i$ is the i^{th} selector of ⊗. For each i ($1 \leq i \leq m$) w_i is called an *implicit group terminal of type* $\mathbf{L}_i$.

Table A.1 shows the symbols in LSD.

Table A.1: Symbols in LSD.

Symbols	Explicit	Link	Interface	Implicit	Constraint		Abstract
Function	Primitive Anchor 	–	Group 	–	Special Functors 		Function Cell 
Predicate	Defined Anchor 	Knot 	–	i-Component (Design) 	Relation Cell 	Constraint Cell (Specification) 	Literal Cell (Definition) 
Denote	**Solid**	**Operation**	**Selector & Interface**		**Constraints**		**Functions & Literals**
Languge Extension	LSD				**Lograph+C**		**Pure Lograph**
Usage	**Geometric Computation**				**Numeric and Constraint Computation**		**Symbolic Computation**

Appendix B: Formal Model

Here, we present the extension to the formal model for solids according to [37]. Each definition is marked with a tag O, M, or N, indicating whether the definition is reproduced from [37], is a modified version of an earlier definition according to [37], or is an entirely new definition introduced in this thesis, respectively. All lemmas in this Appendix are new and therefore are not tagged.

If x and y are sequences, we denote the concatenation of x and y by $x{\bullet}y$. For example, if x is $x_1,x_2,\ldots,x_n$ and y is $y_1,y_2,\ldots,y_m$ then $x{\bullet}y$ means $x_1,x_2,\ldots,x_n,y_1,y_2,\ldots,y_m$. If x is a sequence and y is not a sequence, for consistency, by $x{\bullet}y$ we mean $x_1,x_2,\ldots,x_n,y$ and by $y{\bullet}x$ we mean $y,x_1,x_2,\ldots,x_n$. We use () to represent a sequence of length zero. If x is a sequence of length n and $1\leq i \leq n$, we denote by $\bar{x}_i$ the sequence of length n–1 obtained by removing the i^{th} element from x. If x is a sequence of length n and $p=p_1,p_2,\ldots,p_k$ is a sequence of distinct integers such that $1\leq p_j \leq n$ for all $1\leq j \leq k$, by x_p we mean the sequence $x_{p_1},x_{p_2},\ldots,x_{p_k}$ and by $\bar{x}_p$ we mean a sequence obtained after removing items in x_p from x. If x is a sequence of length n, we denote by x_{m^*} and x_{*m} for some $m \leq n$, the length m prefix and length m suffix of x, respectively. If x is a sequence, we may also use x to denote the set of elements in x.

If **C** is a formula, by var(**C**) we mean the set of all free variables of **C**. If **C** is a formula, and a is a term, we use the standard notation $\mathbf{C}_x[a]$ for the formula obtained from **C** by substituting a for every occurrence of x. We extend this notation to sequences of variables and terms of the same length. That is, if $x_1,x_2,\ldots,x_n$ is a sequence of variables and $a_1,a_2,\ldots,a_n$ is a sequence of terms, then $\mathbf{C}_{x_1,x_2,\ldots,x_n}[a_1,a_2,\ldots,a_n]$ is the formula obtained from **C** by substituting a_i for every occurrence of x_i for each i ($1\leq i \leq n$). We might also combine single variables and sequences of variables in this notation. For example, $\mathbf{C}_{x,y}[a,b]$ means that each of x and y is either a single variable or a sequence, and each of a and b is either a single term or sequence of terms, of the same length as x and y, respectively. Another example, is $\mathbf{C}_{x\bullet y}[a]$ which means that a is a sequence of terms such that $|a| = |x\bullet y|$. $\mathbf{C}_{x\bullet y,z}[a,b]$ means that a is a sequence of terms such that $|a|=|x\bullet y|$, and b is a sequence of terms iff z is a sequence of variables of the same length as b.

We use a set of formulae as an alternate notation for a formula which is the conjunction of the formulae in the set such that none of them are conjunctions. The conjunctions of two or more sets of formulae is understood to be the conjunction of the formulae in the sets. If **C** is a set of formulae, by $\mathbf{C}^=$ we mean the set of equalities in **C**. If x is a variable we denote by $\mathbf{C}_x$ a set of formulae of the form $\{\ \mathbf{E}_x[u] \mid \mathbf{E} \in \mathbf{C}\ ,\ \mathrm{var}(\mathbf{E}_x[u]) \neq \varnothing\ \}$, where u is some constant. If x is a set of variables, $\mathbf{C}_x$ is defined in the obvious way. Clearly, $\mathbf{C}_x$ is not well defined since it depends on the choices of u. However, that will not be important in the context in which we will use this notation. If **C** is a set of formulae and $x \subseteq \mathrm{var}(\mathbf{C})$, by $\mathbf{C}^x$ we mean the set of formulae $\{\ \mathbf{E} \mid \mathbf{E} \in \mathbf{C}\ ,\ x \cap \mathrm{var}(\mathbf{E})=\varnothing\ \}$. If x is a variable, $\mathbf{C}^x$ has the obvious meaning $\mathbf{C}^{\{x\}}$.

If Φ is a partial function from A to B, by scope(Φ) we mean the set $\{\ x \mid x \in \mathrm{A}$ and Φ is defined for $x\ \}$ and by the image of Φ we mean the set $\{\Phi(x) \mid x \in \mathrm{scope}(\Phi)\}$.

If **C** is a formula, Γ is a partial function, and $x\bullet y\bullet u \subseteq \mathrm{var}(\mathbf{C})$, $\Phi(x\bullet y\bullet z)=[\mathrm{C}]\ \Gamma(x\bullet z\bullet u)$ defines the partial function Φ such that $a\bullet b\bullet c \in \mathrm{scope}(\Phi)$ if there is a unique d such that $\mathbf{C}_{x,y,u}[a,b,d]$ holds and $a\bullet c\bullet d \in \mathrm{scope}(\Gamma)$. If the partial function Γ can be defined uniquely and unambiguously by an expression **E**, then we can substitute **E** for Γ in the definition of Φ.

(O) **Definition 5.1:** A *design space in m dimensions over r properties* for some integers $m > 0$ and $r \geq 0$ is the set of all subsets of $\mathbf{R}^m \times \mathbf{R}^r$.

(M) **Definition 5.2:** A *solid* in a design space $\mathcal{D}$ in *n variables* for some $n \geq 0$ is a partial function Φ: $\mathbf{R}^n \rightarrow \mathcal{D}$ such that, if (v, p) and $(v,q) \in \Phi(y)$ for some $y \in \mathbf{R}^n$, then $p=q$. A *variable* of Φ is an integer i such that $1 \leq i \leq n$. Symbolic names may also be used to refer to the variables of a solid.

(N) **Definition 5.3:** A solid Φ in $\mathcal{D}$ is *valid* iff the image of Φ is not empty.

(M) **Definition 5.4:** If Φ and Ψ are solids in a design space $\mathcal{D}$, Φ and Ψ are said to be *equivalent*, denoted $\Phi \equiv \Psi$, iff they have identical images.

(M) **Definition 5.6:** If $\mathcal{D}$ is a design space and n is a positive integer, an *n-ary operation in* $\mathcal{D}$ is a quintuple $((a_1,\ldots,a_n),(z_1\cdot\ldots\cdot z_n),\mathcal{F},\mathcal{L},\mathbf{C})$ where

- $a_1,\ldots,a_n$ and $z_1\cdot\ldots\cdot z_n$ are sequences of distinct variables such that for each i ($1 \leq i \leq n$), z_i is a sequence of variables and $a_1,\ldots,a_n$ and $z_1\cdot\ldots\cdot z_n$ are disjoint.
- $\mathcal{F}$ is a partial function from $\mathcal{D}^n$ to $\mathcal{D}$ such that $\forall x \in \text{scope}(\mathcal{F})$, if (e,g) and $(e,h) \in \mathcal{F}(x)$, then $g=h$.
- $\mathcal{L}=(\mathbf{L}_1,\ldots,\mathbf{L}_n)$ is a sequence of formulae, called *selectors,* such that $\text{var}(\mathbf{L}_i)=a_i\cdot z_i$. The integer $|z_i|$ is called the *size* of $\mathbf{L}_i$.
- $\mathbf{C}$ is an open formula, called the *constraint* of the operation, such that $\text{var}(\mathbf{C})=z_1\cdot\ldots\cdot z_n$.

