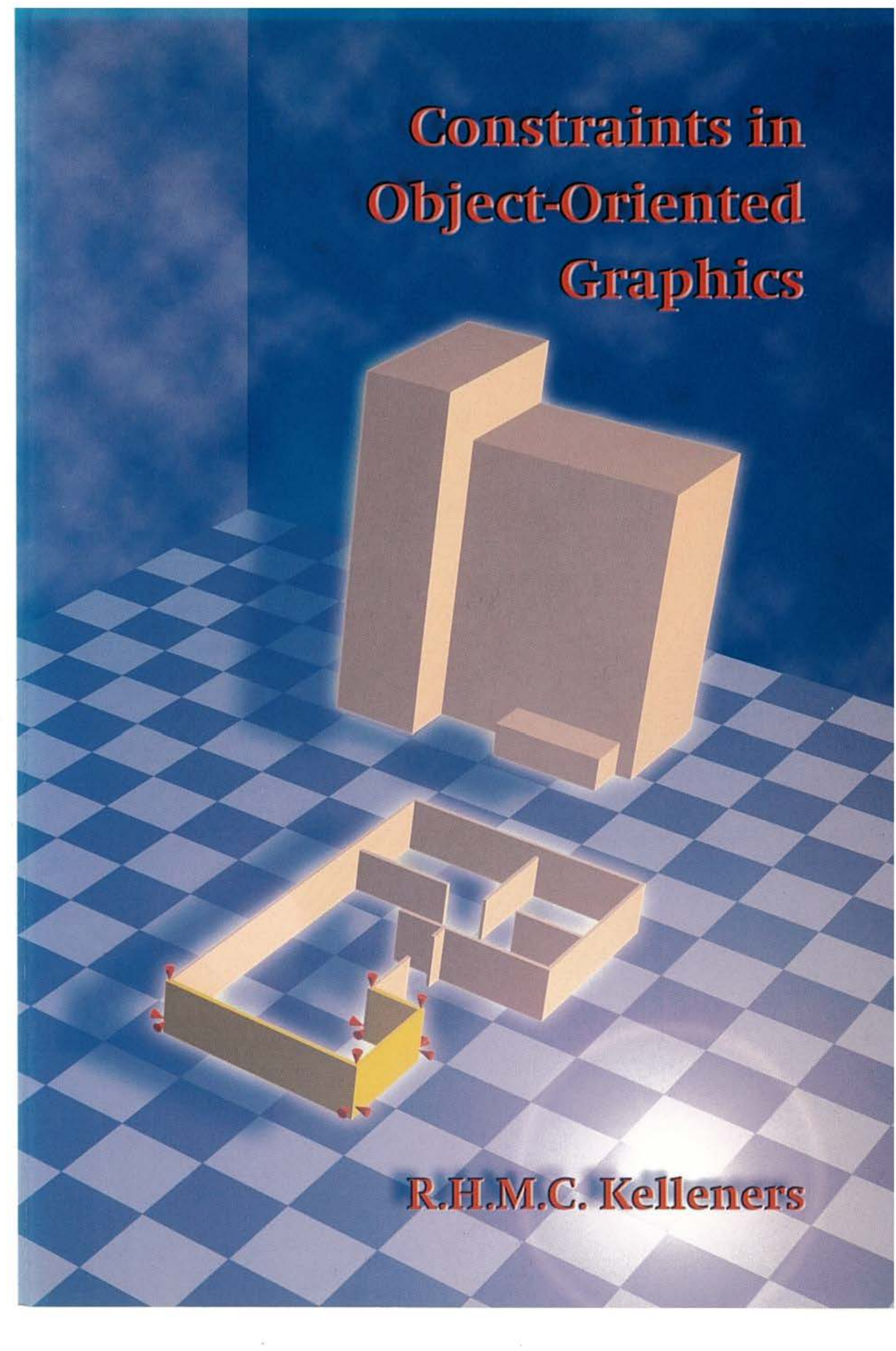

Constraints in
Object-Oriented
Graphics
R.H.M.C. Kelleners

STELLINGEN

behorende bij het proefschrift

Constraints in Object-Oriented Graphics

RICHARD KELLENERS

1. Wanneer een constraint solver in een object-georiënteerde omgeving gelijk wordt gesteld aan alle andere objecten en alleen kan communiceren via *message passing*, dan zal als gevolg van het *information hiding* principe, zijn taak worden bemoeilijkt en kan in veel gevallen geldigheid van de constraints niet worden gegarandeerd [1].

   [1] Hoofdstuk 3, dit proefschrift.

2. *Events* en *data flows* vormen een zeer geschikt protocol om een object-georiënteerd systeem te laten communiceren met een "extern" systeem. Dit komt voort uit het feit dat het protocol niet interfereert met de message passing activiteit van objecten binnen het OO-systeem [2].

   [2] Hoofdstuk 4, dit proefschrift.

3. Door een constraint solver te integreren met een object-georiënteerde programmeer*taal* [3], wordt het aantal typen constraints dat kan worden opgelost binnen die taal beperkt tot de typen die de solver ondersteund. Hierdoor zullen veel ontwerpen die objecten en constraints combineren niet in een dergelijke taal kunnen worden geïmplementeerd of een aanpassing van de taal tot gevolg hebben.

   [3] Gustavo Lopez. *The design and implementation of Kaleidoscope, a constraint imperative programming language*. Ph.D. thesis, University of Washington, Department of Computer Science & Engineering, 1997.

4. *Directionele* constraints en constraints die functionaliteit bezitten voor het berekenen van een lokale oplossing zijn zeer bruikbaar voor het modelleren van bepaalde constraint problemen [4]. Desalniettemin tasten dit soort voorzieningen de zuiverheid van het constraint paradigma aan waarin constraint specificatie en constraint solving van elkaar gescheiden zijn. Eerder genoemde voorzieningen kunnen slechts gezien worden als speciale gevallen van het algemene constraint paradigma.

   [4] Michael Sannella. *Constraint Satisfaction and Debugging for Interactive User Interfaces*. Ph.D. thesis, University of Washington, Department of Computer Science & Engineering, 1994.

5. Het gemak waarmee ontwerpers van object-georiënteerde software interne data van objecten publiekelijk toegankelijk maken is zorgwekkend. De oorzaak kan worden gevonden in het feit dat opleidingen tot OO-ontwerper niet genoeg de kracht en het belang van information hiding benadrukken.

6. De wereldwijde onkunde ten aanzien van het ontwikkelen van software enerzijds en de stijgende behoefte aan betrouwbare, modulaire software componenten anderzijds zal uiteindelijk leiden tot software modules die net zo standaard en toepasbaar zijn als gloeilampen en schroevendraaiers.

7. De ideale ontwerpmethode ten aanzien van software systemen zal de juiste balans moeten vinden tussen methoden en technieken als hulpmiddel om de complexiteit van het ontwerp te beheersen en heuristieken om de creativiteit ruim baan te geven.

8. Een Vedisch spreekwoord zegt: "Al het goede komt slechts langzaam tot ontwikkeling". Een oorzaak voor de huidige erbarmelijke toestand van veel software systemen kan worden gevonden in de hoge vlucht die de technologie in de laatste decennia heeft genomen waardoor het vakgebied van de software ontwikkeling zich nooit "langzaam" heeft kunnen ontwikkelen.

9. De kracht van de unix tekst editor 'vi' bestaat uit het feit dat navigeren door -en bewerken van- tekst snel en efficiënt kan worden aangestuurd via het toetsenbord. Op dit gebied is de editor veruit superieur aan "wysiwyg" tekstverwerkers als Word, Framemaker en WordPerfect en bovendien minder rsi-gevoelig omdat de muis overbodig is.

10. Het feit dat de dogma's van bepaalde religies vaak sterker benadrukt worden dan de leerstelling van onvoorwaardelijke verdraagzaamheid en liefde (die in elke grote religie aanwezig is), is er de oorzaak van dat die religies gebruikt worden voor het kweken van haat en als aanleiding voor oorlogvoering.

11. Als voor het einde van dit jaar enkele belangrijke problemen ten aanzien van internationale conflicten niet zijn opgelost, dan zal het mondiale Jaar-2000 probleem niet te maken hebben met vliegtuigen die "spontaan" (d.w.z. als gevolg van de millenium bug) uit de lucht vallen, maar met vliegtuigen die doelbewust (d.w.z. als gevolg van een "escalatie") uit de lucht worden geschoten.

12. Het verblijf op de aarde zou veel aangenamer zijn, wanneer elk van haar bewoners de wapenen zou opnemen tegen zichzelf in plaats van tegen elkaar.

CONSTRAINTS IN OBJECT-ORIENTED GRAPHICS

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Kelleners, Richard Hendrikus Maria Christina

Constraints in object-oriented graphics / by Richard Hendrikus Maria Christina Kelleners. -
Eindhoven : Eindhoven University of Technology, 1999.
Proefschrift. - ISBN 90-386-0691-5
NUGI 855
Subject headings: object-oriented programming / constraint programming / computer graphics /
computer aided architectural design
CR Subject Classification (1998): D.1.5, D.3.3, D.2.10, I.3, J.5

Printed by University Press Facilities, Eindhoven
Cover design by Ben Mobach
Cover illustration by Richard Kelleners

©1999 Richard H.M.C. Kelleners, Eindhoven.

CONSTRAINTS IN OBJECT-ORIENTED GRAPHICS

PROEFONTWERP

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR AAN DE TECHNISCHE UNIVERSITEIT EINDHOVEN, OP GEZAG VAN DE RECTOR MAGNIFICUS, PROF.DR. M. REM, VOOR EEN COMMISSIE AANGEWEZEN DOOR HET COLLEGE VOOR PROMOTIES IN HET OPENBAAR TE VERDEDIGEN OP DONDERDAG 17 JUNI 1999 OM 16.00 UUR

DOOR

RICHARD HENDRIKUS MARIA CHRISTINA KELLENERS

GEBOREN TE EINIGHAUSEN

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.dipl.ing. D.K. Hammer
en
prof.dr. E.H.L. Aarts

Copromotor:
dr. R.C. Veltkamp

Preface

In every scientific community that is related to an engineering discipline, the craft of designing an artifact is a main topic of research. The research effort is aimed at improving the design process or developing tools or methods to support this process. In computer science, the artifacts that are developed are software systems and tools to improve the design process include descriptions of software engineering life-cycles, formal or informal design methods, programming languages and techniques, and software environments for programming or testing purposes.

The development of software systems is still a very young discipline and tools and methodologies that support the design process are in the early stages. Often, there is also a gap between research that is done in computer science and the engineering discipline that is executed in business life. Ph.D. studies often result in theoretical theses that are not (yet) applicable to industry applications. In order to diminish this gap, Ph.D. research should not only focus on theoretical aspects of computer science, but also on the process of designing and building software.

This thesis is written as a designer's Ph.D. thesis (in Dutch: *proefontwerp*). A designer's Ph.D. is a post-graduate doctorate program in which special attention is paid to the development and the management of the design process of an artifact. It is a well-known program in research areas such as Architecture or Electrical Engineering. Here, a designer's Ph.D. is defined as follows.

> A technical artifact which is designed during a Ph.D. project, which is taken as subject for research or is used as an aid or as a tool in the research, and which is described in a Ph.D. dissertation.

Within computer science, getting a Ph.D. degree based on a design has a shorter history. At the Eindhoven University of Technology, only two (almost three as this is being written) Ph.D. students have received a promotion based on a designer's Ph.D. (see [Hautus, 1997] and [Argante, 1998]).

The difference between a traditional Ph.D. and a designer's Ph.D. is that in the former research is done which is relevant to the scientific community. In the latter, the activities are connected to the design process itself. A designer's Ph.D. also usually deals with *customers* who use the artifact or have direct benefits from the research.

Typical goals in a designer's Ph.D. are (1) to prove one's ability in developing a design and managing the design process, and (2) to gain insight into the design processes of the engineering disciplines involved. Concerning the first goal, the important aspect is to identify the process that leads to the artifact. This encompasses describing the different stages in the design process, explaining design decisions, and accounting for time planning.

Regarding the second goal, maybe the most interesting question that should be gained insight into is: What is *to design*? Often to design is considered as applying some methodology to build a certain artifact. However, this definition does not incorporate the creativity and experience that is needed by a designer. Moreover, one can discuss if designing involves the creation of the actual artifact or only describing how the artifact should be built. For example in Architecture, it is usually assumed that

when a house is being built by building workers, the actual design process that was started by the architect has ended. One could make a boundary line between the design stage that is done by an architect and the realization phase, which is executed by building workers.

In computer science, the boundary line between design and realization is harder to make. First of all, the "designer" and "worker" are often one and the same person or group of persons. Furthermore, during the process, the design of the system and the coding of it often go hand in hand and iterations between different activities frequently occur.

In this thesis, we take the viewpoint that *to design* incorporates all activities that contribute to the realization of an artifact. Examples of activities include the creative thought process, writing ideas down in a structured or non-structured way, and the realization of the artifact. In these activities, different stages can be distinguished, although there are no strict boundaries between the stages, neither is the order in which they are executed rigidly determined (that is, they may be carried out iteratively or in parallel). Consequently, we use the following definition to describe *to design*.

To design is to carry out the process comprising the creation of an artifact.

Acknowledgments

Having reached the end of my Ph.D. project, I wish to express my gratitude towards the people that contributed in one way or another to the completion of the whole project.

First of all, I thank my co-promotor Remco Veltkamp, without whom I would not have been able to start, continue, or finish this Ph.D. project. His constant involvement and drive have been a main motivating force for completing this work. Secondly, I would like to thank Kees van Overveld for inspiring discussions and careful reading of the manuscript. I thank Edwin Blake, Dieter Hammer, and Emile Aarts for their comments on the manuscript and for allowing me to carry out this Ph.D.

Special thanks go to the **MANIFOLD** group at CWI and to the members of the Computer Graphics group and the VR-DIS group at the TUE. I very much appreciate the pleasant atmosphere in which we have always cooperated. I would like to thank Kees Huizing for his input in Chapter 3, Jozef Hooman for checking the model in Chapter 5, and Wieger Wesselink for help on mathematical issues. Thanks also to Sander Rorije for his diligent work on implementing the SCAFFOLD system for the GDP.

Unmeasurable gratitude goes to my friends and colleagues at CWI and at the Technical Applications section at the TUE who provided me with the indispensable diversion and motivating support. In particular, I want to mention here Maurice Venbruex, Tanja Letanoux-van Rij, Paul van Gorp, and Elsa Gelis Escala. An additional word of thanks goes to Paul for his invaluable help in the final stretch of completing the thesis.

Es lêtste, wul ich hie in mien eige taal mien auwers bedanke veur alles wat ze veur mich höbbe gedaon. Ich zal noe weer get dêkker nao hoes kômme!

Contents

Preface v

1 Introduction 1

2 Background 5
- 2.1 Object-Oriented Programming 5
  - 2.1.1 Basic Concepts 6
  - 2.1.2 Object-Oriented Modeling 7
  - 2.1.3 Languages 9
- 2.2 Constraint Satisfaction Techniques 9
  - 2.2.1 Constraint Programming 9
  - 2.2.2 Constraint Solving 12
  - 2.2.3 Tailored Constraint Approaches 16
- 2.3 Conclusion 21

3 Combining Objects and Constraints 23
- 3.1 Problem Demarcation 24
- 3.2 Existing Object-Constraint Models 24
- 3.3 Imperative and Declarative Programming 26
- 3.4 Constraints vs. Information Hiding 27
  - 3.4.1 Case Study 28
- 3.5 Complexity of Constraint Algorithms 31
  - 3.5.1 Backtracking 33
  - 3.5.2 Node and Arc Consistency 33
  - 3.5.3 Local Propagation 34
  - 3.5.4 Relaxation 34
  - 3.5.5 Equate and OOCS 34
  - 3.5.6 Discussion 35
- 3.6 Towards a Solution 37
- 3.7 Conclusion 38

4 Conceptual Model 41
- 4.1 Requirements 41
- 4.2 Overview of a Conceptual Model 42
- 4.3 Entities, Events and Data Flows 44
  - 4.3.1 Entities 44
  - 4.3.2 Events 44

4.3.3 Data Flows 45
4.4 Constrainable 45
4.5 Constraint 46
4.6 Solver 46
4.7 Constraint Manager 47
4.8 Operational Description 47
4.9 Conclusion 50

5 Specification **53**
5.1 Data Flows and Events 53
5.1.1 Constrainable 54
5.1.2 Constraint 55
5.1.3 Solver 56
5.1.4 Constraint Manager 57
5.2 Mode Diagrams 58
5.2.1 Constrainables 60
5.2.2 Constraints 60
5.2.3 Solvers 62
5.2.4 Constraint Managers 63
5.3 Operational Description 65
5.4 Correctness 67
5.5 Conclusion 68

6 Prototype Implementation **69**
6.1 Overview of System Components 69
6.1.1 Manifold 70
6.2 Structure of PROTOM 72
6.2.1 Kernel 72
6.2.2 OO-Application API (OO-API) 76
6.2.3 Constraint Handler API (CH-API) 80
6.3 Constraint Handling Engine 80
6.3.1 Local Propagation Manager 80
6.3.2 Cooperating Managers 82
6.4 Object-Oriented Application Drawtool 86
6.5 Conclusion 87

7 Application Implementation **89**
7.1 Project Backgrounds 89
7.1.1 VR-DIS Project 89
7.1.2 GDP Animation System 90
7.2 Constraint System Top Level Design 91
7.3 SCAFFOLD System 93
7.3.1 Entities 93
7.3.2 Events 94
7.3.3 Data Flows 96
7.4 VD-CHE Constraint Handling Engine 97
7.4.1 Object and Constraint Types 97

7.4.2 Constraint Management and Solving 100
7.5 OO-Applications 107
7.5.1 VR-DIS Behavior Prototype 107
7.5.2 Looks 109
7.5.3 Evaluation by Users 111
7.6 Conclusion 113

8 Conclusions **115**
8.1 Design Process 115
8.2 Conclusions 118
8.3 Further Work 119

A Abbreviations **121**

Bibliography **123**

Index **131**

Summary **135**

Samenvatting **137**

Curriculum Vitae **139**

Chapter 1

Introduction

In modern day software development of computer graphics systems, two main techniques can be distinguished. The first is the *object-oriented* paradigm, the second is the *constraint* paradigm. Both are powerful techniques that allow designers to deal with the complexity of large software systems. Object-oriented programming provides the designer with sound software engineering principles, such as data encapsulation and inheritance. Constraint programming allows for declarative modeling of relations among (graphic) objects, while the maintenance of these relations is done automatically by the system environment.

In two groups at the Eindhoven University of Technology, research is done regarding the usefulness of constraints in a graphics environment. The first research group is the VR-DIS group at the Architecture Department. Here, the use of constraints in a 3-dimensional virtual reality design environment for architects is investigated. Typical constraints that can be useful for architects are, for example, constraints to specify the dimensions of a room, sizes of walls, doors, windows, and relative positions of elements. In the VR-DIS project, an object-oriented environment is under development which allows for 3-dimensional modeling of architectural elements. The environment enables constraints to be specified among these elements. A constraint system is needed that provides a set of constraints and appropriate constraint solvers.

The second research group is the Computer Graphics group at the Department of Mathematics and Computing Science. Here, a 3-dimensional animation system, the GDP, has been developed that applies constraints to model animations. Special constraints to connect rigid bodies and specific techniques for solving the constraints, such as inverse kinematics and dynamics, enable the modeling of animations that simulate physical processes. Some examples are dangling chains, bicycle rides, and roller coaster rides. Animations in the GDP are programmed in an object-oriented scripting language, called Looks. In this animation system, it is not possible to specify constraints on non-rigid objects. Therefore, the need arose for a broader class of constraints, such as constraints for intersection tests, constraints for relative positioning, and constraints that can be applied to non-rigid objects. For this environment, a constraint system is needed which provides a uniform structure for the Looks programmer while allowing him to implement his own constraints and constraint solvers.

The VR-DIS project and the GDP project provide two different areas for which one solution has to be found. The difference is the fact that in the VR-DIS project, an architectural design system is developed. In the GDP project, the animation system is the software environment. Furthermore, the end-user in the VR-DIS project is an architect, while in the GDP project, it is the Looks programmer. However, there are also similarities. These include the facts that both groups deal with object-oriented software development, 3D graphics, and the need to apply constraints on graphical objects.

This thesis describes the design process that lead to the constraint system for the VR-DIS project and the GDP project. In the design process, 6 different phases can be distinguished:

1. Domain analysis (Chapter 2),
2. Problem analysis (Chapter 3),
3. Outline of a solution (Chapter 4),
4. Specification of the solution (Chapter 5),
5. Prototype implementation (Chapter 6),
6. Application implementation (Chapter 7).

In the domain analysis phase, a broad overview is given of the two domains in which the research was performed, object-oriented programming and constraint programming.

In the problem analysis phase, the problems are identified that have to be solved. Before narrowing down to the specific requirements of the constraint system, we first consider the combination of constraint programming and object-oriented programming in general. Although both paradigms offer powerful techniques to build graphics applications the combination of the two is difficult because of the following conceptual incompatibilities.

1. Object-oriented programming is imperative, constraint programming is declarative.
2. The information hiding principle of object-oriented programming encapsulates the data, while solvers need this data for constraint satisfaction.

Most existing systems that combine objects and constraints do not provide a general approach to deal with these incompatibilities.

Therefore, in the outline of a solution (Chapter 4), we decided to first build a generic model. The model should enable the paradigms to co-exist while preventing that either the object-oriented paradigm or the constraint paradigm looses any of its typical strengths. Furthermore, this model should be sufficiently generic to support different types of constraints and solving techniques. Such a generic model provides a well-founded basis to support versatile constraint systems. Additionally, the model can be easily extended to other application areas, such as simulation and visualization, user interface design, multimedia applications, and provide a general way of applying constraints here.

The conceptual model was developed for the combination of objects with constraints. It radically separates the object-oriented paradigm from the constraint paradigm. Communication between the two systems is done via events and data flows. This communication strategy is completely orthogonal to the message passing activity of the objects, and this is used to solve the above mentioned incompatibilities. The separation between the paradigms served as the starting point for defining the entities that occur in the model and their interaction.

In the specification phase (Chapter 5), the conceptual model, now called **CODE**, is further worked out in detail. All events and data flows are determined and a precise operational description of the model is given. Tables are used to specify the communications and events of the various entities. Diagrams describe the characteristics of the modes that each entity can be in.

Because the implementation of the conceptual model is not trivial, first, a prototype implementation was built in the next phase (Chapter 6). This prototype implementation was set up to resemble the conceptual model **CODE** as closely as possible. The entities of the model are implemented as concurrent processes using the language **MANIFOLD**. In this way, the implementation serves as a case study for the verification of the model and as a "blueprint" implementation in which all software components are identified. The prototype implementation, called PROTOM, was used to build a graphical drawing tool which demonstrated a number of constraint solving techniques.

In the application implementation phase (Chapter 7), the application is built for the VR-DIS project and the GDP project. This application, the constraint system called SCAFFOLD, is based on the conceptual model. That is, the entities of the conceptual model occur in the constraint system and communicate as is described by the model. In cooperation with the Architecture department, specific constraint types and solving methods were determined for the VR-DIS project in order to allow architects to experiment with constraints. Examples of constraint types are touch constraints and distance constraints between architectural elements such as walls, doors, and floors. For the GDP project, an interface was created between the constraint handling engine and Looks language. Via this interface, a generic structure is offered to the Looks programmer for the definition of his own constraints and solvers. In order to demonstrate this interface, the constraint types that were developed for the VR-DIS project are also incorporated in the Looks language.

In the last chapter of the thesis, Chapter 8, an evaluation is given regarding the design process and the design methodology. Furthermore, general conclusions are given concerning the combination of object-oriented programming and constraint programming. The chapter concludes with some directions for further research.

Since this thesis is carried out as a designer's Ph.D., an important aspect is the description of the design process. For this purpose, important design decisions are set off from the main text by extra space. They are headed by the text **Design Decision** followed by a label, and are ended by a square (□). For example,

Design Decision 1.1
In a separate piece of text like this, an important design decision is described.

1. This is the first alternative.
2. This is the second alternative.
3. This is a third alternative.

After the evaluation of the alternatives, one of them is chosen and the reasons for the decision are explained. □

Chapter 2

Background

In this chapter, we present two major approaches that are used in the development of computer graphics systems. These are object-oriented programming and constraint programming. The aim is not to give an exhaustive study of these two fields, but to give a comprehensible overview, that will be sufficient for an understanding of the subsequent chapters.

2.1 Object-Oriented Programming

An object-oriented (OO) approach to software design was derived from work on information hiding [Parnas, 1972], abstract data types [Lizkov et al., 1974], and work on object-oriented programming languages like Smalltalk [Goldberg et al., 1983]. Often, object-oriented design and object-oriented programming languages are treated as though they were the same. However, OO-design is a way of considering software design. It is independent of a programming language. OO-languages support notions that are used in OO-design and allow that such a design can be implemented directly. An OO-design can also be implemented in another language which does not provide these specific object-oriented notions.

OO-design is characterized, among other things, by a design strategy called *information hiding*. This means that as much data as possible is hidden within components of the system that is being designed. These components are called the *objects*. Objects communicate with each other via operations called *messages*.

An advantage of OO-design is that shared data areas are eliminated since every object conceals its state within that object [Sommerville, 1989]. This leads to loosely coupled systems in which any one object can be replaced with another object that responds to the same set of messages. A second advantage is that objects are independent entities and control access to their internal state. In this way, illegitimate access to the object's internal state, deliberate or accidental, can be reduced. A third advantage is that objects may also be distributed or may execute either sequentially or in parallel. However, decisions of this kind do not have to be made early in the design process.

In [Wegner, 1989], four software engineering goals of object-oriented programming are identified. These are the development of *software components*, *software libraries*, *capital-intensive software technology*, and *very large scale object-oriented programming*.

Managing the complexity of a large software task is best done by splitting the task into smaller parts or *components*. Software components include functions, procedures, objects, but also processes, actors, and agents. Object-oriented programming provides systematic techniques, such as objects, classes, and inheritance, for managing these software components.

Libraries are repositories of software components that serve as reusable blocks for software development. This idea dates back to the earliest days of computing in the 1940s and 1950s. However, implementation of the idea has proven to be difficult. This is due to diversity of software and hardware environments. Creating libraries for different environments is a complex task and often flexible re-usability may conflict with efficiency requirements. Object-oriented libraries provide classes from which objects can be created. This availability of classes promotes their re-usability and provides a uniform interface to created objects so that they all can be treated alike.

Software like compilers, operating systems, and software components are *capital* goods in industry. This means that there is an economic motivation for software to be reusable so that software productivity is enhanced. Object-oriented programming is an attempt to achieve this goal. However, in practice, object-oriented software systems typically have a non-object-oriented system structure to improve efficiency.

Very large scale object-oriented programming means programming that is large in program size, large in development time, large in number of people, and large in educational and technical infrastructure. Object-oriented programming provides a framework which facilitates the management of such large systems. However, Wegner asserts that a programming language within megamodules of a large system can well be object-oriented, but that interface and coordination requirements among such megamodules might go beyond object-orientation.

Indeed, if we consider the case of combining an OO-system with a constraint system, which is the subject of this thesis, there arise problems that are difficult to solve if we adhere strictly to traditional object-oriented programming techniques. Chapter 3 discusses these problems and in Chapter 4, a solution is presented.

2.1.1 Basic Concepts

Basic concepts in OO-design are *classes*, *objects*, *inheritance*, *polymorphism*, and *information hiding*. In literature, the terminology is not always in accordance with each other. In this section, definitions are given of the concepts that are often utilized and which will be used in the remainder of this thesis.

An object is a discrete, distinguishable entity. It has its own identity and can communicate with other objects. It consists of a data structure, an interface through which other objects can communicate with it, and a behavior. The elements in the data structure are called the *internal variables* or *instance variables* of the object. The values of the internal variables determine the *state* of the object.

The interface of an object is described by its *messages*. A message is a function name with an (optional) list of parameters. A message can be called by the object itself or by other objects. In the latter case, the message is said to be *sent* to the object. The mechanism of objects sending messages to each other is called *message passing*. When a message is called or sent, code that is associated with the message is executed. This associated code of a message is called a *method*. It can be a certain action or a transformation on the internal variables. The set of methods realizes the actual behavior of an object. Messages can be *polymorphic*. This means that the same message can have different behavior when sent to different objects, due to inheritance and subtyping (*inclusion* polymorphism), or the message can be called with different parameter types (*parametric* polymorphism).

A class is an abstraction that serves to describe objects with the same data structure and behavior. An object is an instance of a class and has its own value for each internal variable. Classes can be organized hierarchically.

A class can be refined by creating *subclasses* of that class. The class being refined is called the *superclass*. A subclass is said to *inherit* the data structure and behavior of its superclass. It can add its own internal variables and messages to the ones it inherits, and can also rewrite existing methods. In-

heritance is used to implement different abstraction mechanisms. For example, *specialization*, where a class lower in the hierarchy is a more detailed version of its superclass. Or, *aggregation*, where classes lower in the hierarchy are components of the superclass (see [Taivalsaari, 1996]).

The internal variables of an object are hidden from other objects. This is called *information hiding*. Information hiding is also often called *data encapsulation* since data is encapsulated within the object and cannot be directly accessed from outside.

Data encapsulation is complementary to *data abstraction*. In the general meaning, an abstraction is a simplified description or specification of a system in which important aspects are emphasized while minor details are omitted. In the object-oriented context, this means abstracting from the data representation of an object and focusing on the object's behavior. While abstraction focuses on the outside of an object, encapsulation realizes this by hiding the object's internals.

2.1.2 Object-Oriented Modeling

Object-oriented development methodologies assert to be more than just a way of programming. The difference between object-oriented software development techniques and traditional functional approaches to design is that the latter place primary emphasis on decomposing the system into functional components. This often needs some kind of transformation of the problem space to tackle a problem. Object-oriented techniques intend to decompose a problem directly in objects that approximate more a person's perception of reality [Booch, 1986].

Important object-oriented modeling techniques that are currently widely used in industry, but are also continuously under development are Object-Oriented Design (OOD) [Booch, 1986], object-oriented modeling with use cases [Jacobson, 1995], object-oriented modeling using *design by contract* [Meyer, 1997], and modeling techniques based on the Unified Modeling Language (UML) (see, for example, [Fowler, 1997], [Warmer et al., 1999]).

In general, object-oriented design methodologies are similar to each other on the main points and differ from each other by putting emphasis on different phases or aspects of the design process. Below, an overview of the Object Modeling Technique (OMT) by [Rumbaugh et al., 1991] is given.

Object Modeling Technique (OMT)

OMT is a method for designing software systems in an object-oriented fashion, regardless of the final implementation language. In [Rumbaugh et al., 1991], the software engineering cycle is described as consisting of six stages:

1. Analysis,
2. Design,
3. Implementation,
4. Integration,
5. Maintenance,
6. Enhancement.

OMT deals with the first three stages of this engineering cycle. The latter three stages are not explicitly addressed. However, it is argued that a clean design in a precise notation also facilitates these stages. OMT uses a graphical notation for expressing object-oriented models which is the same in all stages of the method.

The OMT methodology consists of four stages.

1. *Analysis*. In this stage, a model of the real world situation is built. Described is *what* the desired system must do and not *how* this should be done. The objects in the model are application domain concepts.
2. *System design*. In this stage, high-level decisions about the overall system architecture are made.
3. *Object design*. Based on the Analysis and System design models, a model is built that contains implementation details. In this stage, data structures and algorithms are designed to implement classes.
4. *Implementation*. The final stage is to translate the object design into a particular programming language, database, or hardware implementation.

During each of these four stages, three kinds of models are used in OMT to describe a system. These are listed below.

1. *Object model*. The object model presents a static structure of objects and their relationships. It is built up of *object diagrams* in which nodes represent object classes and arcs between nodes represent relations among classes. Using the graphical representation, one can express *classes* and instances of classes, the *objects*. Examples of relations are objects that have references to other objects, objects that contain another object, and inheritance relations among classes.
2. *Dynamic model*. This model specifies and implements the *control* aspect of the system and describes those aspects that change over time. It contains *state diagrams* in which the nodes represent states and the arcs between nodes are transitions caused by *events*. Furthermore, there are means in the model for expressing nested states and features for dealing with concurrency aspects.
3. *Functional model*. This model describes the data value transformations, or computations, that take place in a system. It does not show control information. This belongs to the dynamic model. Its diagrams are *data flow diagrams* in which nodes represent processes and the arcs are data flows.

The three models are orthogonal to each other. They describe different aspects of the system, but contain references to each other. The object model is the most fundamental. It describes *what* is changing, while the other two describe *when* and *how* these changes take place.

In each model, it is also possible to add *constraints* which describe relations among parts of the model and which have to remain invariant. In the object model, constraints describe relations among objects, in the dynamic model, constraints specify relations among states, and in the functional model, constraints put restrictions on operations. Constraints can range from precise specifications of values to vague descriptions of desired behavior.

The OMT methodology is widely used in industry to model large applications in an object-oriented manner. Other object-oriented methodologies identify similar phases and differ in the emphasis that is placed on specific aspects or notation methodology. The main similarities among the software design methodologies are that all consist of several phases, each phase going top-down going from global to detail, and in each phase creating a formal or semi-formal description of the system (using different kinds of diagrams). Usually, the presentation of these phases is as if they are passed through in a sequential manner. However, as is often pointed out in the design methodology itself and as is experienced in practice, different phases can be carried out iteratively or in parallel.

For example, many similarities can be discovered between the design methodologies of Rumbaugh and Booch. They both use roughly three models to represent the system, a model that describes the object and class structure, a model for describing dynamic aspects of a system, and a model for specifying control information. Rumbaugh compares OMT to the design methodology of Booch and

argues that OMT places more emphasis on analysis and less on design, and identifies the increased emphasis on associations in OMT as a major difference between the two methodologies. He concludes by remarking that the resemblance between the two is more striking that the difference.

2.1.3 Languages

Object-oriented programming languages incorporate many of the notions that are used within object-oriented design, such as objects, classes, inheritance, information hiding, and polymorphism. Together with the OO-design technique, development of OO-languages started in the beginning of the 1980s. Since then, many applications have been developed using an object-oriented approach, for example, in the areas of databases, user interface design, computer graphics, animation, computer aided design.

An object-oriented language does not necessarily implement all the features of an OO approach and often an OO-language provides facilities that one would not designate as object-oriented. However, they can be used to easily implement object-oriented designs. Often, design problems that frequently occur in a given application area are solved by applying *design patterns* [Gamma et al., 1995]. For a broad class of problems, design patterns have been developed. They describe the characteristics of such a problem, and provide a solution to it using object-oriented techniques. Next to this, design patterns discuss the consequences of applying them.

A comprehensive introduction into object-oriented programming is given in [Budd, 1997]. The book extensively treats OO mechanisms and compares various languages such as C++, Objective-C, Java, Smalltalk, Object Pascal, and Eiffel.