(M) **Definition 5.8:** If Φ is a solid in n variables and $\mathbf{L}$ is a selector of size k of some operation, where $\text{var}(\mathbf{L})=a\cdot z$ and $|z|=k$, then *an L-interface to* Φ is a partial function ϕ:$\mathbf{R}^n \rightarrow \mathbf{R}^k$ such that $\phi(u)=[u \in \text{scope}(\Phi),\mathbf{L}_{a,z}[\Phi(u),y)]]\ y$, and $\forall u,v \in \text{scope}(\Phi)$, if $\Phi(u)=\Phi(v)$ then $\phi(u)=\phi(v)$.

(M) **Definition 5.10:** Let $\otimes=((a_1,\ldots,a_n),(z_1\cdot\ldots\cdot z_n),\mathcal{F},\mathcal{L},\mathbf{C})$ be an n-ary operation; where $\mathcal{L} = (\mathbf{L}_1,\ldots,\mathbf{L}_n)$, and for each i ($1 \leq i \leq n$) let Φ_i be a solid in n_i variables, and ϕ_i be an $\mathbf{L}_i$-interface to Φ_i. A solid Ψ in $t= \sum_{i=1}^{n} n_i$ variables, called *the application of* $\otimes$ *to* $\Phi_1,\ldots,\Phi_n$ *via* $\phi_1,\ldots,\phi_n$ is defined as follows. If $y \in \mathbf{R}^t$ denote by y_1 the first n_1 elements of y, denote by y_2 the next n_2 elements of y and so forth, then $\Psi(y)$ is defined as

$$\Psi(y)=[\ \mathbf{C}_{z_1,\ldots,z_n}[\phi_1(y_1),\ldots,\phi_n(y_n)]\ ]\ \mathcal{F}(\Phi_1(y_1),\ldots,\Phi_n(y_n)).$$

Lemma 5.11: If Ψ is the solid defined in Definition (M) and $u \in$ scope(Ψ), then $u_i \in$ scope(Φ_i) for each i ($1 \leq i \leq n$), and $(\Phi_1(u_1),\ldots,\Phi_n(u_n)) \in$ scope($\mathcal{F}$).

(M) **Definition 5.14:** If Φ is a solid in n variables, **L** is a selector of size k of some operation, and ϕ is an **L**-interface to Φ, Φ is said to *expose* ϕ iff there exists a sequence $p_1,\ldots,p_k$ of variables of Φ, such that for every $y \in$ scope(Φ), for all i ($1 \leq i \leq k$) $\phi(y)_i = y_{p_i}$. Each of the variables p_i is said to be *required* by ϕ. The variables $p_1,\ldots,p_k$ are not necessarily distinct, and the sequence $p_1,\ldots,p_k$ is not necessarily unique.

(N) **Definition 5.15:** If Φ is a solid in n variables over $\mathcal{D}$, a *factoring* of Φ is a quintuple $(x,y,z,\mathbf{C},\Psi)$ where

- x, y, and z are disjoint sequences of distinct variables of lengths n, m and k, respectively, called the *input*, *internal* and *geometric* variables, respectively. (***a***)
- **C** is a set of formulae, called the *constraint*, such that var(**C**)$=x\bullet y\bullet z$. (***b***)
- $\forall s \in z$, s has only one occurrence in **C** and there exists a unique equality $\mathbf{E} \in \mathbf{C}$ such that var(**E**)$=\{s\}$ or var(**E**)$=\{s,l\}$ and $l \in x \cup y$ (***c***)
- Ψ is a solid in k variables in $\mathcal{D}$. (***d***)
- $\forall u \in \mathbf{R}^n$, if $u \in$ scope(Φ), then $\exists\ v \in \mathbf{R}^m$ and $\exists\ w \in \mathbf{R}^k$ such that $\mathbf{C}_{x,y,z}\ [u,v,w]$ is true. (***validity***)
- $\forall u \in \mathbf{R}^n$, $\forall v \in \mathbf{R}^m$, $\forall w \in \mathbf{R}^k$, if $\mathbf{C}_{x,y,z}[u,v,w]$ is true, then $u \in$ scope(Φ) iff $w \in$ scope(Ψ). (***conformity***)
- $\forall u \in \mathbf{R}^n$, $\forall v \in \mathbf{R}^m$, $\forall w \in \mathbf{R}^k$, if $u \in$ scope(Φ) and $\mathbf{C}_{x,y,z}[u,v,w]$ is true, then $\Phi(u)=\Psi(w)$. (***identity***)

The factoring is said to be *perfect* if the following condition also holds:

- $\forall u \in \mathbf{R}^n$, $\forall v \in \mathbf{R}^m$, $\forall w \in \mathbf{R}^k$, if $\mathbf{C}_{x,y,z}[u,v,w]$ is true, then $u \in$ scope(Φ). (***perfection***)

Lemma 5.17: If $\mathcal{D}$ is a design space, $\otimes$=$((a_1,\ldots,a_n),(p_1\cdot\ldots\cdot p_n),\mathcal{F},\mathcal{L},\mathbf{C})$ is an *n*-ary operation in $\mathcal{D}$, $\Phi_1,\ldots,\Phi_n$ are solids in $\mathcal{D}$, Ψ is the solid resulting from the application of $\otimes$ to $\Phi_1,\ldots,\Phi_n$ via interfaces $\phi_1,\ldots,\phi_n$, and $(x_i,y_i,z_i,\mathbf{C}_i,\Psi_i)$ is a factoring of Φ_i for each i ($1\leq i \leq n$), then $(x,y,z,\mathbf{X},\Delta)$ is a factoring of Ψ where x is $x_1\bullet\ldots\bullet x_n$, y is $y_1\bullet\ldots\bullet y_n$, z is $z_1\bullet\ldots\bullet z_n$, and

- $\mathbf{X} = \mathbf{C}_{p_1,\ldots,p_n}[\phi_1(x_1),\ldots,\phi_n(x_n)] \cup \mathbf{C}_1 \cup \ldots \cup \mathbf{C}_n$, and
- $\Delta(z) = \mathcal{F}(\Psi_1(z_1),\ldots,\Psi_n(z_n))$

Lemma 5.18: If in Lemma 5.17, $(x_i,y_i,z_i,\mathbf{C}_i,\Psi_i)$ is a perfect factoring of Φ_i for each i ($1 \leq i \leq n$), and $\mathbf{C}_{p_1,\ldots,p_n}[\phi_1(u_1),\ldots,\phi_n(u_n)]$ implies that $(\Phi_1(u_1),\ldots,\Phi_n(u_n)) \in$ scope($\mathcal{F}$), then $(x,y,z,\mathbf{X},\Delta)$ is a perfect factoring of Ψ.

(M) **Definition 5.20:** If $\mathcal{D}$ is a design space and n is a positive integer, an *n-ary operation in* $\mathcal{D}$ is a quintuple $(((a_1,\ldots,a_n),q),(z_1\cdot\ldots\cdot z_n\cdot p),\mathcal{F},\mathcal{L},\mathbf{C})$ where

- $a_1,\ldots, a_n, z_1,\ldots, z_n$ and $\mathcal{L}$ are as in Definition 5.6
- p and q are disjoint sequences of distinct variables such that $|q| \leq |p|$ and $p\cdot q$ is disjoint from $(a_1,\ldots,a_n)\cdot z_1\cdot\ldots\cdot z_n$. Variables in p are called the *extra variables* of the operation.
- $\mathcal{F}$ is a partial function from $\mathcal{D}^n \times \mathbf{R}^{|q|}$ to $\mathcal{D}$ such that $\forall x \in$ scope($\mathcal{F}$), if (e,g) and $(e,h) \in \mathcal{F}(x)$, then $g=h$.
- $\mathbf{C}$ is an open formula, called the *constraint* of the operation, such that var($\mathbf{C}$)=$z_1\cdot\ldots\cdot z_n\cdot p\cdot q$, and for each $s \in q$, there exists a unique equality $\mathbf{E} \in \mathbf{C}$ such that var($\mathbf{E}$)=$\{s\}$ or var($\mathbf{E}$)=$\{s,l\}$ and $l \in p$.