2.2 Constraint Satisfaction Techniques

Constraints are relations among variables that have to remain valid. The variables range over a certain domain for which, in general, notions of what constitutes a relation and what should be done with a collection of relations are defined. An example of a domain is the domain of real numbers, and relations over this domain are equality and inequality. Constraints on real numbers can be specified as (non-) linear equations.

Constraint programming systems always have two components. The first is the *declarative* component. The goal here is to *express* the constraints in some kind of (programming) language. The second component has a *procedural* nature and consists of algorithms for *solving* the constraints. Solving a set of constraints can, for example, mean deciding if there is a solution that satisfies a set of constraints, finding a solution that satisfies the constraints, or finding all solutions.

Early research in constraint programming was done in Artificial Intelligence. The main focus here was designing algorithms for solving sets of constraints. Since then, in many other fields constraints became an important research topic, for example, in the area of logic programming. In this area, mainly languages have been developed for expressing constraints. In many cases, algorithms developed in AI were used to solve the constraints. Also in areas such as computer graphics, CAD/CAM, and user interfaces, research is being done to explore whether constraints can be used to solve complex problems. In the next sections, an overview of some important areas is given in which constraints are investigated.

2.2.1 Constraint Programming

Constraint logic programming descended from logic programming which became well-known via the Prolog language. A drawback of this language is its poor efficiency in problems that give rise to a large

solution space due to Prolog's general resolution process. CLP languages generalize the resolution process by adding the notion of constraints. The logic programming resolution process is now seen as just an instance of a more general scheme that can also solve constraints. The family of CCP (Concurrent Constraint Programming) languages can be seen a generalization of the CLP scheme.

In this section, we present a brief overview of logic programming, constraint logic programming, and concurrent constraint programming.

Logic Programming

Logic programming is based on the idea that computation can be seen as a controlled deduction [Hentenryck, 1989]. It was born out of the work of Colmerauer and Kowalski [Kowalski, 1974], [Colmerauer et al., 1973]. The powerful formalism of logic programming was introduced by Kowalski in 1974. It grew out of work on automated theorem proving and was adapted so the new formalism could also be used for computing. Colmerauer and his team designed a new programming language, Prolog, which was based on the logic programming paradigm. Since then, new languages appeared based on this paradigm and they were used to tackle various computationally complex problems.

Logic programming languages operate on uninterpreted *terms* which are constructed from constants and function symbols, generally called the Herbrand Universe. A logic program consists of two parts. The first one is the declarative part, which specifies *what* has to be computed. The second part is the procedural part, which tells *how* the computation takes place. Writing a logic program consists of writing the declarative specifications (the *what* part). The computation mechanism of the language derives the logical conclusions from these specifications (the *how* part).

Prolog is the most prominent representative of logic programming. Although there are differences between Prolog and the "pure" logic programming paradigm (small but important differences that have to do with efficiency, need for better expressiveness, and ease of computing), often the language is used to explain the intrinsics of the general logic programming paradigm.

A Prolog program consists of *clauses* which are of the form, $head \leftarrow goal_1,\ldots,goal_n$. The *head* and *goals* are composed of terms. A clause is read as, "In order to make *head* true, first $goal_1,\ldots,goal_n$ have to be made true." All clauses whose heads have the same name and arity, together are called a *predicate*. There are two kinds of clauses. The first kind is when n equals 0 (no goals). These are called *facts*. The second kind are called *rules* in which case n is greater than 0.

"Headless" clauses are called *queries* and are used to retrieve information from the current set of facts and rules. Given a program and a query, a search space is defined in which a logic programming language will try to find an answer to the query.

In order to find solutions, logic programs use a computation mechanism called SLD-resolution. SLD-resolution stands for *S*election rule driven *L*inear resolution for *D*efinite clauses. The resolution process is a mechanical method for proving statements of first order logic, introduced by J.A. Robinson in 1965 [Robinson, 1965]. It is applied to two clauses in a rule. It eliminates, by *unification* (see below), a term that occurs "positive" in one clause (the *head*) and "negative" in the other (one of the *goals*) to produce a new clause, the resolvent. Linearity means that each resolvent depends only on the previous one, so that derivations become sequences. Definite clauses are the clauses as described above. The selection rule chooses a term from the current resolvent that is used in the next unification step.

Given a set of facts, rules, and a query, the SLD-resolution process derives a computation that yields a substitution θ which assigns values to the terms in the query. SLD-resolution does not only compute a solution, the derived computation is also the proof of the query with valuation θ from the set of facts and rules.

The basic ingredient of SLD-resolution is *unification*. Unification is a mechanism to assign values to terms which differs from assignment in imperative programming languages (unification *substitutes* a term by another term which makes statements like `x:=x+1` more complex to express). Construction of the proof by SLD consists of basic steps which consist of replacing parts in the query using facts and rules of the program. When two terms cannot be unified, backtracking is done to explore other branches in the search space. For a comprehensible introduction into logic programming and Prolog, see [Apt, 1997].

Unfortunately, the implementation of the resolution process in logic programming languages gives bad performances for problems that generate large search trees, since their basic computational paradigm is *generate-and-test*. That is, first the complete search tree is generated and next the validity of each leaf is tested using unification and backtracking. This drawback lead to the development of *constraint logic programming languages* (CLP), where the computational paradigm is *constrain-and-generate*. That is, a CLP language uses specific knowledge to reduce the search tree and thus generates a smaller search tree to traverse.

Constraint Logic Programming

The scheme that underlies the Constraint Logic Programming paradigm defines a family of languages for reasoning about constraints using a logic programming approach [Heintze et al., 1987]. The CLP scheme adds constraints to the resolution algorithm that lies at the basis of Logic Programming. A CLP language operates on a certain computation domain and it uses constraints and knowledge of the domain to rule out possible outcomes before the search tree is constructed.

The main advantage of CLP is that it combines the declarative aspect of logic programming and the efficiency and expressiveness of constraints. The disadvantage is that the complexity of the constraint solver algorithms can affect the performance. To overcome this problem, a lot of research has gone into developing efficient algorithms. Also, constraint algorithms developed in AI are used extensively. Other techniques to improve solver efficiency are to apply compile-time optimizations and parallelism (see below).

The general CLP($\mathcal{X}$) scheme was developed by Jaffar and Lassez in 1987 [Jaffar et al., 1987]. The $\mathcal{X}$ is the domain on which the constraints in the CLP language operate. Jaffar et al. showed that CLP($\mathcal{X}$) has the desirable characteristic that important properties of logic programming also hold for the CLP scheme. Like in logic programming, a constraint logic program consists of facts and rules. The difference is that next to the predicates, there exist constraints which have the same declarative, but different operational semantics.

CLP clauses are of the form, $head \leftarrow cstr_1,\ldots,cstr_m,goal_1,\ldots,goal_n$. The *cstr*s have the same semantics as the *goal*s, however, operationally they are treated differently. The constraints are dealt with by special algorithms depending on the domain of the constraints. For example, finite domain constraints are often solved by algorithms that were developed in AI, numerical constraints are solved by dedicated numerical solvers. Although the order of the *cstr*s and *goal*s in a clause is, theoretically, of no importance, operationally constraints are best put in front. This is because *cstr*s and *goal*s are treated in the order in which they occur. Putting constraints up in front means that first the search space is pruned after which the logic programming part invokes the SLD-resolution scheme to meet the *goal*s.

Various languages have been built based upon the CLP($\mathcal{X}$) scheme. For example, CLP($\mathcal{R}$) for variables which range over real values, CLP(BNR) for variables that take boolean values, CLP(FD) for finite domain variables, and CHIP which provides constraint handling in Prolog. Commercial constraint logic programming systems and languages include ILOG, Eclipse, and Prolog IV. The

ILOG Solver provides optimization solvers and algorithms to work with C++ programs. Eclipse is a constraint logic programming environment that provides several types of solvers for scheduling, planning, resource allocation, and transport. Prolog IV, a successor of earlier Prolog versions, provides solvers for problem domains such as scheduling, linear and non-linear optimization, and simulation. It allows constraints to be specified on variables with various domains, such as real, integers, and booleans.

Concurrent Constraint Programming

The CCP paradigm extends the CLP scheme by adding the notion of concurrency to a constraint language [Saraswat, 1993]. It is based on concurrent logic programming and adds the notions of variables, constraints that relate the variables, and systems that maintain the constraints. CCP languages may be built on top of different constraint systems that operate on different constraint domains, such as arithmetic and finite domain variables. CCP languages provide control to enforce strict sequential control as well as non-determinism.

In CCP languages, it is possible to model object-oriented programming by representing objects as perpetual processes that consume a stream of messages [Kenneth et al., 1986]. However, constraints in such languages are not closely coupled with this object representation, but more closely resemble constraints in CLP languages [Lopez, 1997].

2.2.2 Constraint Solving

Research in constraint reasoning probably started in Artificial Intelligence. Here, many problems are formulated as Constraint Satisfaction Problems (CSPs) and specialized techniques have been (and are being) developed to solve them. In many other application areas, these techniques are used to solve problems, such as temporal reasoning, resource allocation, and scheduling. Also, the role of CSP algorithms play an important role in (constraint) logic programming (see Section 2.2.1).

In this section, the book of [Tsang, 1993] is used as a guideline to give an overview of constraint satisfaction in Artificial Intelligence.

What is the Constraint Satisfaction Problem?

The Constraint Satisfaction Problem (CSP) in AI is formally defined as a tuple $\langle Z, D, C\rangle$. Z is the (finite) set of variables ($Z = \{x_1, ..., x_n\}$). D is the set of domains for each variable in Z ($D = \{D_{x_1}, ..., D_{x_n}\}$). Each D_{x_i} is a finite set of *values* that variable x_i can take. Values in D_{x_i} can be *assigned* to x_i. This is denoted as a *label*, $\langle x_i, v_i\rangle$ (value v_i is assigned to variable x_i). Simultaneous assignments to variables can be done by *compound labels*, $(\langle y_1, v_1\rangle, ..., \langle y_k, v_k\rangle)$. A compound label consisting of k labels is called a k-compound label.

A *constraint* is a set of compound labels that represents *legal* assignments to variables. For example, the constraint $c = \{\langle y, 3\rangle, \langle y, 4\rangle\}$ specifies that variable y may take value 3 or 4. A label $\langle x, a\rangle$ is said to *satisfy* a constraint c, if $\langle x, a\rangle \in c$. A solution to a constraint problem C is an n-compound label which satisfies all constraints $c \in C$.

Solving the Constraint Satisfaction Problem

Finding a solution for a CSP=$\langle Z, D, C\rangle$, means finding an n-compound label $(\langle x_1, v_1\rangle, \ldots, \langle x_n, v_n\rangle)$ such that all constraints $c \in C$ are satisfied. In many cases, the complexity of this problem is NP-hard where the time needed for an unconstrained search increases exponentially (or worse) with

the problem size. Even for a binary CSP, where all constraints are 1- or 2-compound labels, this is the case. Famous examples of problems on which constraint techniques have been applied are the N-queens problem, graph-coloring problem, the scene labeling problem, and the resource allocation problem. Tsang classifies the CSP solving techniques into three categories:

1. Problem reduction,
2. (Tree) search,
3. Solution synthesis.

Problem reduction techniques transform a CSP into an equivalent problem that is hopefully easier to solve. This transformation can take place in two ways.

1. Removing redundant values from variable domains.
2. Removing redundant compound labels from the constraints.

By removing redundant values and labels, possibly the reduced problem is easier to solve, because fewer labels have to be considered. If, by removing redundant values and labels, the domain of any variable or any constraint is reduced to an empty set, the problem is insoluble. Problem reduction is usually used as a pre-processing step to prune the search space. However, there are also algorithms that apply these techniques to prune the solution space during the search. In the literature (see [Mackworth, 1977], [Mackworth et al., 1985], [Freuder, 1982]), problem reduction is often referred to as *consistency maintenance* or *consistency checking*. Consequently, a reduced problem can acquire different "levels" of consistency, such as *node-consistency*, *arc-consistency*, or *path-consistency*. Below, we will take a closer look at the first two.

We call a constraint satisfaction problem *node-consistent* if and only if for all variables, all values in its domain satisfy the unary constraints on that variable [Mackworth, 1977]. The following algorithm achieves node-consistency for discrete domains.

```
1. NC (V, C)
2.   for (∀ v ∈ V)
3.     for (∀ s ∈ domain of v )
4.       if (∃ violated unary constraint in C on v) then
5.         remove s from domain of v
6.       fi
7.     rof
8.   rof
9. CN
```

If the size of each domain is at most a and the total number of unary constraints is e, then the time complexity of this algorithm is $\mathcal{O}(ae)$. The algorithm is easily adapted for continuous domains. The time complexity then is $\mathcal{O}(Re)$, with R the time needed to check a single constraint.

We call a constraint satisfaction problem *arc-consistent* if and only if for all values in the domain of each variable v, and for all constraints c on v, we can find a value in the domain of the other variables of c that satisfies the constraint.

The following algorithm for achieving arc-consistency assumes that the constraints are unary or binary, and have discrete domains. For each binary constraint `c` on variables $\texttt{v}_1$ and $\texttt{v}_2$, the domains of the variables are examined separately by the method `revise`. When the domain is examined, all values are deleted which do not satisfy the constraint. If any value is removed, the other constraints on the variable must be reconsidered. We first present the method `revise` of constraint `c`.

```
 1. Boolean revise(c,v1,v2)
 2.   changed := FALSE
 3.   for (∀ s1 ∈ domain of v1)
 4.     delete := TRUE
 5.     for (∀ s2 ∈ domain of v2)
 6.       if ((s1,s2) satisfies c) then
 7.         delete := FALSE
 8.         break
 9.       fi
10.     rof
11.     if (delete) then
12.       remove s1 from domain of v1
13.       changed := TRUE
14.     fi
15.   rof
16.   return changed
17. esiver
```

The following algorithm for arc-consistency is called AC-3, after [Mackworth, 1977].

```
 1. AC-3 (V, C)
 2.   NC (V, C)
 3.   while ( C ≠ ∅ ) do
 4.     select (c1, v1, v2) from C
 5.     remove (c1, v1, v2) from C
 6.     if (revise(c1,v1,v2)) then
 7.       for (∀(c2,v2,v3) ≠ (c1, v1, v2))
 8.         add (c2,v2,v3) to C
 9.       rof
10.     fi
11.   od
12. 3-CA
```

Its time complexity is $\mathcal{O}(ea^3)$, with a the maximum domain size and e the number of binary constraints, which is linear in the number of constraints. If the constraint graph is planar, then the time complexity is also linear in the number of variables [Mackworth et al., 1985]. A less simple but optimal algorithm, AC-4, is presented in [Mohr et al., 1986]. It has time complexity $\mathcal{O}(ea^2)$.

Achieving node and arc-consistency does not solve the constraint satisfaction problem. These algorithms are used to reduce the search space of the problem. For example, when there are no cycles in the constraint graph, achieving node-consistency and arc-consistency implies that the graph is backtrack-free, that is, a solution can be found without backtracking [Freuder, 1982]. In a backtrack-free graph, a solution is found in time linear in the number of nodes [Dechter et al., 1988]. When there are cycles in the graph, additional algorithms have to be used to reduce the problem, such as the recognition of certain patterns (for example, k-trees).

Solutions are found by *searching*. the basic search algorithm checks all values in the domains of all variables against all constraints. This algorithm uses *simple backtracking* to find solutions in the search space (which is a tree). The basic operation is to pick one variable at a time and assign a value from the domain to it. If the assignment violates some constraints, another value, when available, is chosen. If all variables are assigned a value, the problem is solved. If at any stage no value can be found for a particular variable without violating any constraints, the variable which was last picked is revised and an alternative value, when available, is assigned to that variable. This carries on until either a solution is found or all combinations of values have been tried and have failed. Below, the backtrack algorithm is given in pseudo code (see [Tsang, 1993]).

```
1. Boolean BT (V, C)
2.   if (V = ∅) then
3.     return TRUE
4.   fi
5.   v := select variable from V
6.   while (domain(v) ≠ ∅) do
7.     s := select value from domain of v
8.     v := s
9.     remove s from domain v
10.    if (v violates no constraints in C) then
11.      success := BT(V − {v}, C)
12.      if (success) then
13.        return TRUE
14.      fi
15.    fi
16.  od
17.  return FALSE  // no solution found, backtracking is done
18. TB
```

This recursive algorithm is called with a set of unassigned variables V and the set of constraints C. Each time BT is called, it picks a value from the domain of the current variable, assigns this to the variable, and removes the value from its domain. Then, it is tested if the current assignment together with all previous assignments violate any constraints in C. If this is the case, another value from the variable's domain is chosen. If the domain runs empty, no solution can be found for this particular variable and backtracking is performed. The time complexity is $O(a^n e)$ (where a is the maximum domain size, n is the number of variables, and e is the number of constraints).

In order to speed up the search, more advanced algorithms have been developed that exploit specific features of a constraint problem. An easy way to improve the simple backtracking algorithm is to introduce a threshold, say b, that prevents the algorithm to exhaust one branch of the search tree before turning to another one. If a certain node in the tree has been visited b times, then unvisited children will be ignored. It can be proven, using arguments from probability theory, that this strategy spreads the computational effort across the choices for variables and values more evenly [Ginsberg et al., 1990].

Other strategies make use of the constraints to reduce the search space. For example, one strategy is to *look ahead* each time an assignment is done to a variable. The constraints are used to remove values from the domains of still unassigned variables that would violate any constraint. This technique is also often called *constraint propagation*.