(M) **Definition 5.21:** Let $\otimes = (((a_1,\ldots,a_n),q),(z_1\cdot\ldots\cdot z_n\cdot p),\mathcal{F},\mathcal{L},\mathbf{C})$ be an *n*-ary operation; where $\mathcal{L} = (\mathbf{L}_1,\ldots,\mathbf{L}_n)$, and for each i ($1 \leq i \leq n$) let Φ_i be a solid in n_i variables, and ϕ_i be an $\mathbf{L}_i$-interface to Φ_i . A solid Ψ in $t = \sum_{i=1}^{n} n_i + |p|$ variables, called *the application of* $\otimes$ *to* $\Phi_1,\ldots,\Phi_n$ *via*

$\phi_1,\ldots,\phi_n$ is defined as follows. Suppose $d \in \mathbf{R}^{|q|}$ and $y \in \mathbf{R}^t$. If we denote by y_1 the first n_1 elements of y, denote by y_2 the next n_2 elements of y and so forth, then $\Psi(y)$ is defined as

$$\Psi(y) = [\mathbf{C}_{z_1,\ldots,z_n,p,q}[\phi_1(y_1),\ldots,\ \phi_n(y_n),y_{*|p|},d]]\ \mathcal{F}((\Phi_1(y_1),\ldots,\Phi_n(y_n)),d).$$

Lemma 5.22: If Ψ is the solid defined in Definition (M) and $u \in \text{scope}(\Psi)$, then $u_i \in \text{scope}(\Phi_i)$ for each i ($1 \le i \le n$), and $((\Phi_1(u_1),\ldots,\Phi_n(u_n)),c) \in \text{scope}(\mathcal{F})$ for some c.

Lemma 5.24: If $\mathcal{D}$ is a design space, $\otimes = (((a_1,\ldots,a_n),q),(p_1\cdot\ldots\cdot p_n\cdot p),\mathcal{F},\mathcal{L},\mathbf{C})$ is an n-ary operation in $\mathcal{D}$, $\Phi_1,\ldots,\Phi_n$ are solids in $\mathcal{D}$, Ψ is the solid resulting from the application of $\otimes$ to $\Phi_1,\ldots,\Phi_n$ via interfaces $\phi_1,\ldots,\phi_n$, and $(x_i,y_i,z_i,\mathbf{C}_i,\Psi_i)$ is a factoring of Φ_i for each i ($1 \le i \le n$), then $(x,y,z,\mathrm{X},\Delta)$ is a factoring of Ψ where x is $x_1\bullet\ldots\bullet x_n\cdot p$, y is $y_1\bullet\ldots\bullet y_n$, z is $z_1\bullet\ldots\bullet z_n\cdot q$, and

- $\mathrm{X} = \mathbf{C}_{p_1,\ldots,p_n}[\phi_1(x_1),\ldots,\phi_n(x_n)] \cup \mathbf{C}_1 \cup \ldots \cup \mathbf{C}_n$, and
- $\Delta(z) = \mathcal{F}((\Psi_1(z_1),\ldots,\Psi_n(z_n)),q)$.

Lemma 5.25: If in Lemma , $(x_i,y_i,z_i,\mathbf{C}_i,\Psi_i)$ is a perfect factoring of Φ_i for each i ($1 \le i \le n$), and $\mathbf{C}_{p_1,\ldots,p_n,p,q}[\phi_1(u_1),\ldots,\ \phi_n(u_n),u_{*|p|},d]$ implies that $((\Phi_1(u_1),\ldots,\Phi_n(u_n)),d) \in \text{scope}(\mathcal{F})$, then $(x,y,z,\mathrm{X},\Delta)$ is a perfect factoring of Ψ.

(O) **Definition 5.26:** If Φ be a solid in n variables and $\mathbf{N}$ is a set of interfaces to Φ, a subset P of its variables is said to be *sufficient for* Φ *with* $\mathbf{N}$ iff P includes all the variables required by the interfaces in $\mathbf{N}$, and for all y, $z \in \text{scope}(\Phi)$, if $y_i = z_i$ for all $i \in P$, then $\Phi(y) = \Phi(z)$. P is called *a parameter set for* Φ *with* $\mathbf{N}$ iff P is sufficient for Φ with $\mathbf{N}$, and no proper subset of P is sufficient for Φ with $\mathbf{N}$.

Lemma 5.28: Each over-constrained set of equalities has at least one well-constrained subset.

Lemma 5.29: If $\mathbf{C}$ is a minimally well-constrained set of equalities, then every proper subset of $\mathbf{C}$ is under-constrained.

(N) **Definition 5.30:** If $\mathbf{C}$ and $\mathbf{D}$ are sets of equalities, then $\mathbf{D}$ is said to be a *trimming* of $\mathbf{C}$ iff:

- $\mathbf{D}$=$\mathbf{C}$ where $\mathbf{C}$ has no well-constrained subset.

- **D** is a trimming of $\mathbf{C}_{\mathrm{var(X)}}$ where $\mathbf{X} \subseteq \mathbf{C}$ and **X** is minimally well-constrained.

Lemma 5.31: If **C** is a properly over-constrained set of equalities, then for each subset **V** of var(**C**), either $\mathbf{C}_{\mathbf{V}}=\varnothing$, or $\mathbf{C}_{\mathbf{V}}$ is properly over-constrained.

Lemma 5.32: If **C** is a minimally well-constrained set of equalities, then for each non-empty subset **V** of var(**C**), either $\mathbf{C}_{\mathbf{V}}=\varnothing$, or $\mathbf{C}_{\mathbf{V}}$ is properly over-constrained.

Lemma 5.33: If **C** is a set of equalities and **X** a properly over-constrained subset of **C**, then **D** is a trimming of **C** iff **D** is a trimming of $\mathbf{C}_{\mathrm{var(X)}}$.

Lemma 5.34: If **C** is a set of equalities and **D** is a trimming of **C**, then either **D**=**C**, or for every minimally well-constrained subset **X** of **C**, **D** is a trimming of $\mathbf{C}_{\mathrm{var(X)}}$.

Lemma 5.35: If **C** is a set of equalities, and **D** and **E** are trimmings of **C**, then **D**=**E**.

Lemma 5.36: If **C** is a set of equalities and $u \in \mathrm{var}(\mathbf{C})$, then u occurs in a well-constrained subset of **C** iff $u \notin \mathrm{var}(\perp\mathbf{C})$.

(N) **Definition 5.37:** If **C** is a set of equalities, then a subset s of var(⊥C) is said to be a *reduced set for C* iff

- $s=\varnothing$ and $\perp\mathrm{C}=\varnothing$, or
- $s\neq\varnothing$ and $\perp\mathrm{C}\neq\varnothing$ and for each $v \in s$, $s-\{v\}$ is a reduced set for $(\perp\mathrm{C})_v$.

Lemma 5.48: If **C** is a set of equalities, then s is a reduced set for **C** iff it is a reduced set for ⊥C.

Lemma 5.49: If **C** is a set of equalities and $z \in \mathrm{var}(\perp\mathrm{C})$, then $\perp((\perp\mathrm{C})_z)=\perp(\mathrm{C}_z)$.

Lemma 5.50: If **C** is a set of equalities, s a reduced set for **C**, and y a subset of s, then $s-y$ is a reduced set for $\mathbf{C}_y$.

(N) **Definition 5.51:** If **C** is a set of equalities and $v \subseteq \mathrm{var}(\mathbf{C})$, then v is said to be *strictly constrained in C* iff v=var(E) for some equality $\mathrm{E} \in \mathbf{C}$.

Lemma 5.52: If **C** is a set of equalities and s a reduced set for **C**, then no subset of s is strictly constrained in **C**.

Lemma 5.53: If **C** is a set of equalities and $v \in \mathrm{var}(\perp\mathbf{C})$, then $\mathrm{var}(\perp(\mathbf{C}_v)) \subseteq \mathrm{var}(\perp\mathbf{C})$.

Lemma 5.54: If **C** is a set of equalities, $v \in \mathrm{var}(\perp\mathbf{C})$, s a reduced set for $\mathbf{C}_v$, and $u \in s$, then $v \in \mathrm{var}(\perp(\mathbf{C}_u))$.

Lemma 5.55: If **C** is a set of equalities, $v \in \mathrm{var}(\perp\mathbf{C})$, and s is a reduced set for $\mathbf{C}_v$, then $s \cup \{v\}$ is a reduced set for **C**.

Lemma 5.56: If **C** is a set of equalities then **C** has a reduced set.

Lemma 5.57: If **C** is a set of equalities, then a variable in var(**C**) is in a reduced set for **C** iff it does not occur in any well-constrained subset of **C**.

Lemma 5.58: If **C** is a set of equalities with no over-constrained subset and $\perp\mathbf{C}=\emptyset$, then **C** is well-constrained.

Lemma 5.59: If **C** is a set of equalities, then $\perp\mathbf{C}$ has no over-constrained subset.

Lemma 5.60: All reduced sets for a set of equalities are the same size.

Corollary 5.61: If **C** is a set of equalities and s is a reduced set for **C**, then no subset of s except s can be a reduced set for **C**.

(N) **Definition 5.62:** If **C** is a set of constraints, not all of which are equalities, then s is said to be a *reduced set for C* iff $s = x \cup (\mathrm{var}(\mathbf{C}) - \mathrm{var}(\mathbf{C}^=))$ and x is a reduced set for $\mathbf{C}^=$.

(N) **Definition 5.63:** If **C** is a set of constraints and $v \subseteq \mathrm{var}(\mathbf{C})$, then s is said to be a *reduced set for C with respect to v* iff $s = v \cup u$ and u is a reduced set for $\mathbf{C}_v$.

Lemma 5.64: If **C** is a set of equalities and s is a reduced set for **C** with respect to v, then $s - v$ is a reduced set for $\mathbf{C}_v$.

(N) **Definition 5.65:** If Φ is a solid with a factoring $(x,y,z,\mathbf{C},\Psi)$, **N** is a set of interfaces exposed by Φ, and v is the set of variables required by the interfaces in **N**, then Φ *is said to be reduced with respect to* **C** *and N* iff x is a reduced set for **C** with respect to v.

Lemma 5.66: If Φ is a solid with factoring $(x,y,z,\mathbf{C},\Psi)$ exposing an interface ϕ, and Δ is a solid equivalent to Φ with a factoring $(x',y',z,\mathbf{C},\Psi)$ such that $x'\bullet y'$ is a permutation of $x\bullet y$ and the set of variables required by ϕ is a subset of x', then Δ exposes an interface δ equivalent to ϕ.

Lemma 5.67: If **C** is a set of equalities, $z \subseteq \mathrm{var}(\mathbf{C})$, $w \subseteq \mathrm{var}(\mathbf{C})-z$, and for every $x \in z$ there exists $\mathrm{E} \in \mathbf{C}$ such that $x \in \mathrm{var}(\mathrm{E})$, $x \notin \mathrm{var}(\mathrm{E}')$ for all $\mathrm{E}' \in \mathbf{C}-\{\mathrm{E}\}$, $|\mathrm{var}(\mathrm{E})|\leq 2$, and $\mathrm{var}(\mathrm{E})\cap z=\{x\}$, then there exists u, a reduced set for **C** with respect to w, such that $u \cap z = \emptyset$.

Lemma 5.68: If Φ is a solid in a design space $\mathcal{D}$ with factoring $(x,y,z,\mathbf{C},\Psi)$, and Φ exposes a set of interfaces **N**, there exists a solid Δ equivalent to Φ which exposes a set of interfaces **M** equivalent to those in **N**, and has a factoring $(u,v,t,\mathrm{X},\Omega)$ such that Δ is reduced with respect to X and **M**.

Lemma 5.69: If $(x,y,z,\mathbf{C},\Psi)$ is a factoring of a solid Φ, **C** contains only equalities, **N** is a set of interfaces exposed by Φ, and Φ is reduced with respect to **C** and **N**, then x is a parameter set for Φ with **N**.

Lemma 5.72: If **C** is a minimally well-constrained set of equalities and $|\mathbf{C}|>1$, then no variable in var(**C**) is a solitary variable in **C**.

Lemma 5.74: If **C** is a set of equalities, **C** has no well-constrained subset, and $v \in \mathrm{var}(\mathbf{C})$, then $\mathbf{C}_v$ has no over-constrained subset.

Lemma 5.75: If **C** and **D** are sets of equalities, and $\perp\mathbf{D}=\emptyset$, then $\perp(\mathbf{C}\cup\mathbf{D})=\perp(\mathbf{C}_{\mathrm{var}(\mathbf{D})})$.

Lemma 5.76: If **C** is a set of equalities, s is a reduced set for **C**, y is a subset of s, and **Z** is a subset of **C** such that $\#(\mathbf{Z})=0$ and y is a subset of var(**Z**), then $|\mathrm{var}(\mathbf{Z})|-|\mathbf{Z}|\geq|y|$.

Lemma 5.77: If **C** and **D** are sets of equalities, s is a reduced set for **C**, v is a reduced set for **D**, and $\mathrm{var}(\mathbf{C})\cap\mathrm{var}(\mathbf{D})\subseteq s$, then for each $x \in v$, $x \in \mathrm{var}(\perp(\mathbf{C}\cup\mathbf{D}))$.

Lemma 5.78: If **C** and **D** are sets of equalities, s is a reduced set for **C**, v is a reduced set for **D**, and var(**C**)$\cap$var(**D**) is a subset of s, then $v \cup (s-\text{var}(\mathbf{D}))$ is a reduced set for **C**$\cup$**D**.

Lemma 5.79: If for each i ($1 \le i \le n$) $\mathbf{C}_i$ is a set of equalities, var($\mathbf{C}_i$) and var($\mathbf{C}_j$) are disjoint for each $j \ne i$, s_i is a reduced set for $\mathbf{C}_i$, and **D** is a set of equalities such that for each i ($1 \le i \le n$) var($\mathbf{C}_i$)$\cap$var(**D**) is a subset of s_i, and v is a reduced set for **D**, then $v \cup (\left(\bigcup_{i=1}^{n} s_i\right) - \text{var}(\mathbf{D}))$ is a reduced set for $\mathbf{D} \cup \left(\bigcup_{i=1}^{n} \mathbf{C}_i\right)$.

Lemma 5.80: If for each i ($1 \le i \le n$) $\mathbf{C}_i$ is a set of equalities, var($\mathbf{C}_i$) and var($\mathbf{C}_j$) are disjoint for each $j \ne i$, s_i is a reduced set for $\mathbf{C}_i$ with respect to v_i, and **D** is a set of equalities such that for each i ($1 \le i \le n$) var($\mathbf{C}_i$)$\cap$var(**D**)$\subseteq s_i$, then $v \cup (\left(\bigcup_{i=1}^{n} s_i\right) - \text{var}(\mathbf{D}))$ is a reduced set for $\mathbf{D} \cup \left(\bigcup_{i=1}^{n} \mathbf{C}_i\right)$ with respect to $w = \left(\bigcup_{i=1}^{n} v_i\right)$, where v is a reduced set for $\mathbf{D}_w$.

(N) **Definition 5.81:** Let $\mathcal{D}$ be a design space, Φ be a solid in $\mathcal{D}$ in n variables and p be a variable of Φ. The *behaviour of* Φ *with respect to* p is the partial function ϕ_p: $\mathbf{R} \rightarrow (\mathbf{R}^{n-1} \rightarrow \mathcal{D})$ such that $\forall y \in \mathbf{R}$, $z \in \mathbf{R}^{n-1}$, $\phi_p(y)(z) = [\, y = x_p \,,\, z = \bar{x}_p \,]\Phi(x)$. Here $(\mathbf{R}^{n-1} \rightarrow \mathcal{D})$ denotes the set of all partial functions from $\mathbf{R}^{n-1}$ to $\mathcal{D}$. p is called the *controlling variable* of the behaviour.

(N) **Definition 5.88:** Let Φ be a solid in a design space $\mathcal{D}$. A *free set* for a solid Φ is any proper subset of a parameter set for Φ.