Another strategy is to *gather-information-while-searching*. In this case, it is exploited that sibling subtrees in the search space are often similar to each other. Using this, an algorithm can identify and record sources of failure whenever traversing the tree and use this information to decrease the search space.

The third category of solving CSPs (the first one was problem reduction and the second one, searching) is called *solution synthesis*. In solution synthesis the distinctive feature is that solutions are constructively generated. Starting with an empty set of labels, in each step a label is added, such that the partial solution acquired thus far does not violate any constraints. Solution synthesis algorithms are especially useful for problems in which all solutions are required and so-called *tight* problems (see below).

A wide range of problem types have been subject to research to investigate how and whether they can be solved using constraint technologies. Some of these problem types are the following.

- Search Problem. Here, the aim is to find *one* solution that satisfies all the constraints.
- Optimization Problem. In case there are more solutions, find the *best* one (where it has to be defined what *best* means, using, for example, an objective function).
- "Search-for-all-solutions" Problem. The aim is to find *all* solutions that satisfy the constraints.
- Over-constrained Problem. There is *no* solution that satisfies all constraints. In that case, *near* solutions can be searched for. That is, assign values to variables, such that a given objective function reaches an optimal value.

In these problem types, a number of characteristics are distinguished that can be used to guide the solving process. Some examples are the following.

- The number of solutions that is required.
- The size of the problem (number of variables, number of values in the domain of a variable, number of constraints).
- The types of the variables.
- The *tightness* of a problem. That is, a *tight* problem has a relatively small number of solutions compared to the total number of values that all variables can take.
- The *quality* of the solution. For example, to be used in optimization problems.
- Partial solutions. When the problem is overconstrained, partial solutions can be searched for that violate as few constraints as possible.

Many algorithms to find solutions for CSPs exploit the fact that the domains of the variables are finite. In some cases these algorithms can be generalized to tackle constraint problems with infinite domains, however in general, the 'finite domain' algorithms are difficult to tailor to specific 'continuous domain' problems.

2.2.3 Tailored Constraint Approaches

Outside the AI and Logic Programming, there are many other areas in which constraints and constraint system are used. Examples are user interface design, CAD/CAM, animation, geometric modeling, multimedia. The difference is that the development of constraint systems in these areas is typically motivated by implementation issues or by specific applications. Within AI and logic programming, algorithms and languages are developed based on a formal framework with pre-defined semantic properties. The advantage is that formal proofs can be constructed concerning properties of these algorithms, such as the soundness, consistency, and complexity. A disadvantage is that it is often difficult to tailor these general algorithms to efficient implementations for a particular problem.

This section presents an overview of constraint techniques and constraint systems that have arisen outside the AI and Logic Programming enterprises.

Constraint Satisfaction Techniques

There are basically two different models that constraint solvers use for determining solutions (values for variables) to the constraints:

1. Alternation model,
2. Refinement model.

In the alternation (also called perturbation) model, each variable v_i gets assigned a single value s_i. If a variable's value does not satisfy the constraints on it, the constraint solver tries alternative values

from its domain (possibly eliminating the incorrect value from the domain). Satisfaction is completed if each variable has a solution such that all constraints hold.

In the refinement model, each variable v_i gets assigned a subset S_i of its domain D_i. If a value s_i in the proposed solution S_i does not satisfy the constraints on v_i, then the constraint solver eliminates that value. Satisfaction is completed when the solution set of each variable satisfies the constraints on that variable. After satisfaction, not every valuation $(s_1, \ldots, s_n) \in (S_1, \ldots, S_n)$ needs to satisfy all constraints, but for every $s_i \in S_i$, values $s_j \in S_j$, $j \neq i$ can be found that satisfy all constraints.

Many constraint algorithms in computer graphics use the alternation model with continuous domains of the variables. Many algorithms in artificial intelligence use the alternation model with discrete domains. In constraint logic programming, both models occur, although the refinement model is mostly used since the CLP scheme is based on the logic programming scheme. For example, languages CLP(FD), CLP(BNR), ILOG, and so on, use the refinement model to determine possible values for variables with discrete domains. CLP($\mathcal{R}$) uses the refinement model on variables with continuous domains.

Most constraint techniques and systems that are developed outside the AI or Logic Programming areas (and thus is characterized as a tailored constraint approach) apply a variation or combination of three basic algorithms. These algorithms are (see also [Leler, 1988]):

1. Propagation of known states,
2. Relaxation,
3. Propagation of degrees of freedom.

Propagation of known states, or just local propagation, can be performed when there are parts in the network whose states are completely known (have no degrees of freedom). The satisfaction system looks for one-step deductions that will allow the states of other parts to be known. This is repeated until all constraints are satisfied or no more known states can be propagated. If not all constraints can be satisfied, the remaining constraints must be resolved by, for example, numerical relaxation (see below).

Many constraint satisfaction systems use some form of local propagation. For an arbitrary constraint graph, we can use the following local propagation algorithm.

```
1.  LP (v1)
2.    for (∀ c1 on v1)
3.      fifo.push(c1,v1)
4.      while (NOT fifo.empty()) do
5.        fifo.pop(c1,v1)
6.        changed := revise(c1,v1)
7.        for (∀ v2 ∈ changed)
8.          for (∀ c2 ≠ c1 on v2)   // effectively
9.            fifo.push(c2,v2)      // LP (v2)
10.         rof
11.       rof
12.     od
13.   rof
14. PL
```

Algorithm `LP` is invoked with the variable that triggers the propagation. The constraints on the variables are pushed onto a first-in-first-out queue. In the while-loop every constraint in this queue is revised by possibly changing the variables subject to the constraint other than the variable that triggered the constraint. All other variables that have been modified are put in the set `changed`. These

variables and the constraints on them are put into the queue. Because constraints on changed variables are placed in a first-in-first-out queue (as opposed to an unordered set), they are guaranteed to be revised later on, whether the propagation is finite or infinite. This is called fair propagation [Güsgen et al., 1988].

If there are no cycles in the constraint graph, LP solves each constraint exactly once, and so the time complexity is linear in the number of constraints. Let R be the time needed to revise a constraint. The time complexity for LP is then $\mathcal{O}(Re)$. The time R depends on the problem. For discrete domains of size a, R is typically $\mathcal{O}(a)$ in the alternation model, and $\mathcal{O}(a^2)$ in the refinement model. If there are cycles in the constraint graph, there is in general no way to determine the complexity of the algorithm except for the refinement model on discrete domains. For example, for binary constraints, it becomes $\mathcal{O}(ea^3)$, equal to the time complexity of AC-3. For the other cases, let λ be the number of loops done by the algorithm. The time complexity is then $\mathcal{O}(\lambda Re)$.

In the case that the constraint graph contains no cycles, the variables of a solved constraint are not changed by the same constraint again during further propagation. However, if that value prevents other variables to satisfy constraints, no solution is found, since backtracking is not possible. This problem can be circumvented by allowing a constraint to change the value of the variable it was triggered from.

Relaxation is an iterative numerical approximation technique that is often used in cases where local propagation fails, such as cycles in the constraint graph. Relaxation can only be used on variables with continuous numeric values. It makes an initial guess at the values of the unknown variables, and then estimates the error that would be caused by assigning these values to the variables. New guesses are then made, and new error estimates calculated. This process repeats until the error is minimized. The number of iterations can also be limited by a constant in case the error does not converge fast enough. An algorithm for relaxation is given below.

```
 1. RX (V, C)
 2.   times := 0
 3.   error := ε + 1
 4.   while (error > ε AND times < λ) do
 5.     error := 0
 6.     times := times + 1
 7.     for (∀ v ∈ V)
 8.       guess initial value for v
 9.       error := max (error,error(v,C))
10.     rof
11.   od
12. XR
```

The algorithm RX applies to all the variables in the set `V`, subject to the constraints in the set `C`. The local variables of the algorithm, `times` and `error`, denote the number of iterations that have elapsed and the maximum error estimate, respectively. The variable `error` could also be a vector containing the error terms for all variables. In the while-loop, first initial values are guessed for the variables. Then the error is determined by testing the values of the variables against the constraints. The while-loop is repeated until the error term is smaller than a certain value ϵ or the maximum number of iterations λ, is reached.

Let again b be the number of variables and e the number of constraints. If we assume that a variable value can be guessed, and its error estimated, in time linear in the number of constraints on the variable, the complexity for guessing and error estimation of all variables is $\mathcal{O}(be)$. Since the number of iterations is at most λ, the time complexity of RX is $\mathcal{O}(\lambda be)$.

Although the relaxation algorithm is linear in the number of variables and constraints, in practice it can be slow because expensive floating point calculations have to be performed to guess values or estimate errors. Using local propagation however, relaxation can often be speeded up.

Prior to performing local propagation or relaxation, one can perform an analysis to plan the best order to propagate constraints [Sannella, 1994]. It is sometimes efficient to plan how to solve constraints before actually doing it. For example if one drags a graphics object which is constrained and must remain constrained, the planning can be done at the beginning of the drag operation, and the execution can be done in real time.

Propagation of degrees of freedom (DoF) is a technique where parts of the constraint graph that can easily be solved, are (temporarily) pruned off. DoF looks for a variable in the network with few enough constraints so that it can be changed to satisfy the constraints. If such a variable is found, it is removed together with the constraints that are imposed on it. If there are no variables with enough degrees of freedom, the resulting graph is solved by some other technique such as, for example, relaxation. After values for the variables in the resulting graph are determined, these values are propagated back to the removed branches.

DoF is often used in combination with relaxation. Parts of the graph that do not contain cycles are first removed and then the constraints inside the cycles are solved using relaxation. When the cycles are solved, the pruned off parts can be resolved using local propagation. The advantage is that the relaxation algorithm does not have to guess values for all variables in the network and thus the constraint graph can be solved much faster.

A problem with DoF is how to determine which variables in the graph have enough degrees of freedom, so that they can be removed. An often used heuristic is to look for variables which have only one constraint on them. Below an algorithm is given.

```
1.  DoF()
2.    v := find_var()
3.    while (v ≠ ε) do
4.      for (∀ c on v)
5.        fifo_push(c)
6.        remove_constr(c)
7.      rof
8.      fifo_push(v)
9.      remove_var(v)
10.     v := find_var()
11.   od
12. FoD
```

The procedure `find_var` looks for a variable with enough degrees of freedom and assigns this to `v`. If a variable is found, the `while`-loop is entered and, after the constraint and variable are stored in a fifo-list, they are removed from the graph.

Every constraint and every variable in the graph can be removed only once, thus the complexity is $\mathcal{O}(b+e)$. Despite the fact that DoF can speed up the relaxation algorithm in practice, it does not have an influence on the time complexity. The worst case is when no variables can be removed by DoF and relaxation still has to relax all variables in the constraint graph.

Constraint Systems

Many constraint systems outside AI and logic programming somehow find their roots in the Sketchpad system that was developed by Ivan Sutherland in the beginning of the 1960s [Sutherland, 1963].

Sketchpad was a geometric drawing application that allowed the user to draw objects such as lines, circles, rectangles, and relate these objects using constraints, such as point-on-line, point-on-circle, and collinear. The Sketchpad constraint system used local propagation and relaxation to satisfy the constraints. Sketchpad also included the facility for defining new constraint types in the underlying language.

After the pioneer work of Sketchpad, a silence set in that lasted for more than 15 years. At the end of the 1970s, Alan Borning developed ThingLab [Borning, 1981], a graphic simulation laboratory for constructing interactive, graphic simulations of experiments in physics and geometry, such as electrical circuits and mechanical linkages. The principle issue addressed in ThingLab was the representation and satisfaction of constraints which specified relations among parts of the simulation. ThingLab applied many of the ideas of Sketchpad to object-oriented programming in the Smalltalk language [Goldberg et al., 1983]. The used constraint satisfaction techniques in ThingLab were local propagation, relaxation, and propagation of degrees of freedom. One of the main contributions was the fact that for a fixed set of constraints, ThingLab could generate constraint satisfaction plans which could be compiled. This would tremendously improve the performance of constraint satisfaction. ThingLab influenced many other constraint systems and had some successors that implemented constraint solvers for user interface and animation construction systems (see [Malony et al., 1989], [Borning et al., 1986]).

In [Freeman-Benson et al., 1990], the authors present the DeltaBlue algorithm, a fast local propagation constraint solver. The DeltaBlue algorithm implements an *incremental* constraint solver, which means that the solver maintains an evolving "current solution" to the constraints. When constraints are added or removed, the solver does not solve the complete set of constraints again, but modifies the current solution to find a new one that satisfies all constraints. Implementations of DeltaBlue turn out to be fast algorithms. However, drawbacks are that DeltaBlue cannot handle constraints in a cycle.

A successor of the DeltaBlue algorithm is the SkyBlue algorithm [Sannella, 1994]. The most significant difference between DeltaBlue and SkyBlue is that the latter can handle cyclic constraints. This is done by calling external (that is, external to the SkyBlue algorithm) solvers to satisfy these constraints. The SkyBlue algorithm is implemented in the Multi-Garnet User Interface Development System [Sannella, 1994], which is a successor of Garnet [Myers et al., 1990]. Garnet is a user interface toolkit based on Common Lisp and X windows and provides a simple local propagation algorithm for so-called "one-way" constraints, that is, constraints that provide only one way to calculate a local solution.

In [Gleicher et al., 1994], Gleicher and Witkin describe a drawing program called Briar. Briar is similar to Sketchpad. It provides various drawing primitives and direct manipulation techniques to edit the primitives. Constraints among primitives, like controlling distances, positions, and orientations can be established, edited, removed, and visualized. Briar addresses the problem of adding constraints to a direct manipulation drawing program without detracting from what has made these programs so successful. An overview of the practical issues that concern a graphics drawing tool is given in [Gleicher, 1994].

In the area of geometrical constraints, many constraint solving techniques and implementations thereof have been developed. Geometric constraints often deal with overconstrained or underconstrained situations. Such situations can be resolved by giving practical feedback to the user who can provide new information to solve the problem (see [Noort et al., 1997], [Veltkamp, 1995]). Solving geometrical constraints usually involves solving sets of (non-) linear equations. Dealing with overconstrained or underconstrained situations can then be solved by using an objective function that should be minimized or maximized (see [Wesselink et al., 1995], [Donikian et al., 1995]). To reduce the number of numerical equations to be solved, often numerical equation solving is combined with

propagation techniques (see [Arbab et al., 1991], [Kramer, 1992]).

In early nineties, [Freeman-Benson, 1991] introduced a new family of languages that combined the imperative aspect of object-oriented programming with the declarativity of constraint programming. These were called the Constraint Imperative Programming (CIP) languages. A first instance of this family was Kaleidoscope [Freeman-Benson et al., 1992]. It is an OO language which provides constraints that can be imposed on objects. The constraint solver is an integral part of the language. The first version of Kaleidoscope served as a proof of concept and had some limitations. For example, the language semantics did not model state changes in the usual object-oriented sense and objects did not have an identity. In [Lopez, 1997], these drawbacks are solved and more and new constraint types are added. For example, *identity* constraints can be specified between objects which can be used to explicitly declare alias relations between objects. Other imperative constraint programming languages that were developed in this family are Siri [Horn, 1991] and COPE [Li, 1995].

CIP languages provide a smooth integration of object-orientation and constraints. However, since the constraint solver is an integral part of the language, it is not possible for a programmer to design new constraint types that the solver of the language cannot handle.

2.3 Conclusion

In this chapter, we have seen two major approaches that are currently being used and are being researched for the development of (large) software systems. Object-oriented design provides a software engineer with techniques and tools to decompose and control large and complex systems. Constraints allow an engineer to declaratively specify intricate problems, while distantiating from the actual solving of the constraints.

OO-design is not always the most natural way to design a system. In particular, systems that retain only minimal state information are best designed using another methodology than OO-design. In OO-design, decisions concerning representations of states can be made in a later stage of the design process. Thus, it promotes decoupling of system components and makes it more flexible to change when representation decisions have to be made [Wegner, 1989]. In [Borning, 1986], it is observed that the general inheritance mechanism poses some problems which are caused by the static definition of classes. For example, if a certain object of a class needs a (slightly) different interface than the class provides, a new (sub)class has to be defined. In the paper, the use of *prototypes* is described and compared to using classes. In [Aksit et al., 1992], obstacles that have been encountered in object-oriented software development are described. Problems are identified on three levels, the prepatory work (for example, during domain analysis), structured relations (for example, inheritance), and interactions among objects (message passing). In many cases, the OO model is too restrictive to describe relations and interactions in a flexible manner. For example, many OO methods cannot expres multiple views on objects. Despite the shortcomings, however, OO is considered a good starting point for building large systems and there is optimism that these problems can be solved.

Constraints are applied to declaratively specify relations. Depending on the domains of the constraint variables, different solving techniques can be used to solve or maintain them. The general CSP problem is a hard problem and satisfaction algorithms are often slow. One way to speed up the process is to perform computations in parallel. An interesting approach is taken in [Lipton, 1995], where massively parallel computations are done by means of chemical reactions on DNA strands. However, a more common approach to speed up the solving process, is to reduce the solution space by achieving some level of (node, arc) consistency.

In Table 2.1, an overview is given of the complexities we have discussed for the different constraint

	discrete		continuous	
	alt	ref	alt	ref
BT	ea^n			
NC		ae		Re
AC-3		a^3e		λRe
LP	λae	a^3e	λRe	λRe
RX			λne	

Table 2.1: Time complexities of some constraint algorithms. e is the number of constraints, a is the maximum domain size, n is the number of variables, λ is the maximum number of iterations, and R is the time needed to solve a single constraint.

solving techniques. We discriminate between the alternation and the refinement model of solving, and between discrete and continuous domains. As usual, a is the size of a discrete domain, n is the number of variables, e the number of constraints. R is the time needed to solve a single constraint, and λ the number of loops or iterations made by the algorithms.