(N) **Definition 5.89:** Let $\mathcal{D}$ be a design space, and let Φ be a solid in $\mathcal{D}$ in n variables and $p_1,\ldots,p_k$ denoted by p be an arbitrary but fixed ordering of a free set for Φ. The *behaviour of* Φ *with respect to variables* $p_1,\ldots,p_k$ is the partial function ϕ_p: $\mathbf{R}^k \rightarrow (\mathbf{R}^{n-k} \rightarrow \mathcal{D})$ such that $\forall y \in \mathbf{R}^k$,

$z \in \mathbf{R}^{n-k}$, $\phi_p(y)(z) = [\, y = x_p\, ,\, z = x_{\bar{p}}\,]\ \Phi(x)$. Here $(\mathbf{R}^{n-k} \rightarrow \mathcal{D})$ denotes the set of all partial functions from $\mathbf{R}^{n-k}$ to $\mathcal{D}$. $p_1, \ldots, p_k$ are called the *controlling variables* of the behaviour.

(N) **Definition 5.90:** If ϕ_p is the behaviour of a solid Φ with respect to a free set p for Φ, then Φ is said to have $|p|$ *degrees of freedom* with respect to the behaviour ϕ_p.

(N) **Definition 5.93:** Let $\mathcal{D}$ be a design space, Φ a solid in $\mathcal{D}$, $p_1, \ldots, p_k$ denoted by p an arbitrary but fixed ordering of a free set for Φ, ϕ_p a multiple behaviour of Φ with respect to p, $[t_s, t_e]$ a closed interval in $\mathbf{R}$, and $\mathbf{f}$ a function from $\mathbf{R}^{2k+1}$ to $\mathbf{R}^k$ such that $\forall x, y \in \mathbf{R}^k$, $\mathbf{f}(x, y, t_s) = x$ and $\mathbf{f}(x, y, t_e) = y$. A *motion of* Φ *with respect to* ϕ_p *and* $\mathbf{f}$ *on the interval* $[t_s, t_e]$ is a partial function $\mathrm{M}: \mathbf{R}^k \times \mathbf{R}^k \rightarrow (\mathbf{R} \rightarrow (\mathbf{R}^{n-k} \rightarrow \mathcal{D}))$ such that $\forall (x, y) \in \mathbf{R}^k \times \mathbf{R}^k$, $\mathrm{M}(x, y)(t) = [\, t \in [t_s, t_e]\, ,\, z = \mathbf{f}(x, y, t)\,]\ \phi_p(z)$.

Appendix C: Algorithms

The implementations of the algorithms discussed in this thesis are presented here. All are expressed in the SICStus implementation of Prolog [126].