All complexities are at least linear in the number of constraints e. It is immediately clear why local propagation LP is such a widely used constraint solving method. It is applicable in a wide variety of applications and linear in the number of constraints, if R is independent of e. Note however that LP need not always find a solution, which is why it is often preceded by a planning stage.

Both object-orientation and constraints are very applicable in the area of computer graphics. In the next chapter, we consider whether the two approaches can be combined and how they affect each other.

Chapter 3

Combining Objects and Constraints

Computer graphics systems are typically very large integrated programs that use a wide range of techniques. They may deal with various types of data and allow concurrent interaction with a user and between a large number of agents, actors, or active objects.

A well founded and appropriate underlying abstraction is needed to deal with the complexity of computer graphics. *Object-orientedness* provides such an underlying abstraction to deal with complexity. The concept of *constraints* is another such abstraction. Object-oriented programming provides powerful software engineering principles, such as inheritance and data encapsulation, which are needed to cope with large complex software systems. The principal benefit of declarative approaches is that they shift the burden of deciding how something has to be done from the application programmer to the system environment he is working in.

The justification for combining objects and constraints derives from the fact that both address the problems of complexity in large interactive graphical systems. Complexity arises in specifying the behavior of animations and interactions with many components or objects. Constraints allow the declarative modeling of the behavior of such systems. Secondly, complexity is due to the fact that we are dealing with large software systems.

In both the VR-DIS project and the GDP project, the applications, the architectural design system and the animation system respectively, are object-oriented. In these OO-systems, a constraint system has to be incorporated that enables relations between the objects to be specified and maintained. However, in order to set up a design in which the two paradigms are combined, we have to deal with the incompatibilities mentioned in the introduction of this thesis.

This chapter presents the combination of the two paradigms and treats the incompatibilities that arise. It is the second step in the software engineering trajectory that was outlined in Chapter 1 (the problem analysis). First, the problem domain is demarcated in Section 3.1. Section 3.2 gives an overview of existing systems that combine objects and constraints. In Section 3.3, the declarative nature of constraint programming is compared with the imperative nature of object-oriented programming. Section 3.4 treats the information hiding principle of the object-oriented paradigm in relation to the constraint paradigm. Section 3.5 compares several constraint satisfaction algorithms with respect to the amount of data that an object encapsulates. Finally, Section 3.7 gives the conclusions and proposes an approach for solving the problems that are set out in this chapter.

3.1 Problem Demarcation

In the VR-DIS project, a design system is developed to support 3D architectural modeling. Geometric constraints offer a designer powerful 3D modeling tools in such an environment. The design system aims to assist in the early phases of architectural design. The primary goal of these phases is to get a first impression about the overall shape of a building. Constraints as 3D modeling tools add geometric design knowledge which enables the modeling of rooms and elements in a building on a semantically high level.

In the GDP project, an animation system is developed to create 3D interactive animations. The animation system is controlled by a scripting language, called Looks. Looks can be used to create and delete objects that occur in the animation and prescribe their movements. A powerful tool to describe motions of objects is the use of constraints. Constraints that specify connections among objects allow for easy modeling of complex scenes. While the motion of one object is prescribed, the constraint system of the animation system can calculate motions of other objects that are connected via constraint to the prescribed object.

The design system of the VR-DIS project and the Looks language in the GDP project are both object-oriented in which constraints have to be incorporated. The goal of this thesis is to describe the design of a constraint system that provides constraint types in object-oriented graphics environments. In particular, a constraint system is designed and implemented for the VR-DIS project and the GDP project.

In defining a model for combining constraints and objects, two incompatibilities arise.

1. Object-oriented programming is imperative, while constraint programming is declarative (see Section 3.3).
2. Object-oriented design methodologies encourage data encapsulation, while this encapsulation is an obstruction for powerful constraint solving (see Section 3.4).

These incompatibilities exist on a conceptual level. The practical combination of the paradigms does not cause insurmountable problems. However, although existing implementations might lead us to believe that combining objects and constraints is a closed matter, in order to make a clean design that combines objects and constraints the approach that is laid out by the existing implementations do not satisfactory solve to the above mentioned incompatibilities. It is therefore necessary to analyze these incompatibilities and provide a solution to it.

In literature, various systems are described that combine both paradigms. In the next section, we examine some important systems that combine objects and constraints in a graphics environment.

3.2 Existing Object-Constraint Models

Existing systems that combine objects and constraints hardly ever address the information hiding conflict problem. Solutions in literature that explicitly deal with the information hiding conflict, try to assign values to an object by using the object's interface. In order to set an object's internal variables, the interface is extended with messages that can be called by a solver. In some cases, specific features of a system are used to allow a solver direct access to the internal variables, such as giving it certain privileges.

In [Laffra et al., 1991], the methods of an object that may violate constraints are guarded by so-called propagators. The propagators send messages to other objects to maintain the constraints. This technique is similar to the pre- and postcondition facilities in Go [Davy, 1991]. It is limited to con-

straint maintenance (that is, truth maintenance as opposed to starting with an inconsistent situation that is then resolved).

A more powerful technique is presented in [Wilk, 1991]. Here, a constraint system, Equate, uses *term rewriting* as a guide to find solutions. Constraints are specified as equations. Rewrite rules convert equations into equivalent sets of equations that can more easily be solved. This repeats recursively (zero or more times) until equations are simple enough to be rewritten to a set of instructions (messages to an object). The *rewrite rules* which rewrite the equations are provided by the classes and are similar to the program clauses of logic programs. Several rules may apply to rewrite an equation.

The partial solutions for every constraint have to be combined in order to provide a solution program for the whole constraint problem. Equate uses so-called *read-sets* and *write-sets* to determine which instructions interfere with the accomplishments of others. These sets contain the (internal) variables of objects that will be read or written to during the execution of the instruction. For example, when an instruction A writes to a variable which is read by another instruction B, instruction A can undo some of the achievements of B and is therefore executed first. If no proper order can be found, Equate finds no solution. In this way, a set of solution programs is generated, which is offered to the application. The application must choose a solution program from that set to execute in order to satisfy the constraints.

Another system, which is quite similar to Equate, is the Object-Oriented Constraint System, OOCS [Hoole et al., 1994]. The main difference between these systems is that OOCS does not use term rewriting. Instead, an object supplies a set of *solution program segments* for each constraint that has been imposed upon it. The object guarantees that execution of any of these segments will leave the object in a state which satisfies the constraint. OOCS then solves a set of constraints by determining which program segment steps interfere with each other. This is achieved in the same way as in Equate by using read-sets and write-sets. By arranging the solution steps using these sets, the OOCS solver is able to decide which are feasible solutions.

Constraint solving with Equate takes the following four steps.

1. Rewrite all the constraints.
2. Order the partial solutions into solution programs.
3. Make a selection out of the set of solution programs.
4. Execute a program.

OOCS does not perform the first step. The last two have to be executed by the application program.

The problem of the above approaches is the local character of the solution. More powerful solutions are necessarily global in nature. The danger is that all objects need methods to get and set their internal data. This however, allows every other object to get and set these values which is clearly against the object-oriented philosophy.

One way to restrict this, is to have an object allow value setting only when its internal constraints remain satisfied (see [Rankin, 1991]). A constraint could be made internal by constructing a 'container object', which contains the constraint and the operand objects, but this does not solve the basic problem. In particular, the state of active objects cannot be changed without their explicit cooperation. (Active objects, or actors, conceptually have their own processor and behave autonomously, which is typical in animation and simulation.) Another approach is to limit access to private data to constraint-objects or the constraint solver-objects only. For example, C++ provides the 'friend' declaration to grant functions access to the private part of objects. This is also comparable to the approach taken by [Cournarie et al., 1995], where special variables (slots) are accessible by constraints only. One can argue that encapsulation is still violated (and specifically that the C++ friend construct could be easily misused).

3.3 Imperative and Declarative Programming

An imperative program is executed by performing an ordered sequence of statements. The statements operate on variables that contain values which can be updated by the statements. The set of values stored in the variables is called the state. The execution of an imperative program can be described in terms of a state machine model, where each statement causes the program to move from one state to the next. Control over the order of execution is provided by selection and iteration statements. The statements in the program are responsible for transforming an initial state into the final state.

The imperative paradigm closely resembles the actual machine that runs a program. Because of such closeness, imperative languages are familiar, efficient, and widespread. A disadvantage is that abstraction is more limited than with some other paradigms. Furthermore, function or procedure calls in an imperative program often have side effects, which can make it complex to understand, prove, or debug.

Object-oriented programming belongs to the family of imperative programming languages. It has the same notion of state that is operated on by sequentially executed statements. It extends the traditional imperative paradigm with techniques for building large, modular software systems.

Declarative programming entails the writing of declarations and logic-like rules among declarations. Typically, the rules describe that one set of declarations is true if some other set of declarations is true. There is no notion of a state that is sequentially manipulated by statements. The declarations specify one timeless state, the solution to the problem. Therefore, declarative languages are often called executable specification languages. Because declarative languages entail the representation of declarative models as compositions of logic-like rules, it is easier to relate such languages (and models) to formal logics The logic-like structures can be used to model a problem and construct correctness proofs thereof.

Constraint programming languages form a subclass of the declarative languages. Constraints declaratively specify restrictions on and relations between states (values of variables). In contrast to pure functional languages, where states are totally abandoned, there is a notion of states in constraint programming. However, constraints do not describe how states are manipulated or how restrictions or relations are satisfied. (This is the task of an underlying constraint solver.)

When combining imperative programming with declarative programming, in particular constraint programming, a conflict arises concerning the manipulation of states. Imperative updates of a state can break constraints (destructive assignments) that restrict that state. One could choose to let the update fail, thus violating the semantics of the imperative language, or allow the constraint to be broken, thus violating the semantics of the constraint language.

To find a general solution to this problem might be very difficult. However, in many cases it can be desirable to be able to change a state imperatively while constraints maintain relations on that state. For example, suppose a graphics object drawn on the screen may not cross the borders of the window in which it is displayed. It can be moved around and modified by an end-user. The operations that a user can perform are modeled as imperative actions on state of the object. The fact that the object may not step over the window borders is modeled by constraints. As soon as the user changes the state in such a way that constraints are violated, action is taken by the underlying system to solve the constraints. For example, the state of the object is set back to the values it had before the constraints were violated.

The possibility of specifying the relations in a declarative way can save an enormous number of lines of code that would be needed if the relations were updated in an imperative way. Once the constraints are specified, they are maintained ever after. In the above example, the restrictions that the object may not cross window borders can be specified by four constraints. One constraint specifies

that the object may not be positioned above the line that coincides with the upper border of the window and three other constraints specify similar restrictions for the three other window borders. If it were modeled in an imperative way, each time the user performs an action a series of *if* statements or a *case* statement would have to be coded to check whether the object crossed any borders. If so, the update of the object's state has to be coded too.

The functionality of these two approaches is exactly the same. However, besides the saving of lines of code, constraints provide a clearer specification of the relations that have to hold. This enables more complex structures to be built, which will be less error-prone, than when done in an imperative way.

When combining declarative constraints and imperative statements, it is important to describe the semantics of an assignment in relation to the constraints. A straightforward approach is to solve the constraints as soon as a constrained variable or object changes. However, in an object-oriented environment, one might want to have more control over triggering of solving. For example, when dragging a constrained object across the screen, a user might want the constraints to be solved after the dragging operation has been finished, but not during the dragging process itself. On the other hand, the user should not be forced to explicitly invoke the solver himself.

Furthermore, if an imperative assignment has been done that violates constraints a decision has to be made how to solve them. For example, a possible solution would be to restore all object states to the situation that existed before the assignment was done. However, in many cases, this solution might not be satisfactory.

3.4 Constraints vs. Information Hiding

In the glossary of [Booch, 1991], the following definition is given.

> Information hiding is the process of hiding all the details of an object that do not contribute to its essential characteristics. Typically, the structure of an object is hidden, as well as the implementation of its methods. The terms *information hiding* and *encapsulation* are usually interchangeable.

Information hiding, or encapsulation, is a key concept in object-oriented programming and design and serves to separate external, communication aspects of an object from the internal, implementation details of the object. This promotes modular design and prevents that small changes in some part of a system has massive ripple effects [Micallef, 1988]. However, information hiding obstructs the specification of relations in the OO paradigm [Blake et al., 1992]. In particular, constraints specify relations among the internal variables of objects. In [Aksit et al., 1992], a similar problem is identified in the observation that many object-oriented languages do not allow multiple views on objects. A solution is developed in [Aksit et al., 1993], where *object composition filters* are used to filter incoming messages with respect to conditions specified in the class definition. In our discussion, we will restrict ourselves to the encapsulation of variables inside an object, since this is of interest from the constraint solving viewpoint.

Access to the hidden variables of an object is provided through an interface of messages. Instead of directly reading or writing a variable, one object *requests* another object to perform some action by sending a message. This allows the receiving object to control assignments to variables, for example to maintain a class-invariant, or to update internal variables in order to maintain internal relations (or *internal constraints*).

We speak of the *violation* of information hiding when hidden variables of an object are directly accessed. That is, they are directly read, written, or referred to. For example, in `A.y = A.x + 2;`

where `A` is an object and `x` and `y` are hidden variables of `A`, information hiding is broken twice, once by directly reading `A.x` and once by doing a direct assignment to `A.y`.

Constraints in an object-oriented environment specify relations among objects that have to remain valid. It is the responsibility of a constraint solver to calculate values for the internal variables of the objects to validate the constraints. In an object-oriented context, these variables are usually hidden inside the objects and manipulation is only allowed via the interface of messages. This interface can obstruct powerful constraint solving. The following cases make it hard, if not impossible, for a solver to efficiently satisfy the constraints.

1. Execution of a method leads to violation of internal constraints of the object. As a result, the object can refuse to do the assignment or can update other variables (*side-effects*) in order to maintain the internal constraints.
2. Side-effects of the methods (possibly undocumented) can break previously satisfied constraints.
3. The provided interface does not provide enough facilities for the solver to put the object in a state that satisfies the constraints.

Let us look at an example to clarify these points.

3.4.1 Case Study

The objects under consideration are line segments. An object of class `LineSegment` has four internal variables of type `float`. The class definition in a C++ style is presented below.

```
class LineSegment
{
private:
    float x_begin, y_begin,
          x_end,   y_end;
public:
    void move_begin(float x, float y);
    void move_end(float x, float y);
    void rotate(float degrees);
}
```

The internal variables, defined in the `private` part of the class definition, are the hidden variables. The messages, specified in the `public` part, manipulate these variables and are the means by which other objects can communicate. The hidden variables of class `LineSegment` are not known outside the class.

Suppose, there are two line segments, l_1 and l_2. On these line segments, constraints are imposed in the form of linear equations. Consider the following constraints on the line segments.

$$
\begin{aligned}
C_1:&\quad l_1.\texttt{x_begin} \neq l_1.\texttt{x_end}\\
C_2:&\quad l_2.\texttt{x_begin} \neq l_2.\texttt{x_end}\\
C_3:&\quad l_1.\texttt{x_end} = l_2.\texttt{x_begin}\\
C_4:&\quad l_1.\texttt{y_end} = l_2.\texttt{y_begin}
\end{aligned}
$$

The constraints C_1 and C_2 specify that none of the line segments may be positioned vertically (the x-coordinates of their endpoints may not be equal to each other). The constraints C_3 and C_4 express the facts that endpoint `(x_end, y_end)` of l_1 must coincide with endpoint `(x_begin, y_begin)` of l_2.

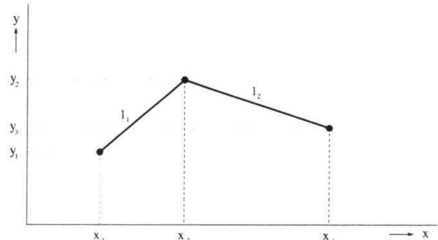


Figure 3.1: A solution to the constraint problem. The endpoints of the line segments should be positioned such that a solution to the constraint problem is obtained.

A possible solution is the following (for some x_1, x_2, x_3 ($x_1 \neq x_2, x_2 \neq x_3$), y_1, y_2, and y_3) (see also Figure 3.1).

```
Solution 1: l1.x_begin = x1;              (1)
            l1.y_begin = y1;              (2)
            l1.x_end   = x2;              (3)
            { C1 }
            l1.y_end   = y2;              (4)
            l2.x_begin = x2;              (5)
            { C1 ∧ C3 }
            l2.y_begin = y2;              (6)
            { C1 ∧ C3 ∧ C4 }
            l2.x_end   = x3;              (7)
            { C1 ∧ C3 ∧ C4 ∧ C2 }
            l2.y_end   = y3;              (8)
```

After execution of statements `(1)`, `(2)`, and `(3)`, constraint C_1 is satisfied. Statement `(5)` satisfies constraint C_3. Statement `(6)` satisfies C_4 and after statement `(7)` all constraints are solved. Between braces is indicated which constraints are valid after execution of an assignment.

`Solution 1` guarantees that the constraints are satisfied if all the assignments are carried out without interruption, that is, if the solution is executed atomically. However, it is clear that information hiding is violated. Firstly, the constraint specifications address the hidden variables of the line segments, which are not known outside class `LineSegment`. Secondly, when expressing the solution, direct assignments are done to hidden variables.

A way for stating the constraints and presenting the solution that would respect the information hiding principle could be the following.

```
C'1:  not_vertical(l1)
C'2:  not_vertical(l2)
C'3:  touch_eb(l1,l2)

Solution 2a: float deg, xb, yb, xe, ye;
             l1.rotate(deg);               (1)
             { C'1 }
             l2.move_begin(xb,yb);         (2)
             { C'1 ∧ C'3 }
             l2.move_end(xe,ye);           (3)
             { C'1 ∧ C'2 ∧ C'3 }
```

The constraints are stated as functions that take the constrained objects as arguments. The solution is presented as a set of messages back to the objects. Execution of statement (1) satisfies constraint C'_1. Statement (2) satisfies constraint C'_3 and statement (3) satisfies constraint C'_2. This approach does not violate the information hiding principle.

However, `Solution 2a` is not complete. In order for the solver to calculate values for the variables `deg`, x_b, y_b, x_e, and y_e, it has to know the current values of the endpoints of the line segments. But class `LineSegment` provides no messages to determine these values. In this case, the interface of the class does not provide enough facilities for the solver to satisfy the constraints.

Therefore, we extend class `LineSegment` to class `LineSegment'` with messages to retrieve the values of the endpoints and present a `Solution 2b` that satisfies the constraints (see below).

```
class LineSegment'
{
private:
    float x_begin, y_begin,
          x_end,   y_end;
public:
    void get_begin(float* x, float* y);
    void get_endfloat* x, float* y);
    void move_begin(float x, float y);
    void move_end(float x, float y);
    void rotate(float degrees);
}

C'1: not_vertical(l1)
C'2: not_vertical(l2)
C'3: touch_eb(l1,l2)

Solution 2b: float b1x, b1y, e1x, e1y,
                   b2x, b2y, e2x, e2y;
             float deg, valx, valy;
             // Initialization of deg, valx, and valy,
             // such that deg mod 180 ≠ 0 and valx ≠ 0
             l1.get_begin(&b1x,&b1y);              (1)
             l1.get_end(&e1x,&e1y);                (2)
             if (b1x == e1x)                       (3)
                l1.rotate(deg);
             { C'1 }
             l2.get_begin(&b2x,&b2y);              (4)
             l2.get_end(&e2x,&e2y);                (5)
             l2.move_begin(e1x-b2x,e1y-b2y);       (6)
             { C'1 ∧ C'3 }
             if (e1x == e2x)                       (7)
                l2.move_end(valx,valy);
             { C'1 ∧ C'2 ∧ C'3 }
```

Statements (1) and (2) retrieve the values of the endpoints of line segment l_1. Statement (3) satisfies constraint C'_1. Statements (4) and (5) retrieve the values of the endpoints of line segment l_2. Statement (6) satisfies constraint C'_3 and (7) satisfies constraint C'_2.

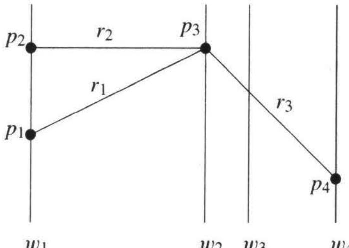


Figure 3.2: Example of the Linkage Positioning problem.

This solution would be correct if the methods were to be carried out unconditionally and without side-effects. However, suppose that class `LineSegment'` has an internal constraint which dictates that the length of a segment may not exceed some maximum value. When one endpoint is moved and the internal constraint gets violated, the object can either refuse to execute the move operation or modify the other endpoint to maintain the internal constraint.

In `Solution 2b`, if the object would refuse to perform the move when the maximum length is exceeded, it cannot be concluded after statement `(6)` that constraint c_3' is satisfied. Alternatively, if moving one endpoint would have the side-effect of a possible update of the other endpoint, executing statement `(7)` could falsify constraint c_3' that was previously solved by statement `(6)`.

Consequently, a solver would have to check, after calling a message, if the assignment has been carried out. If this is not the case, it has to try other messages to achieve the desired result. This complicates the task of a solver and, if an object refuses all attempts, it cannot satisfy the constraints.

Another drawback of `Solutions 2a` and `2b` is that assigning the values in this way is less efficient than in `Solution 1`. In `Solution 1`, direct assignments were done to achieve a solution. In `Solutions 2a` and `2b`, the object has to perform rotations and translations to arrive at the solution which is more computation intensive.

Related to this, the question rises whether the time complexity of the constraint solving algorithm changes if it has to take into account the way in which assignments can be done to variables. In [Veltkamp et al., 1995], a number of constraint satisfaction methods has been reviewed, their time complexities are determined, and a particular example is used to illustrate how the complexity changes due to the way in which the problem is modeled. The results of this work are presented in the next section.

3.5 Complexity of Constraint Algorithms

We use the following running example. Let $R = \{r_1, \ldots, r_m\}$ be a set of m rods. The set of endpoints of these rods is $P = \{p_1, \ldots, p_n\}$, a set of n points $p_i = (x_i, y_i)$ in $\mathbb{R}^2$. Each rod r_i has a fixed length d_i. The rods r_i are alternatively denoted as its pair of endpoints, $r_i = (p_{i_1}, p_{i_2})$, $i_1, i_2 \in \{1, \ldots, n\}$. The length of rod (p_{i_1}, p_{i_2}) is alternatively denoted as $d_{i_1 i_2}$. The rods form a single linkage structure, ℓ; if two rods are joined, then they share an endpoint. Let $W = \{w_1, \ldots, w_k\}$ be a set of k vertical walls (in $\mathbb{R}^2$); each w_i is represented by its x-coordinate. The rods must be positioned on the walls in such a way that the endpoints of each rod lie on different walls. See Figure 3.2 for an example.

This Linkage Positioning problem can be modeled as a constraint problem in various ways.

Constraint Problem 1 (CP-1)

The set of constraint variables is $V = \{p_1, \ldots, p_n\}$. The domain of each variable (x_i, y_i) is $W \times \mathbb{R}$. The constraints are a set of m binary constraints that fix the length between two points that belong to the same rod and prohibit these points to lie on one wall.

Constraint Problem 2 (CP-2)

The set of constraint variables is $V = \{r_1, \ldots, r_m\}$. Each variable is responsible for maintaining its internal integrity, that is, it retains a fixed length. The domain of each variable $r_i = (p_{i_1}, p_{i_2})$ is $(W \times \mathbb{R})^2$. The constraints are a set of m unary constraints that prevent each rod from being vertical, and q binary constraints to specify incident rods. The q binary constraints construct the linkage, and the value of q depends on the structure of the linkage, but is at least $m - 1$. If the maximal number of rods incident to a point is constant (independent of n or m) then $q = \mathcal{O}(n)$. If a constant number of points are incident to $\mathcal{O}(n)$ points then $q = \mathcal{O}(n^2)$. If $\mathcal{O}(n)$ vertices are incident to $\mathcal{O}(n)$ points then $q = \mathcal{O}(n^3)$.

Constraint Problem 3 (CP-3)

The constraint variable is the whole linkage ℓ. The domain of this variable is $(W \times \mathbb{R})^n$. The linkage should maintain the fixed lengths of the rods, and the incidence relations between the rods. As in CP-2, there are m unary constraints to prevent each rod from being vertical. These are unary constraints on the single linkage variable.

In the respective constraint problems, we let the amount of data that is hidden inside the objects increase. In CP-1, the coordinates of the points are hidden, in CP-2, the points themselves are hidden in the rods, and in CP-3, the rods are hidden in the linkage. In all three cases, we assume there are messages that can do assignments to the object as a whole. That is, in CP-1, assignments can be done to position the points, in CP-2, assignments position the rods, and in CP-3, assignments to the whole linkage can be done.

Already, it can be observed that by moving constraint variables inside an object, constraints on these variables disappear while new (different) constraints have to be created. Increasing encapsulation of variables in objects changes the constraint variables and, thus, creates a different constraint problem. This makes it less trivial to compare the time complexities of CP-1, CP-2, and CP-3. However, since the problems are still the same and they are derived from each other, it is worthwhile to investigate to what extent the time complexity is influenced.

Typically, the internal variables of the objects are the points, and they attain a value from $W \times \mathbb{R}$, or they are unspecified, that is, they have no value yet. Instead of restricting the domain of the x_i's to W in each of the three constraint problems above, we could let the domain be $\mathbb{R}$, and add extra unary constraints to place the points on the wall. This will be necessary for solving techniques that require continuous domains for the constraint variables, such as relaxation.

The Linkage Positioning Problem illustrates different types of constraints (unary, binary) and different types of domains (discrete, continuous). It is suitable to demonstrate both the alternation and the refinement models of constraint satisfaction.

The Linkage Positioning Problem is similar to a famous problem in computability and complexity theory, (k-)Graph Coloring. The problem there is to assign a color from a given set of k colors to each vertex of the graph, such that vertices connected by an edge have different colors. The set of vertices corresponds to P from the Linkage Problem, the set of edges to R, and the set of colors to W. The

condition of 'different colors' corresponds to 'different walls, plus fixed distance'; both conditions take constant time to check. Just like general constraint satisfaction, k-Graph Coloring is known to be NP-complete [Baase, 1978]. Determining if a graph is 2-colorable is easy (polynomial). Determining if it is 3-colorable is NP-complete. It is still NP-complete if the graphs are planar and the maximal degree is four. If the maximal degree is at most two, the k-coloring problem is easy. Coloring bipartite graphs (for example, trees) is also easy.

3.5.1 Backtracking

The first constraint satisfaction algorithm we consider is backtracking (BT). As we recall from Section 2.2.2, the time complexity of BT is $\mathcal{O}(ea^b)$, where b s the number of variables, a is the maximum domain size for a variable, and e is the number of constraints.

The backtracking algorithm assumes the domains of a variable to be discrete and finite. In Constraint Problem 1 (CP-1) of the Linkage Positioning Problem, the domain of each variable p_i ($= (x_i, y_i)$) is $W \times \mathbb{R}$. If there are no cycles in the linkage structure, the y-coordinates are of no significance. If the x-coordinates have values which do not violate the constraints, proper values can always be found for the y-coordinates in time linear in the number of variables. The complexity is in that case $\mathcal{O}(mk^n)$.

In CP-2, the set of constraint variables is $V = \{r_1, \ldots, r_m\}$. The domain of each variable $r_i = (p_{i_1}, p_{i_2})$ is $(W \times \mathbb{R})^2$. In order to be able to apply the backtracking algorithm, we restrict the domain to W^2 only, which has size k^2. The time complexity changes accordingly. In the case the linkage structure contains no cycles, there are k^{2m} candidate solutions. For every candidate solution, all constraints will be checked. The number of constraints is $m+q$, so the complexity of the algorithm is $\mathcal{O}((m+q)k^{2m})$.

In the case of CP-3, the single variable is a complete linkage. An assignment to this variable is a tuple of n walls. The size of its domain equals k^n, which is also the number of candidate solutions to the problem. For every candidate solution, all constraints have to be checked. The number of constraints is m, so the complexity of the algorithm is $\mathcal{O}(mk^n)$, which is the same as for CP-1.

3.5.2 Node and Arc Consistency

Node and arc consistency applies to the refinement model of satisfaction. See Section 2.2.2 for the pseudo code of the corresponding algorithms NC and AC. The time complexities for these algorithms are $\mathcal{O}(ae)$ and $\mathcal{O}(ea^3)$ (for AC-3), respectively, where a is the maximum domain size and e the number of constraints.

In CP-1, there are no unary constraints on the variables, thus, by definition it is node-consistent. Node-consistency for CP-2 is achieved by removing all wall-tuples from the domain of a rod r_i on which the segment cannot lie. These are the wall-tuples whose distance is larger than the rod length and the tuples which contain two the same walls. In CP-2 the domain size is k^2 and the number of unary constraints is m. This results is a time complexity of $\mathcal{O}(mk^2)$. The domain size in CP-3 equals k^n, and the number of unary constraints is m. This results is a time complexity of $\mathcal{O}(mk^n)$ for NC.

In CP-1, arc-consistency means that if we assign to a point p_i an arbitrary wall from its domain, then all other points to which it is adjacent, can be placed on walls so that the constraints still hold. The domain size in CP-1 is k, the number of constraints m, thus the complexity for AC-3 is $\mathcal{O}(mk^3)$. If we assume that there are no cycles in the linkage, finding an actual solution takes an additional $\mathcal{O}(n) = \mathcal{O}(m)$ amount of time (see Section 2.2.2).

In the case of CP-2, arc-consistency means that once a rod is placed on two walls, all adjacent

rods can also be placed. The difference with CP-1 is that the domain size has increased to k^2 and the number of constraints has become $(m+q)$. The complexity of algorithm AC-3 is now $\mathcal{O}((m+q)(k^2)^3)$. Again, if there are no cycles in the linkage, an actual solution can be found in $\mathcal{O}(n) = \mathcal{O}(m)$ additional time.

Since CP-3 only contains one variable which is the complete linkage, there are no binary constraints. So, if it is node-consistent, it is also arc-consistent.

3.5.3 Local Propagation

Propagation of known states propagates known values of variables through the network. It can apply to both the alternation model and the refinement model of constraint satisfaction. In the first case, values of variables are propagated. In the second case, sets of values or intervals are propagated [Davis, 1987].

The pseudo code of the LP algorithm that solves the linkage positioning problem is the alternation model which is given in Section 2.2.3. The time complexity of this algorithm is $\mathcal{O}(ae)$, for a maximum domain size a and e constraints.

In CP-1, the constraint variables are the n points. These are connected to each other by m binary constraints. The number of walls, which is also the size of the domain for a point, is k. If we apply LP to CP-1, the time complexity is $\mathcal{O}(km)$, assuming that there are no cycles. However, as is pointed out in Section 2.2.3, it cannot always find a solution. In constraint problem CP-2, the variables are the m rods. The domain size has increased to k^2, because every rod contains two points. The total number of constraints are the m unary constraints and the q binary constraints. The complexity for LP is then $\mathcal{O}(k^2(q+m))$. In CP-3, the constraint variable is the complete linkage, consisting of n points and m rods. Consequently, the domain size has increased to k^n, the number of unary constraints is m. The resulting complexity is $\mathcal{O}(k^n m)$.

3.5.4 Relaxation

Relaxation strictly applies to the alternation model of constraint satisfaction. The variables have infinite, continuous domains since relaxation uses mathematical functions for computing initial guesses and error estimates. The algorithm is given in Section 2.2.3. The time complexity is $\mathcal{O}(\lambda be)$, for b variables and e constraints, and where λ is the maximum number of iterations.

In order to be able to use the relaxation algorithm on the Linkage Positioning Problem, we allow the x-coordinates of the variables to be continuous and infinite. This means that the domain for points is $\mathbb{R}^2$.

For CP-1, the set of m binary constraints is now extended with n unary constraints, which state that the points must reside on walls. Error estimation is done by calculating distances between points and distances between points and walls. There are n variables and $m+n$ constraints, so the complexity for RX is $\mathcal{O}(n(m+n)\lambda)$. For CP-2, the number of variables is m and the number of constraints is increased by n unary constraints to constrain the endpoints of a rod to lie on walls. The total number of constraints is thus $n+q+m$. The complexity for RX is then $\mathcal{O}(m(q+m+n)\lambda)$. For CP-3, the number of constraints increases to $n+m$, resulting in a complexity of $\mathcal{O}((m+n)\lambda)$.

3.5.5 Equate and OOCS

Equate and OOCS are two systems that explicitly express the solution to a constraint problem as messages to the objects. The structure of the algorithms of these systems are outlined in Section 3.2.

For a detailed description, we refer to [Wilk, 1991] and [Hoole et al., 1994], respectively.

In [Choppy et al., 1989], an approach to determine the complexity of term rewriting systems is given. A notion of the complexity is given by means of the cost of terms. In [Leler, 1988], it is mentioned that if the set of terms is strictly left-sequential, there is a fast algorithm which can find a rule in the rule-base in time linear in the length of the expression to be rewritten (the head of the rule). Based on these observations, it is determined in [Veltkamp et al., 1995] that the time complexity of the rewrite rules in Equate and OOCS are cubic in the number of constraints ($\mathcal{O}(e^3)$, where e is the number of constraints).

We shall use Equate here to exemplify how the Linkage Positioning Problem can be solved using this strategy. The line of reasoning for OOCS remains the same. In order to use Equate, classes, constraint equations, and rewrite rules must be defined. The purpose of Equate is to generate a solution in the form of a sequence of method calls that satisfy constraints, rather than to generate and test individual values from the variables' domains.

In CP-1, the objects involved are points, their domain is the set of walls. It is the responsibility of the points to attain values in the domain. A typical rewrite rule could be,

```
d(point1, point2)=exp <--- fail_unless point1.wall(exp) ≠ NIL;
                           point2.move_to(point1.wall(exp))
```

where $point_1$`.wall(`exp`)` returns a position on a wall other than its current wall, at distance exp. In CP-1, the number of constraints is m, so the time complexity becomes $\mathcal{O}(e^3) = \mathcal{O}(m^3)$.

In CP-2 the objects are the rods. Considering the working of Equate, we choose to combine the unary and binary constraints. For example, a typical rewrite rule could be,

```
incident(rod1, rod2)=TRUE <--- fail_unless unary_constraint(rod1)=TRUE;
                               rod1.rotate(rod2)
```

where `unary_constraint` needs further rewriting and rod_1`.rotate(`rod_2`)` rotates rod_1 to be incident with rod_2. The number of constraints is now q. The time complexity $\mathcal{O}(e^3)$ thus becomes $\mathcal{O}(q^3)$.