C.1 Interpreter Engine

```
:-use_module(library(lists)).
:-use_module(library(jasper)).

/*    "mergeall" performs all possible merges in a set of       */
/*    equalities.                                               */
/*    finds and reports sources of failures.                    */

mergeall(_,[],[],VarsIn,VarsIn):-!.
mergeall(JVM,EqsIn,EqsOut,VarsIn,VarsOut):-
          element(X,EqsIn,Rem),
          element(Y,Rem,Rest),
          same-root(X,Y),!,
          search_for_failures(EqsIn,[],Failures),!,
          report_failures(JVM,Failures),!,
          merge(JVM,X,Y,VarsIn,VarsTemp),
          delete_all(JVM,[X|Rest],EqDeleted,VarsTemp,VarsTemp2),
          mergeall(JVM,EqDeleted,EqsOut,VarsTemp2,VarsOut).
mergeall(_,EqsIn,EqsIn,VarsIn,VarsIn).
```

```
/*    "search_for_failures" finds sources of failure (if any)   */
/*    and reports back to Java.                                 */

search_for_failures(EqsIn,Sofar,Failures):-
           element(f(V1=Term1,Eq1),EqsIn,Rem),
           element(f(V2=Term2,Eq2),Rem,Rest),
           V1==V2,
           Term1 \= Term2,
           search_for_failures(Rest,
                                [pair(Eq1,Eq2)|Sofar],Failures).
search_for_failures(_,Failures,Failures).


/*    "element" picks an element from a list                    */

element(X,[X|Rem],Rem).
element(X,[Y|Rem],[Y|Rest]):-
                element(X,Rem,Rest).


/*    "same-root" checks that two equalities have the same root */

same-root(f(V1=_,_),f(V2=_,_)):- V1==V2.


/*    "merge" merges two equalities which have the same root.   */
/*    "merge" never fails.                                      */

merge(JVM,f(V=Term,Eq1),f(V=Term,Eq2),L1,L3):-!,
                debug(JVM),
                (ask_merge(JVM,Eq1,Eq2) |
                           ask_undo_merge(JVM,Eq1,Eq2),fail),
                compress(JVM,L1,L2),
                Term=..[_|Arguments],
                subtract_terminals([V|Arguments],L2,L3).


/*    "preprocess_variables" adds the new variables to the list */
/*    of variables and then compresses the list of variables    */

process_variables(L1,L2,L4):-
                append(L1,L2,L3),
                compress(L3,L4).
```

```
/*    "compress" compresses the list of variables              */

compress(JVM,[],[]).
compress(JVM,L1,L2):-
                element(f(V1,N1,Type1,Ids1),L1,Rest),
                element(f(V2,N2,Type2,Ids2),Rest,Rem),
                V1==V2,!,
                set_type(JVM,Type1,Type2,Type,Ids1,Ids2),
                append(Ids1,Ids2,Ids),
                N is N1+N2,
                compress(JVM,[f(V1,N,Type,Ids)|Rem],L2).
compress(JVM,L,L).


/*    set_type determines the type of a variable.              */


set_type(JVM,X,X,X,_,_):-!.
set_type(JVM,false,true,true,Ids1,_):-
                mark_undeletable_terminals(JVM,Ids1) |
                (mark_deletable_terminals(JVM,Ids1),!,fail).
set_type(JVM,true,false,true,_,Ids2):-
                mark_undeletable_terminals(JVM,Ids2) |
                (mark_deletable_terminals(JVM,Ids2),!,fail).


/*    "delete_all" deletes all deletable equalities from       */
/*    the list of equalities "In".                             */
/*    L1 and L2 are the list of variables before and after     */
/*    the deletions, respectively.                             */

delete_all(_,[],[],L,L):-!.
delete_all(JVM,EqIn,EqOut,VarIn,VarOut):-
           element(f(V2,1,false,_),VarIn,RestVar),
           element(f(V1=Term,Eq),EqIn,RestEq),
           V1==V2,!,
           Term=..[_|Arguments],
           subtract_terminals(Arguments,RestVar,Temp),!,
           debug(JVM),
           (ask_delete(JVM,Eq) | ask_undo_delete(JVM,Eq),fail),
           delete_all(JVM,RestEq,EqOut,Temp,VarOut).
delete_all(_,In,In,L,L).
```

```
/*   "subtract_terminals" decrements the number of          */
/*   occurrences of the variables in Arguments.             */
/*   "In" and "Out" are the list of variables before        */
/*   and after the decrement, respectively.                 */

subtract_terminals([],In,In).
subtract_terminals(Arguments,In,Out):-
                element(V1,Arguments,Rest),
                element(f(V2,N1),In,Rem),
                V1==V2,!,
                N is N1-1,
                subtract_terminals(Rest,[f(V1,N)|Rem],Out).
subtract_terminals(_,In,In).


/*   debug determines if the program should run in step     */
/*   forward or step backward                               */

debug(JVM):-
           ask_debug(JVM,Mode),get_direction(Direction),!,
                      up_down(Mode,Direction),recall_debug(JVM).


/*                                                          */

recall_debug(_).
recall_debug(JVM):-debug(JVM).


/*                                                          */

up_down(X,X):-!.
up_down(_,_):-fail.


/*   "element" picks an element from a list                 */

element(X,[X|Rem],Rem).
element(X,[Y|Rem],[Y|Rest]):-
                element(X,Rem,Rest).
```

```
/*                                                                  */

tell(JVM,Id,Cn,Term):-
          get_direction(Direction),
          jasper_call(JVM,method('lograph/engine/PrologToJava',
                    'mine',[static]),
                    mine(+integer,+integer,+boolean,[-term]),
                    mine(Id,Cn,Direction,Term)
          ).


/*                                                                  */

animate_replace(JVM,Id):-
          jasper_call(JVM,method('lograph/engine/PrologToJava',
                    'replace',[static]),
                    replace(+integer),replace(Id)
          ).

/*                                                                  */

tell_undo_replace(JVM,Id,Cn):-
          get_direction(Direction),
          jasper_call(JVM,method('lograph/engine/PrologToJava',
                    'undo_replace',[static]),
                    undo_replace(+integer,+integer,+boolean),
                    undo_replace(Id,Cn,Direction)
          ).


/*                                                                  */

ask_merge(JVM,Eq1,Eq2):-
          get_direction(Direction),
          jasper_call(JVM,method('lograph/engine/PrologToJava',
                    'merge',[static]),
                    merge(+integer,+integer,+boolean),
                    merge(Eq1,Eq2,Direction)
          ).
```

```
/*                                                                */

ask_undo_merge(JVM,Eq1,Eq2):-
           get_direction(Direction),
           jasper_call(JVM,method('lograph/engine/PrologToJava',
                      'undo_merge',[static]),
                      undo_merge(+integer,+integer,+boolean),
                      undo_merge(Eq1,Eq2,Direction)
           ).


/*                                                                */

ask_delete(JVM,Eq):-
           get_direction(Direction),
           jasper_call(JVM,method('lograph/engine/PrologToJava',
                      'delete',[static]),
                      delete(+integer,+boolean),
                      delete(Eq,Direction)
           ).


/*                                                                */

ask_undo_delete(JVM,Eq):-
           get_direction(Direction),
           jasper_call(JVM,method('lograph/engine/PrologToJava',
                      'undo_delete',[static]),
                      undo_delete(+integer,+boolean),
                      undo_delete(Eq,Direction)
           ).


/*                                                                */

ask_debug(JVM,Term):-
           jasper_call(JVM,method('lograph/engine/PrologToJava',
                      'debug',[static]),
                      debug([-integer]),
                      debug(Dir_abort)
           ),
           abort_query(Dir_abort,Term).
```

```
/*                                                           */

abort_query(0,_):-
               retract(get_direction(_)),abort.
abort_query(1,true):-!.
abort_query(2,false).


/*                                                           */

report_failures(_,[]).
report_failures(JVM,Failures):-
          debug(JVM),
          get_direction(Direction),
          toggle_direction(Direction),
          get_direction(Dir),
          jasper_call(JVM,method('lograph/engine/PrologToJava',
                    'failure',[static]),
                    failure(+term,+boolean),
                    failure(Failures,Dir)
          ),
          fail.


/*                                                           */

mark_undeletable_terminals(JVM,Ids):-
          jasper_call(JVM,method('lograph/engine/PrologToJava',
                    'make_unremovable',[static]),
                    make_unremovable(+term),
                    make_unremovable(Ids)
          ).


/*                                                           */

mark_deletable_terminals(JVM,Ids):-
          jasper_call(JVM,method('lograph/engine/PrologToJava',
                    'make_removable',[static]),
                    make_removable(+term),
                    make_removable(Ids)
          ).
```

```
/*                                                                  */

replace(Literal):-
           Literal =..[Name|Args],
           Args=[CaseNumber,Max|Rest],
           length(Args,Arity),
           current_predicate(Name/Arity),!,
           get_direction(Direction),
           bounds(Direction,CaseNumber,NewCaseNumber,Max),
           NewArgs = [NewCaseNumber,Max|Rest],
           NewLiteral =..[Name|NewArgs],
           replace2(NewLiteral).
replace(Literal):-
           Literal =..[Name|Args],
           Args=[_,_,JVM|Rest],
           last(Rest,Id),
           user_input(JVM,Id,Clause),!,
           (assert((Clause),Ref) | retract((Clause)),fail),
           clause(Head,_,Ref),
           Head =..[NewName|_],
           NewLiteral =..[NewName|Args],
           NewLiteral.


/*                                                                  */

send_back(JVM,Clause):-
           jasper_call(JVM,method('lograph/engine/PrologToJava',
                      'printClause',[static]),
                      printClause(+integer,[-boolean]),
                      printClause(Clause,_)
           ).


/*                                                                  */

send_clause(JVM,Clause):-
           jasper_call(JVM,method('lograph/engine/PrologToJava',
                      'print_Clause',[static]),
                      print_Clause(+chars,[-boolean]),
                      print_Clause(Clause,_)
           ).
```

```
/*                                                                  */

replace2(Literal):-
          Literal.
replace2(Literal):-
          Literal =..[Name|Args],
          Args=[CaseNumber,Max,JVM|Rest],
          get_direction(Direction),
          snap_out(Direction,CaseNumber,Max),
          ask_debug(JVM,Mode),
          compute(Mode,Direction,CaseNumber,NewCaseNumber),
          NewArgs = [NewCaseNumber,Max,JVM|Rest],
          NewLiteral =..[Name|NewArgs],!,
          replace2(NewLiteral).


/*                                                                  */

bounds(true,1,1,_).
bounds(false,_,Max,Max).


/*                                                                  */

snap_out(true,1,_):-
               !,fail.
snap_out(false,Max,Max):-
               !,fail.
snap_out(_,_,_).


/*                                                                  */

compute(X,X,CaseNumber,CaseNumber).
compute(true,false,CaseNumber,NewCaseNumber):-
                    get_direction(D),
                    toggle_direction(D),
                    NewCaseNumber is CaseNumber+1.
compute(false,true,CaseNumber,NewCaseNumber):-
                    get_direction(D),
                    toggle_direction(D),
                    NewCaseNumber is CaseNumber-1.
```

```
/*                                                              */

toggle_direction(true):-
                retract(get_direction(_)),
                assert(get_direction(false)).
toggle_direction(false):-
                retract(get_direction(_)),
                assert(get_direction(true)).


/*                                                              */

preamble(JVM):-
                get_direction(Direction),
                set_buttons(JVM,Direction),
                ask_debug(JVM,Mode),
                enable_button(JVM,Mode),!,
                recall_preamble(JVM).


/*                                                              */

set_buttons(JVM,true):-
                enable_do(JVM),
                disable_pause(JVM),
                disable_undo(JVM),
                no_more_solution(JVM,true),!.
set_buttons(JVM,false):-
                enable_undo(JVM),
                disable_pause(JVM),
                disable_do(JVM),!,
                no_more_solution(JVM,false).


/*                                                              */

enable_button(JVM,true):-
                enable_undo(JVM),!.
                enable_button(JVM,false):-
                enable_do(JVM).
```

```
/*                                                                  */
next_solution(JVM):-
           ask_debug(JVM,Mode),!,
           next(Mode).

/*                                                                  */

next(true):-
           toggle_direction(true),!,fail.
next(false):-
           fail.

/*                                                                  */

recall_preamble(_).
recall_preamble(JVM):-
           preamble(JVM).

/*                                                                  */

disable_undo(JVM):-
           jasper_call(JVM,method('lograph/engine/PrologToJava',
                     'disable_undo',[static]),
                     disable_undo([-boolean]),
                     disable_undo(Term)
           ).

/*                                                                  */

disable_do(JVM):-
           jasper_call(JVM,method('lograph/engine/PrologToJava',
                     'disable_do',[static]),
                     disable_do([-boolean]),
                     disable_do(Term)
           ).

/*                                                                  */

disable_pause(JVM):-
           jasper_call(JVM,method('lograph/engine/PrologToJava',
                     'disable_pauseButton',[static]),
                     disable_pauseButton([-boolean]),
                     disable_pauseButton(Term)
           ).
```

```
/*                                                                  */
enable_undo(JVM):-
           jasper_call(JVM,method('lograph/engine/PrologToJava',
                      'enable_undo',[static]),
                      enable_undo([-boolean]),
                      enable_undo(Term)
           ).

/*                                                                  */

enable_do(JVM):-
           jasper_call(JVM,method('lograph/engine/PrologToJava',
                      'enable_do',[static]),
                      enable_do([-boolean]),
                      enable_do(Term)
           ).


/*                                                                  */

disable_next(JVM):-
           jasper_call(JVM,method('lograph/engine/PrologToJava',
                      'disable_nextSolutionButton',[static]),
                      disable_nextSolutionButton([-boolean]),
                      disable_nextSolutionButton(Term)
           ).

/*                                                                  */

disable_previous(JVM):-
           jasper_call(JVM,method('lograph/engine/PrologToJava',
                      'disable_previousSolutionButton',[static]),
                      disable_previousSolutionButton([-boolean]),
                      disable_previousSolutionButton(Term)
           ).

/*                                                                  */

no_more_solution(JVM,Direction):-
           jasper_call(JVM,method('lograph/engine/PrologToJava',
                      'no_more',[static]),
                      no_more(+boolean,[-boolean]),
                      no_more(Direction,Term)
           ).
```

```
/*                                                                */
report_success(JVM,Direction):-
          get_direction(Direction),
          jasper_call(JVM,method('lograph/engine/PrologToJava',
                    'success',[static]),
                    success(+boolean,[-boolean]),
                    success(Direction,Term)
          ).

/*                                                                */

user_input(JVM,Id,Clause):-
          jasper_call(JVM,method('lograph/engine/PrologToJava',
                    'get_input',[static]),
                    get_input(+integer,[-chars]),
                    get_input(Id,Term)
          ),
          read_from_chars(Term,Clause).

/*                                                                */

cross_over(JVM,true,true):-
          toggle_direction(true),
          disable_next(JVM),
          enable_do(JVM),!.
cross_over(JVM,true,false):-
          disable_next(JVM),
          enable_do(JVM),!,fail.
cross_over(JVM,false,true):-
          disable_previous(JVM),
          enable_undo(JVM),!,fail.
cross_over(JVM,false,false):-
          toggle_direction(false),
          disable_previous(JVM),
          enable_undo(JVM),!.

/*                                                                */

postscript(JVM):-
          get_direction(Direction),
          report_success(JVM,Direction),
          ask_debug(JVM,Mode),!,
          cross_over(JVM,Direction,Mode),
          postscript(JVM).
```