In CP-3 there is only a single variable, the linkage structure. A typical method of this object is to rotate one part of the linkage around a hinge vertex, leaving the other part fixed. The object is responsible for maintaining internal constraints on those points that have a value. As in CP-1, the number of constraints is m, resulting in $\mathcal{O}(m^3)$ time complexity.

Equate need not always find a solution. Checking the read and write sets can be overly restrictive and may abort valid solutions. Another reason is that run time checks may fail. In CP-1 for example, in order to satisfy the distance constraint, Equate moves a point to only one of the possible solutions. This assignment satisfies the current constraint, but it is possible that it obstructs other variables to satisfy their constraints. Equate then succeeds in producing a solution program, but if the application executes the program, it fails, that is, one of the `fail_unless` statements evaluates to false and the program aborts.

3.5.6 Discussion

The constraint problems CP-1, CP-2, and CP-3 differ in the amount of variables that are hidden inside the objects which increases from CP-1 to CP-3. Consequently, hiding variables into objects changes the type and number of constraints. Therefore, Table 3.1 does not reflect a direct relation between information hiding and the complexity of the constraint algorithms. The table merely indicates that the way an application is modeled can largely influence the time complexity of constraint satisfaction.

	CP-1	CP-2	CP-3
BT	mk^n	$(m+q)k^{2m}$	mk^n
NC		mk^2	mk^n
AC-3	mk^3	$(m+q)(k^2)^3$	
LP	mk	$(m+q)k^2$	mk^n
RX	$n(m+n)\lambda$	$m(q+m+n)\lambda$	$(m+n)\lambda$
Equate/OOCS	m^3	q^3	m^3

Table 3.1: Time complexities for the constraint problems. m and q are the number of constraints, k is the maximum domain size, n is the number of variables, and λ is the number of iterations.

Particularly the number of constraints and the size of domains can vary. By encapsulating more variables into objects (like the vertices in the rods), constraints may move into the objects (like the fixed distances of the rods). On the other hand, new constraints may be needed to describe the relations among the objects (like the incidence constraints between the rods). Furthermore, if the domains of the object variables are discrete, the complexity of the solving algorithm is increased when the domains are enlarged.

In the alternation model of satisfaction, many algorithms do hard solution assignments to the objects (possibly in combination with domain refinement). For example, the backtracking algorithm BT by means of `v.assign(s)`, and the arc consistency algorithm AC-3 through `c.revise()`. The complexities above are valid on the assumption that the objects accept the assignment. However, to maintain any form of information hiding, the assignment should be done through message passing, encapsulating the internal implementation of the objects (as Equate and OOCS do).

Equate and OOCS obey information hiding to some extend. Neither the constraints nor the procedural solutions refer directly to an object's implementation. Actually, neither the solver nor the object determines a solution alone. The objects offer possible local solutions, and the solver tries to combine them into a global solution. However, a global solution need not be found, and the complexity is cubic in the number of constraints, which limits the use to small scale applications. Note that in the end the use of read-sets and write-sets in Equate and OOCS, which contain implementation specific knowledge of an object, infringes the concept of information hiding after all. In [Hoole et al., 1994] is put forward that encapsulation is maintained from the application programmer's perspective. The read-sets and write-sets are only available to the constraint solver, thus keeping the benefits of object-oriented programming for the programmer.

In the latter case the time complexity typically becomes exponential in a, or worse for continuous domains. The relaxation algorithms were formulated such that the objects themselves determine a value. However, this also gives them the freedom not to satisfy constraints, which destroys the compelling and declarative nature of constraints.

For the refinement model of satisfaction similar problems hold. One may model the domain as a separate object, but conceptually it is the exclusive property of the variable, and in fact a part of it. In that sense, changing the domain of a variable is equivalent to assigning a value.

3.6 Towards a Solution

From Sections 3.4.1 and 3.5, it shows that it is undesirable to restrict a solver's communication with objects to the (regular) interface that the objects provide. It could decrease the power of constraint solving. To avoid this, it is necessary to provide constraint solvers with direct access to the internal variables and, inevitably, violate information hiding. However, if this violation can be limited, it might still be acceptable.

In finding a solution, we want to preserve as much information hiding as possible. Although we allow a constraint solver to violate information hiding, it would still be desirable to maintain information hiding for all non-solver objects. For example, by restricting the interface that provides direct access to solvers only or by defining separate interfaces for solver objects and non-solver objects (see Figure 3.3).

Furthermore, even though a solver is granted direct access, we still would like to preserve the benefits of information hiding as much as possible also from the solver point of view. Basically, we can identify two goals that are realized by information hiding in object-oriented languages.

1. *Implementation independence.* Internal variables of an object are hidden behind an interface of messages, so that their implementation can be changed *independently* of other objects that use these variables.
2. *Internal consistency.* Internal variables of an object are operated on only by its messages, so that relations among these variables (either specified as *internal constraints* or as *class invariants*) can be maintained by the object itself.

Providing direct access for a constraint solver to internal variables can be done in two ways.

A. Declare the internal variables as directly accessible.
B. Provide messages to set and get internal variables.

By choosing option A., both benefits that are achieved by information hiding will be lost. If a solver accesses variables in this way, it becomes dependent on the object's implementation, while the object itself looses its control over assignments done to the variables.

By choosing option B., *implementation independence* can still be maintained. It could be achieved by providing *direct-access* messages, for example, `set_`*var*`()` and `get_`*var*`()` messages for each variable *var*, which would 'directly' assign and retrieve its value. It should be mentioned that in general, independent of implementation issues, exposing variables through set and get messages goes against the object-oriented philosophy. Technically speaking, information hiding is not violated since variables are accessed through their interface. However, it is often seen as an indication that a class is not well-designed, since the message interface should describe the behavior of the object independent of the internal representation. Although information hiding is technically not broken, the idea that lies behind it clearly is.

Whether each pair of `set_`*var*`()` and `get_`*var*`()` messages is implemented as an actual *var* variable is of the object's concern. Indeed, the implementation can change without having an effect on the objects that use the variable, as long as the messages behave *as if* they directly access the variable connected to them. That is, the returned value of an arbitrary `get_`*var*`()` message should be equal to the value that was used in a previously called `set_`*var*`()` message.

By providing direct-access messages, in particular the `set_`*var*`()` messages, class invariants or internal constraints can be violated. However, we cannot change a `set_`*var*`()` message to respect the invariants, since this would mean that it could ignore assignments or cause other variables to be

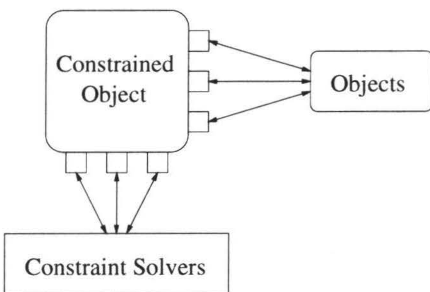


Figure 3.3: A constrained object has two separate interfaces, one for constraint solvers and another for all non-solver objects.

updated (*side-effects*) and thus loose its direct character. It is therefore not possible for the class to guarantee the validity of class invariants or internal constraints when direct access is provided.

A possible solution is to let the solvers that use the direct-access messages take these internal relations into account. This is quite possible since constraint solvers are specialized on 'relations'. For example, internal relations of objects can be specified as unary constraints on the objects that will be solved by the constraint solver.

Summarizing, if an end-user imposes constraints on his objects, it is desirable that the underlying constraint solver can efficiently solve them. But since information hiding can be an obstacle in achieving this, a constraint solver needs to have direct access to the object's variables. If we can restrict a direct-access interface to solvers only, a solution can be found that respects implementation independence and maintains internal consistency of the objects, provided that the solver will take the internal object relations into account.

3.7 Conclusion

If we design a system which combines objects and constraints, we are confronted with two incompatibilities. The first is the fact that object-oriented programming is imperative while constraint programming is declarative. Imperative actions of the object-oriented might violate constraints that are imposed on the objects or imperative actions can fail due to imposed constraints. Secondly, if an object has to maintain internal relationships and refuses an assignment, then either the constraints are not satisfied, or the solver has to negotiate with the objects to accept values. Obviously, to avoid a situation that an object can refuse a solution via a message, the methods should be designed so as to obey the internal relationships, or the solver must take into account these relationships. In the first case the satisfaction power of the solver may be limited, in the second case the information hiding principle is broken. Encapsulating the objects and completely hiding them from the constraint solver prevents the solver from doing any global solution. This leads to the following

> **Principal Observation** Under strict information hiding, constraint satisfaction on objects cannot be guaranteed.

If the constraint system is part of a programming language, the infringement of information hiding is under control of the constraint solver. From the application programmer's point of view the data

encapsulation is still preserved. As pointed out by [Freeman-Benson et al., 1992], requiring the language's internal constraint system to respect information hiding is similar to requiring an optimizing compiler to respect information hiding, which would make part of its task impossible.

If one is to sacrifice strict information hiding in order to facilitate constraint satisfaction, care should be taken not to allow abuse. Chapter 4 presents a radical separation of the constraint system and the normal object-oriented framework by means of two orthogonal communication strategies for objects, messages on the one hand, and events and data-flow on the other hand. In this way, the process of constraint management via data-flows does not interfere with the communication of the object-oriented world via messages. Because of the global and compelling nature of constraints, this strict separation facilitates the design and debugging of constraints and the constraint system.

Chapter 4

Conceptual Model

A conceptual model is an idealized description, used to communicate our understanding of the problem and to be a starting point for the design of some solution to it. In order to describe a solution to the incompatibilities of combining objects and constraints, it is important make clear the concepts that exist and specify their relations with each other.

In this chapter, the conceptual model is designed. It is the third step in the software design process that was stated in Chapter 1. In Section 4.1, the requirements that we impose on the model are described. Section 4.2 presents a top-level overview of the model and determines the elements should be present. These elements are described in more detail in Sections 4.3 through 4.7. Section 4.8 demonstrates the operational practice of the conceptual model by means of an example. Section 4.9 concludes the chapter.

4.1 Requirements

Before laying out the specific requirements for the constraint systems for the VR-DIS project and the GDP project, we first focus on combining objects and constraints. To combine constraints and objects when developing a computer graphics system, we define a conceptual model. This model should find solutions to the incompatibilities that are described in the previous chapter. Furthermore, it should enable design of an object-constraint system that is both modular and powerful. Modular means that the development of the object-oriented system should not interfere with development of the constraint system. This is done in order to prevent that the strengths of either the object-oriented paradigm or the constraint paradigm are sacrificed. Powerful means that constraint solving should be fast and efficient. Since the general constraint problem is NP-complete, this implies that we cannot implement a general constraint solver for solving all constraint problems. In practice, a specific type of constraint problem has a dedicated solver type that is designed to solve the problem efficiently.

We want to maintain the strict separation between constraint specification and constraint solving that exists in traditional constraint programming languages. That is, an application designer who designs objects and imposes constraints on them should not be concerned with how to solve the constraints or be bothered by specific solver details. By decoupling the usage of constraints and the development of constraint solvers, the solvers can be changed, upgraded, or replaced without affecting the application.

In a graphics system, many different types of constraints can exist (finite, infinite, numerical) at the same time that have to be solved fast. Consequently, there will be multiple dedicated solvers that can solve specific problems. To arrive at a powerful system, the solvers will have to cooperate to solve

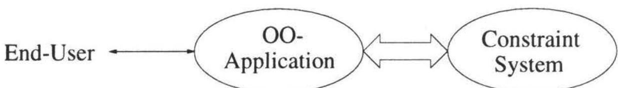


Figure 4.1: Top level overview of the conceptual model.

a complex constraint problem.

Finally, a model should provide a practical method for the development of a graphics system which incorporates constraints and objects. That is, it should provide a basis for a software engineer to build a working system.

Summarizing, we list the requirements that a model or methodology should fulfill that combines constraints and objects.

1. Deal with the incompatibility of imperative versus declarative. That is, define how the constraint world and object world interact.
2. Provide direct access for a constraint solver while information hiding is respected by all other objects.
3. Shield the constraint system off from the application designer who is using constraints to specify relations among his objects.
4. Allow multiple solvers to cooperate to solve a constraint problem.
5. Be a practical basis for designing systems that combine objects and constraints.

4.2 Overview of a Conceptual Model

In order to deal with the incompatibilities, the following main design decision is taken.

Design Decision 4.1
The main approach of the conceptual model is to rigorously separate the *constraint world* from the *object world*. The separation keeps both paradigms distinct and prevents that one of the two (or both) has to sacrifice any of its typical strengths. □

This approach will lead to better design, analysis, and implementation of complex computer graphics systems. Separating the two brings the advantage that an object-oriented system can be developed independently from the constraint system which makes it better to manage both of them.

Changes in the object system can be made without affecting the constraint system. Similarly, changes in the constraint system do not affect the object world. This implies that in the constraint system solvers can be replaced or improved, or satisfaction techniques can be changed independent of the object-oriented application.

Concerning the communication in the conceptual model, the following decision is taken.

Design Decision 4.2
The communication between the two worlds is managed by a protocol different from message passing, namely data flows and events. □

The choice for data flows and events is made because these communication primitives are completely orthogonal to message passing. In this way, communication between solvers and objects does not interfere with communication among objects.

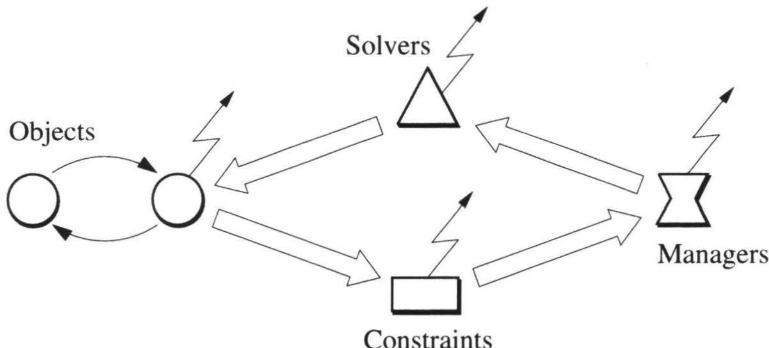


Figure 4.2: Objects communicate among each other via messages (curved arrows). The objects communicate with constraints and solvers via events (zigzag arrows) and data flows (big arrows). Constraints, managers, and solvers communicate with each other via events and data flows.

Events can be used to indicate state changes of the objects in the OO-application. Data flows can be used to provide access the internal state of an object for solvers and constraints, while direct access is not possible via ordinary message passing.

Figure 4.1 presents an top-level overview of the conceptual model. Seen from the viewpoint of the OO-application, there are two external system with which is communicated. On one side, there is an end-user who works with the application via a graphical interface. On the other side, there is the constraint system, which has to solve constraints that directly or indirectly are imposed by the end-user. The broad arrow represents the events and data flows by which the OO-application and the constraint system communicate. The constraint application is hidden from the end-user, but also from the developer of the object-oriented application.

There are two kinds of elements in the conceptual model:

1. The entities in a constraint problem,
2. Elements for describing communication between the entities.

The elements for describing the communication are the events and the data flows. The entities in a constraint problem are first of all the constraints themselves. Furthermore, there are the constraint variables which are the objects (of an object-oriented application). Next, there are solver entities that can solve the constraints. Finally, since there can be multiple solvers working on one constraint problem, we need entities that can manage the solving of such a problem. These entities are called the constraint managers.

Objects on which constraints are imposed are called *constrained* objects. Objects on which constraints *can be* imposed are called *constrainable* objects, or short, *constrainables*.

In Figure 4.2, an overview of the elements of the conceptual model is shown. The following elements are visible in the picture:

- (Constrainable) objects,
- Constraints,
- Solvers,
- Constraint managers,

- Messages,
- Events,
- Data flows.

The left part of the picture represents an object-oriented application, the right part is a constraint solving system. The (constrainable) objects exist in an object-oriented application. Constraints, managers, and solvers exist in the constraint system.

In the next sections, we will treat in more detail the different elements of the conceptual model.

4.3 Entities, Events and Data Flows

The basic building blocks of the conceptual model are *entity types*, *events*, and *data flows*. In this section, we will describe the features of these elements. The characteristics for data flows and events were inspired by the language MANIFOLD and its underlying concepts (see Section 6.1.1 and [Arbab, 1996]).

4.3.1 Entities

Entities are the basic components to model a constraint problem. An entity can exist independently and concurrently to other entities and it communicates with other entities via events and data flows. There are four different entity types in the conceptual model. These are constrainable, constraint, manager, and solver. In a system that is based on the conceptual model, there exist a number of instantiations of each type.

4.3.2 Events

An event is an asynchronous, non-decomposable (atomic) message, broadcasted by a constrainable, constraint, solver, or manager entity. *Raising* an event means that it is broadcasted into the environment. An entity that raises an event is called the *event source*. A raised event has a data structure holding the *name* (or *type*) of the event and an identifier of the source that raised it.

An event can be raised and detected by any entity (constrainable, constraint, solver, or manager). An entity can show its interest for the following cases:

- A certain event type raised by a specific entity,
- A certain event type raised by a certain entity type,
- A certain event type raised by any entity,
- Any event raised by a specific entity,
- Any event raised by a certain entity type,
- Any event raised by any entity.

An entity that raises an event will, in general, not know who are the receivers of the event. For example, an entity raises an event and continues with whatever it is doing without waiting for, or expecting, a reaction. However, entities that have knowledge of other entities in its environment may indicate which specific entities or entity types should be the receivers of a raised event. That is, events can be *targeted* to trigger a specific entity or entities. Targeted events are used when an entity desires that another entity performs some action. For example, one entity can raise a targeted event to trigger another entity to send data.