C.2 Reduction

In this section, we present the implementation of reduction and optimised reduction algorithms discussed in Chapter 5. In Section C.2.3, we present the code used by both reduction and optimised reduction algorithms.

C.2.1 Basic Reduction Algorithm

```
/*                                                                  */
/*                                                                  */

:-consult('reduce_support.txt').


/*                                                                  */

reduce(VarC,C,[]):-
                trim(VarC,_,C,[]),!.
reduce(VarC,C,[V|Reduced]):-
                trim(VarC,VarTrimmedC,C,TrimmedC),
                next(V,VarTrimmedC,Rest),
                removeOccurGraph(V,TrimmedC,TrimmedCsubV),
                compress(Rest,VarPrunedC,TrimmedCsubV,PrunedC),
                reduce(VarPrunedC,PrunedC,Reduced).


/*   Since Subset generates subsets of a set from smaller      */
/*   to larger, this predicate in fact checkes if a set is     */
/*   minimally well-constrained.                               */

trim(VarC,VarTrimmedC,C,TrimmedC):-
                subset(X,C),
                \+length(X,0),
                connected(X),
                wellConstrained(X,VarX),
                removeAll(VarX,C,CsubVarX),
                removeEmpties(CsubVarX,PrunedC),
                subtract(VarC,VarX,VarPrunedC),
                trim(VarPrunedC,VarTrimmedC,PrunedC,TrimmedC),!.
trim(VarC,VarC,C,C).
```

```
/*                                                                   */

reduceInterface(X,Y,Required,C,XReduced,YExpanded):-
                removeRequired(Required,C,VarPrunedC,PrunedC),
                reduce(VarPrunedC,PrunedC,Reduced),
                reArrange(X,Y,Required,C,Reduced,
                                           XReduced,YExpanded).


/*    This prdicate reports all reduced sets for a set of         */
/*    equalities through forced backtracking                      */

allSets(VarC,C):-
                reduce(VarC,C,Reduced),
                print(Reduced),fail.
allSets(_).
```

C.2.2 Optimised Reduction Algorithm

```
/*                                                                   */
/*                                                                   */

:-consult(reduce_support.txt').

/*                                                                   */

reduce(VarC,C,Original,[V|Reduced]):-
                next(V,VarC,Rest),
                removeOccurGraph(V,C,SimplifiedC),
                compress(VarC,VarPrunedC,SimplifiedC,PrunedC),
                intersect(Rest,VarPrunedC,Candidates),
                reduce(Candidates,PrunedC,Original,Reduced).
reduce(_,C,Original,[]):-
                notOverConstrained(C,Original).


/*                                                                   */

notOverConstrained([Eq|Rest],Original):-
                getGraph(Eq,Rest,C,Others),!,
                isValid([Eq|C],Original),
                notOverConstrained(Others,Original),!.
notOverConstrained([],_).
```

```
/*                                                               */

isValid(C,Original):-
                originalOverConstrainedSubSet(C,Original,Subset),
                simplify(Subset,C,Rest),!,
                isValid(Rest,Original).
isValid(C,_):-
                wellConstrained(C,_).
/*                                                               */

originalOverConstrainedSubSet(C,Original,Subset):-
                subset(Subset,C),
                \+length(Subset,0),
                subset(Subset,Original),
                getVariables(Subset,V),
                length(Subset,M),
                length(V,N),
                M > N.


/*                                                               */

simplify(Subset,C,Rest):-
                getVariables(Subset,VarSubset),
                removeAll(VarSubset,C,Temp),
                removeEmpties(Temp,Rest).


/*                                                               */

reduceInterface(X,Y,Required,C,XReduced,YExpanded):-
                removeRequired(Required,C,VarPrunedC,PrunedC),
                reduce(VarPrunedC,PrunedC,PrunedC,Reduced),
                reArrange(X,Y,Required,C,Reduced,XReduced,
                                                YExpanded).


/*    This prdicate is used to report all reduced set
      for a set of equalities through forced backtracking       */

allSets(VarC,C):-
                reduce(VarC,C,C,Reduced),
                print(Reduced),fail.
allSets(_).
```

C.2.3 Reduction Algorithm Support

```
/*    filename: 'reduce_support.txt'                          */
/*    This file includes predicates common to reduction       */
/*    and optimised reduction algorithms.                     */


:-use_module(library(lists)).

/*                                                            */

wellConstrained(X,VarX):-
                getVariables(X,VarX),
                length(VarX,N),
                length(X,N).


/*                                                            */

removeOccurGraph(Var,C,[Rem|Rest]):-
                element(E,C,Others),
                element(Var,E,Rem),
                removeOccurGraph(Var,Others,Rest),!.
removeOccurGraph(_,C,C).


/*                                                            */

compress(_,_,C,_):-
                element([],C,_),!,fail.
compress(VarC,VarPrunedC,C,PrunedC):-
                element([V],C,Rest),!,
                removeOccurGraph(V,Rest,Rem),
                remove(V,VarC,Temp),
                compress(Temp,VarPrunedC,Rem,PrunedC),!.
compress(VarC,VarC,C,C).

/*                                                            */

connected(Set):-
                element(E,Set,_),
                getGraph(E,Set,C,_),
                length(C,N),
                length(Set,N).
```

```
/*                                                    */
getGraph(E,C,[Eq|Graph],Others):-
                element(V,E,_),
                element(Eq,C,Rest),
                element(V,Eq,_),
                append(E,Eq,Temp),
                remove_duplicates(Temp,NewVars),
                getGraph(NewVars,Rest,Graph,Others),!.
getGraph(_,C,[],C).

/*                                                    */

remove(X,L,LminusX):-
                element(X,L,LminusX),!.
remove(_,L,L).


/*                                                    */

next(X,[X|Rem],Rem).
next(X,[_|Rem],Rest):-
                next(X,Rem,Rest).

/*                                                    */

subset([],_).
subset([X|Rest],Set):-
                next(X,Set,Rem),
                subset(Rest,Rem).

/*                                                    */

removeAll(VarX,C,SimplifiedC):-
                element(V,VarX,RestVarX),
                removeOccurGraph(V,C,Inter),
                removeAll(RestVarX,Inter,SimplifiedC),!.
removeAll([],C,C).

/*                                                    */
getVariables([],[]).
getVariables([Eq|Rest],VarC):-
                getVariables(Rest,Temp),
                append(Eq,Temp,Vars),
                remove_duplicates(Vars,VarC).
```

```
/*                                                                  */

removeEmpties(C,PrunedC):-
                element([],C,Rest),
                removeEmpties(Rest,PrunedC),!.
removeEmpties(C,C).


/*    "element" picks an element from a list                        */

element(X,[X|Rem],Rem).
element(X,[Y|Rem],[Y|Rest]):-
                element(X,Rem,Rest).


/*                                                                  */

subtract(L1,L2,L1minusL2):-
                element(X,L1,Rest),
                element(X,L2,Rem),!,
                subtract(Rest,Rem,L1minusL2).
subtract(L1,_,L1).


/*                                                                  */

intersect(L1,L2,[X|R]):-
                element(X,L1,Rem),
                element(X,L2,Rest),
                intersect(Rem,Rest,R),!.
intersect(_,_,[]).


/*                                                                  */

removeRequired(Required,C,VarPrunedC,PrunedC):-
                getVariables(C,VarC),
                intersect(VarC,Required,R),
                removeAll(R,C,Inter),
                removeEmpties(Inter,Eqs),
                getVariables(Eqs,VTemp),
                compressNoFail(VTemp,VarPrunedC,Eqs,ETemp),
                removeEmpties(ETemp,PrunedC).
```

```
/*                                                                  */

compressNoFail(VarC,VarPrunedC,C,PrunedC):-
                element([V],C,Rest),!,
                removeOccurGraph(V,Rest,Rem),
                remove(V,VarC,Temp),
                compressNoFail(Temp,VarPrunedC,Rem,PrunedC),!.
compressNoFail(VarC,VarC,C,C).


/*                                                                  */

reArrange(X,Y,Required,C,Reduced,XReduced,YExpanded):-
                getVariables(C,VarC),
                append(Required,Reduced,Temp),
                subtract(VarC,Temp,Removed),
                append(X,Required,XTemp1),
                append(XTemp1,Reduced,XDup),
                remove_duplicates(XDup,XTemp2),
                subtract(XTemp2,Removed,XReduced),
                append(X,Y,Combined),
                subtract(Combined,XReduced,YExpanded).
```

C.3 Translator

The translator algorithm reads an anchor-knot graph and eventually collapses it into a single anchor, while at the same time translating it in to a corresponding PLaSM program. The data structure that represents an anchor-knot graph consists of a list of anchors and a list of knots. The output of the translator is a list of function definitions. An anchor is a term anchor(*name*,*terminals*) where *name* corresponds to a function definition in PLaSM and *terminals* is the list of anchor's terminals. A knot is a term knot(*operation,operands,newpars,function*) where *operation* is the name of the operation associated with the knot, *operands* is the list of operation's operands, *newpars* is a list of new parameters, and *function* is a function in PLaSM.

```
/*   Translates an anchor-knot graph to a PLaSM prgram.        */


:-use_module(library(lists)).


/*                                                             */

translator(_,_,[],_,DefIn,DefIn).
translator(AnchorsIn,AnchorsOut,Knots,N,DefIn,DefOut):-
                apply(AnchorsIn,AnchorsTemp,Knots,KnotsOut,
                           N,DefIn,DefInter),
                M is N+1,
                translator(AnchorsTemp,AnchorsOut,KnotsOut,M,
                           DefInter,DefOut).

/*                                                             */

apply(AnchorsIn,[Anchor|AnchorsIn],[Knot|Rest],KnotsOut,N,
                           DefIn,DefOut):-
                Knot =.. [knot,_,Operands,NewPars,Function],
                buildAnchor(Operands,NewPars,N,Anchor),
                addDef(DefIn,Anchor,NewPars,Function,
                           Operands,DefOut),
                reconstruct(Rest,Operands,Anchor,KnotsOut).

/*                                                             */


buildAnchor(Operands,NewPars,N,anchor(Name,Terminals)):-
                name(N,Temp),
                append("solid",Temp,Name),
                buildParameters(Operands,[],OperandPars),
                append(OperandPars,NewPars,Terminals).


/*                                                             */

buildParameters([],Pars,Pars).
buildParameters([Operand|Rest],Pars,Terminals):-
                Operand =.. [anchor,_,Vars],
                append(Pars,Vars,Temp),
                remove_duplicates(Temp,SoFar),
                buildParameters(Rest,SoFar,Terminals).
```

```
/*                                                                  */

addDef(DefIn,Anchor,NewPars,Function,Operands,DefOut):-
                Anchor =.. [anchor,Name,Pars],
                append(DefIn,[def(Name,Pars,NewPars,
                           Function,Operands)],DefOut).


/*                                                                  */

reconstruct([],_,_,[]).
reconstruct([Knot|Rest],RemovedOperands,Anchor,[NewKnot|Rem]):-
                Knot =.. [knot,Operation,Operands,NewPars,
                           Function],
                replaceOperands(RemovedOperands,Operands,
                           Anchor,NewOperands),
                NewKnot =.. [knot,Operation,NewOperands,
                           NewPars,Function],
                reconstruct(Rest,RemovedOperands,Anchor,Rem).

/*                                                                  */

replaceOperands([],Operands,_,Operands).
replaceOperands([X|Rem],Operands,Anchor,NewOperands):-
                replace(X,Operands,Anchor,SoFar),
                replaceOperands(Rem,SoFar,Anchor,NewOperands).
/*                                                                  */

replace(_,[],_,[]).
replace(X,[Y|Rem],Anchor,[Anchor|Rem]):-
                                X==Y,!.
replace(X,[Y|Rem],Anchor,[Y|Rest]):-
                                replace(X,Rem,Anchor,Rest).


/*    "element" picks an element from a list                        */

element(X,[X|Rem],Rem).
element(X,[Y|Rem],[Y|Rest]):-
                element(X,Rem,Rest).
```

```
/*   The remainig predicate are to support printing           */
/*   the translated code in PLaSM.                            */

printDefs(Defs):-
          open('program.psm',write,Stream),
          plasm(Stream,Defs),
          close(Stream).



plasm(_,[]).
plasm(Stream,[Def|Rest]):-
          printDef(Stream,Def),
          plasm(Stream,Rest).


printDef(Stream,Def):-
          Def=.. [def,Name,Vars,ExtraVars,Function,Operands],
          format(Stream,'~2nDEF ~s(',[Name]),
          printPars(Stream,Vars),
          format(Stream,'::IsRealPos) = product ~n',[]),
          format(Stream,'WHERE ~n',[]),
          printOperands(Stream,Operands,1),
          format(Stream,'     product = ~s:<',[Function]),
          printOperandsList(Stream,Operands,1),
          printNewPars(Stream,ExtraVars),
          format(Stream,'> ~n',[]),
          format(Stream,'END~2n',[]).
printPars(_,[]).
printPars(Stream,[Par]):-
          format(Stream,'~p',[Par]).
printPars(Stream,[Par|Rest]):-
          format(Stream,'~p,',[Par]),
          printPars(Stream,Rest).


printOperands(_,[],_).
printOperands(Stream,[Operand|Rest],N):-
          Operand=..[anchor,Name,Vars],
          format(Stream,'     op~d=~s:<',[N,Name]),
          printList(Stream,Vars),
          format(Stream,'>, ~n',[]),
          M is N+1,
          printOperands(Stream,Rest,M).
```

```
printOperandsList(_,[],_).
printOperandsList(Stream,[_],N):-
                format(Stream,'op~d',[N]).
printOperandsList(Stream,[_|Rest],N):-
                format(Stream,'op~d,',[N]),
                M is N+1,
                printOperandsList(Stream,Rest,M).


printList(_,[]).
printList(Stream,[Var]):-
                format(Stream,'~p',[Var]).
printList(Stream,[Var|Rest]):-
                format(Stream,'~p,',[Var]),
                printList(Stream,Rest).


printNewPars(_,[]).
printNewPars(Stream,[Var|Rest]):-
                format(Stream,',~p',[Var]),
                printNewPars(Stream,Rest).
```