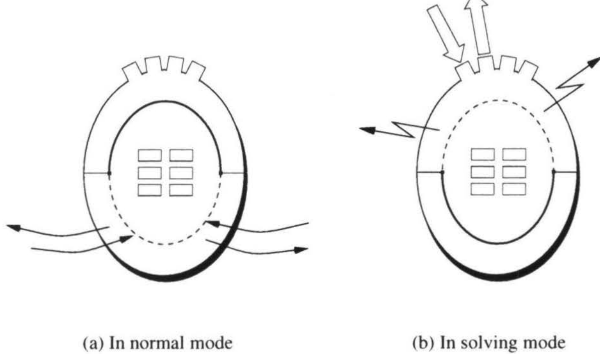


Figure 4.3: A constrainable communicates with other objects via message passing (a) and with entities of the conceptual model via events and data flows (b).

4.3.3 Data Flows

A data flow is the transport of units of data. This flow is always in one direction and has a beginning and an end. At the beginning and end of a data flow, there are entities. The entities have connection points through which data can be exchanged.

4.4 Constrainable

Objects on which constraints can be imposed have extra functionality to make them suitable for constraint solving. They must allow the following.

- Provide access to their internal data for the constraint solvers via data flows.
- Communicate with other objects via message passing.
- Disable message passing during constraint solving and disable data flows while not constraint solving.
- Detect events and raise an event when the state has changed.

Objects with this extra functionality are called *constrainable objects* or just *constrainables*.

A constrainable object communicates with its environment in two ways. The first way is via messages that it receives from and sends to other objects. In Figure 4.3(a), the curved arrows are messages sent or received. When constraints are being solved, the constrainable communicates with other entities through data flows, and message passing is suspended. This is illustrated in Figure 4.3(b), where the zigzag arrows depict events and the broad arrows are the data flows. The squares on the object boundary depict the ports to which channels can be connected. The rectangles inside the constrainable object represent its internal data.

The two communication strategies of a constrainable are mutually exclusive. When a constrainable communicates via events and data flows, it cannot at the same time send or receive messages. And equally for the other way around.

Design Decision 4.3
A constrainable has two *modes*, the *solving mode* and the *normal mode*. When the constrainable is in normal mode, it communicates with other objects via messages. When it is in solving mode, it communicates with other entities through data flows. □

The two modes and the two ways of communication of a constrainable object form the core of the separation between the object-oriented and the constraint worlds. Objects cannot communicate with other objects via events and data flows. Constraint solvers cannot communicate with objects via message passing.

4.5 Constraint

Constraints can be imposed on objects as a whole or on internal variables of objects. The domains of the variables are usually continuous and infinite, when they represent positions or dimensions of objects, but can also be discrete and finite. Constraints on continuous variables lead to numerical equations which may be linear, non-linear, or contain inequalities. Furthermore, constraints can be created and removed dynamically in an interactive application, which can easily lead to underconstrained or overconstrained situations.

A constraint specifies a relation among a number of constrainables. This relation has to remain valid at all times and limits the values that the internal variables of the constrainable can take. The constrainables on which the constraint is imposed are called the *operands* of that constraint.

Design Decision 4.4
A constraint is an entity, which means it has ports and communicates via events and data flows. □

Constraint entities have the following characteristics.

- A constraint is created in the OO-application.
- Constraints specify a relation among constrainables.
- They detect changes in constrainables on which they are imposed.
- Communication with constrainables takes place via events and data flows.
- They provide info for solvers and managers concerning the relation.
- Optionally, a constraint can provide additional functionality, such as checking its own validity, details on error size.

4.6 Solver

A constraint solver has to be able to deal with different situations. In a graphics application, exactly one solution has to be calculated, so that it can be displayed. This solution has to be calculated fast in order to meet performance requirements. Another difficulty is that a solver has to calculate a solution that an end-user does 'expect', which can be very hard to specify uniquely.

At this level, a solver is viewed as a black box that takes a set of constraints, together with a set of constrainables, as input. A solver is either a system that calculates values for the internal variables of the constrainables, or it simplifies the constraints. The first type of solvers calculates values for the constrainables that validate the constraints. The second type simplifies the constraints, which means they are transformed into other constraints that are possibly easier to solve. The new set of constraints is equivalent to the old one, that is, it describes the same constraint problem. This implies that, when the new set of constraints is solved, also the old set is solved.

If the solver cannot assign the proper values to the constrainables or cannot successfully transform the constraints, for example, because the constraints are conflicting, this is communicated to the entity that triggered the solver (which is a constraint manager). The triggering entity can then decide, for example, if solving has to be retried with different parameters, or if another solver should be chosen.

A solver provides the following functionality.

- Calculate values for a collection of constrainables to satisfy the set of constraints which the solver has to solve.
- Transform a set of constraints into a different set of constraints.
- Communicate states with constrainables via data flows.
- Give information concerning the result of the solving process to a manager.
- Allow the manager to control the solving process or to set solver options.

4.7 Constraint Manager

A constraint manager is an entity that deals with a series of tasks. They are listed below.

- Store and manage networks of constraints.
- Detect violated constraints.
- Analyze network of constraints.
- Communicate with solvers to solve parts of the network.
- Cooperate with other constraint managers to solve a network.
- Process information concerning the result of the solving process of a solver or another manager.
- Dealing with underconstrained, overconstrained, and failing situations.

A manager maintains a data structure in which it stores all constraints and their operands, the so-called *constraint network*. From this network, it can derive how constrainables and constraints are interrelated.

A constraint manager is concerned with conducting a number of constraint solvers or other managers. It is the entity that initially triggers the constraint solving process. Any particular manager can have a specific solving strategy to solve a collection of constraints. For example, there can be a manager which guides local propagation, a manager for relaxation, or one for solving numerical constraints. The task of the constraint manager is to select a number of constraints from the network and let it be solved by a specific solver or delegate them to another constraint manager.

Each network of constraints can be solved by multiple solvers or managers, but there is always exactly one main manager that supervises the complete network. This main manager determines when a solver or other manager should start solving constraints. In principle, for the main manager there is no difference between triggering a solver or another manager. If a solver is triggered, the manager expects the constraints to be solved or being replaced by a new set of simplified constraints. If another manager is triggered, the main manager delegates part of its work and expects an account of the end result.

4.8 Operational Description

In this section, an operational description is given of the conceptual model. First, we describe the general case. Next, the general algorithm is applied to a specific example.

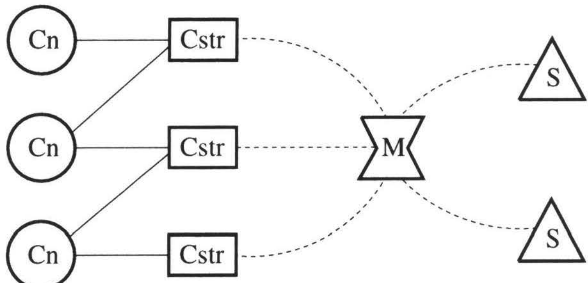


Figure 4.4: A graphic representation of a constraint problem. Constraints are depicted as rectangles, constrainables as circles, solvers as triangles, and the manager as a bow-tie. Constraints are imposed on the constrainables that are connected to it by lines. Managers handle a number of constraints and solvers that are connected to it by dotted lines.

The following algorithm describes point-by-point a general case in which there are several constrainables on which constraints are imposed, some solvers that can solve the constraints, and one manager (see also Figure 4.4).

1. A constrainable raises an event, E-changed, to indicate a change in its internal state, which is caused by message passing activity.
2. The E-changed event is detected by the constraints that are imposed on the constrainable.
3. The constraints that detected E-changed raise E-solving to put their operands in solving mode.
4. The constraints retrieve the internal states of the constrainables via data flows and check their validity.
5. Violated constraints raise the E-violation event.
6. Valid constraints raise E-normal to put the constrainables back to normal mode.
7. E-violation events are detected by the manager that manages the constraints, called the *main* manager.
8. The main manager puts all constrainables in solving mode by raising an event, E-solving.
9. The main manager determines which constraints have to be solved by which solvers and managers and sends identifications of these constraints via data flows.
10. The main manager triggers a solver or another manager, or several solvers or managers, to compute a solution for a set of constraints via the E-start event.
11. Triggered solvers request via events and receive via data flows the states of constrainables and start their computation.
12. Triggered managers start to execute their own solving strategy.
13. When a solver has finished its solving algorithm successfully and has calculated states for the constrainables that satisfy the constraints, it assigns these states to the constrainables via data flows.
14. When a triggered solver or manager is finished, it raises an event to notify the manager by raising an event E-finished.
15. The E-finished event is detected by the main manager.
16. Via data flows, the main manager retrieves additional information from the solvers concerning the exact results of the solving processes and, possibly, triggers other solvers or managers to

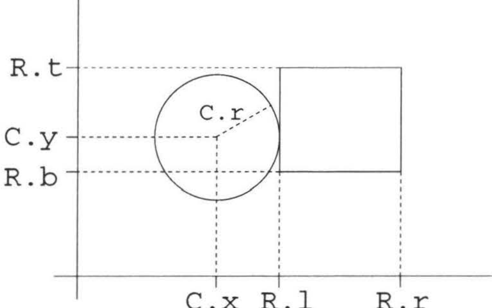


Figure 4.5: Circle C and rectangle R have two constraints which dictate that R and C have to touch and should have equal areas.

continue solving.

17. After all solvers have finished, the manager raises the E-normal event to put the constrainables back to normal operation.

We shall apply this general algorithm to the following example. Suppose we have a circle C and a rectangle R on which two constraints are imposed (see Figure 4.5).

One constraint, `EqualArea`, describes that both objects should have equal areas. The other constraint, `Touch`, specifies that the circle and rectangle have to touch. There are two solvers. One that can solve the `EqualArea` constraint and another that solves the `Touch` constraint. Both solvers calculate a solution by changing one of the objects. The `Touch` solver satisfies a constraint by *moving* one object. The `EqualArea` solver calculates a solution by *resizing* one object. Which object is changed and which one should remain unchanged is an option of the solver that can be set using parameters. If the option is not set, the solver communicates with the constraint it has to solve to find out which of the objects has changed most recently. It will then calculate a solution that does not change this object.

There is one manager that deals with both constraint types and solves the network using a local propagation algorithm. The manager knows both solver types and the effects of the solutions they calculate. In Figure 4.6, a graphical representation of the constraint problem is given.

Suppose the right side of rectangle R is moved by a message call. That is, the value of `R.r` is changed. The following events will then occur (each point corresponds to the same point in the previous algorithm).

1. Rectangle R raises the E-changed event due to the change in its internal state (the value of `R.r`).
2. The E-changed event is detected by both the `Touch` and the `EqualArea` constraints.
3. Both constraints raise the E-solving event to put their operands in solving mode.
4. The constraints raise events to retrieve the internal states of the rectangle and the circle via data flows. Next, they check their validity.
5. The `EqualArea` constraint is violated and raises the E-violation event.
6. `Touch` is not violated and raises the E-normal event to put the operands in normal mode (which does not happen, since `EqualArea` is still holding them in solving mode).
7. E-violation event is detected by the manager M.
8. The manager puts all constrainables in solving mode by raising the E-solving event.

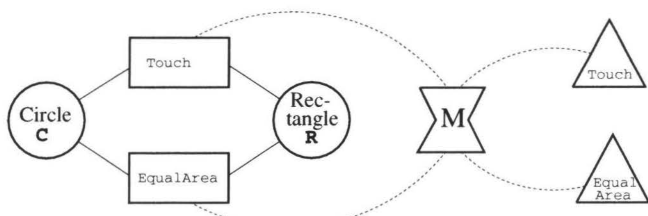


Figure 4.6: Circle `C` and rectangle `R` have two constraints which dictate that they have to touch and should have equal areas. Manager `M` handles the constraints which can be satisfied by the solvers.

9. The solver receives the identifiers of the `EqualArea` constraint, circle `C`, and rectangle `R` via data flows from the manager.
10. The manager triggers the `EqualArea` solver via the **E-start** event.
11. The solver receives via data flows the states of the circle and the rectangle.
12. The solver requests parameter data from the constraint which consists of an identifier of the object that changed (the rectangle). During solving, the solver takes care that only the state of the other constrainable (the circle) is changed.
13. The solver calculates a new value for circle by changing its radius `C.r` and assigns the new state to circle `C` via data flows.
14. Next, it raises the **E-finished** event.
15. The **E-finished** event is detected by the manager.
16. (a) Via data flows, the manager receives the result from the `EqualArea` solver and determines that the state of the circle has changed and that constraint `Touch` might be violated.
    (b) The manager triggers the `Touch` solver and summons it to calculate a solution by not changing the circle.
    (c) The `Touch` solver calculates a value for the position of the rectangle that satisfies the constraint and raises **E-finished**.
17. The manager receives the **E-finished** event, and determines that, since the `Touch` solver cannot violate any `EqualArea` constraints, all constraints are satisfied and that solving has finished. Finally, it raises the **E-normal** event to put the constrainables back to normal operation.

4.9 Conclusion

In this chapter, we presented a conceptual model for the combination of objects and constraints. There are four entity types and two means by which the entities communicate. The entities are the constrainables, constraints, solvers, and managers. They communicate via events and data flows. In Section 4.1, five requirements were defined that the model should fulfill. They are evaluated below.

1. The declarative aspect of constraint programming is coupled with the imperative aspect of OO programming by using events. Solving of constraints that are imposed on objects is triggered by changes in the internal state of an object. Such a change is indicated by raising an event.
2. Information hiding among objects is guaranteed by using data flows. Solvers have direct access while other objects communicate via messages. Since, data flows and events are orthogonal to message passing, the communication strategies do not interfere.

3. The data flows and events protocol shields the constraint solver off from the application designer.
4. The model introduces constraint managers to control multiple solvers. The managers take care of the administration of constraints and are responsible for the coordination of possibly multiple, concurrently operating constraint solvers. A solver computes values for the (internal variables of) constrainables that satisfy the constraints imposed on them.
5. The identification of entities and the way in which they communicate can be used to model constraint systems (see Chapters 6 and 7).

In the next chapter, the entities and the communication protocol among entities is worked out in detail.

Chapter 5

Specification

In this chapter, the events and data flows that exist among entities are described in detail. The model is called **CODE**, which is an acronym for Constraints on Objects via Data flows and Events. **CODE** is an elaboration of the requirements and characteristics that were determined in the previous chapter. Because all important design decisions were determined there, this chapter does not contain explicit design decisions.

In Section 5.1, the data flows and events that exist for each entity are described by a data flow diagram and tables. In Section 5.2, the modes that entities can be in are specified by means of so-called mode diagrams.

5.1 Data Flows and Events

From the entity descriptions and operational description in Section 4.1, we can derive which data flows have to exist among the entities. In Figure 5.1, all possible data flows are depicted. In the figure, the entities are represented by circles and the data flows by broad arrows. The data flows have names associated with them which indicate the type of the associated data and Roman numbers for easy identification.

The state of a constrainable can flow from the constrainable to a constraint (State Flow I) or to a solver (State Flow III). A solver can also send a new state to a constrainable (State Flow II). There is an entity flow from a manager to a solver (Entity Flow VIII). This flow contains identifiers of the constraints and constrainables that the solver has to solve. There is also an entity flow from the solver to the manager (Entity Flow IX). This flow contains constraint identifiers in case the solver does not compute states for constrainables, but instead, transforms the constraints into other constraints which are sent back to the manager. An example of this kind of solver is described in Section 7.4.2. Finally, there are parameter flows from constraints to solvers (Parameter Flow V), from constraints to managers (Flow IV), and among solvers and managers (Flows VI and VII). Parameter data is additional data needed for solving, such as numeric coefficients of a constraint, priority of a constraint, or control variables for steering a solver. Also, a manager can send entity identifiers to another manager (Flow XII) and they can communicate parameter data (Flows X and XI) among each other. In the figure, an extra manager is depicted to show these data flows.

The events that cause data to flow are listed in the event tables that are presented in this section. How the data flows are used is explained in the mode diagrams in Section 5.2. The tables describe which entities are sensitive to which events and how events and data flows are related. Like for the data flows, this information is extracted from the descriptions in Section 4.1.

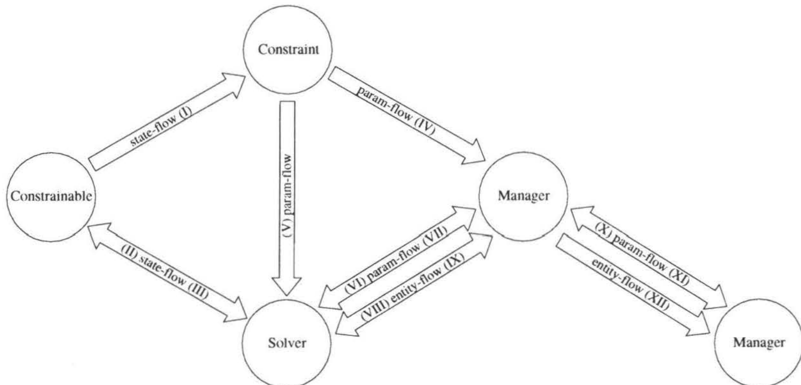


Figure 5.1: Data flows among the entity types.

A table presents the events that an entity can raise and detect. A table for an entity consists of two parts, *Raised events* and *Observed events*. The first part shows the names of the events that an entity can raise, the receivers of the event, and the purpose for raising the event. A raised event can be targeted, which is indicated by a superscripted "T" (targeted-event^{T}) The second part shows the names of the events that are observed by an entity, the source of the event, and the action taken upon detection of the event.

Table 5.1 shows an example of the events of an entity. An entity of type Entity-type-1 can raise an event with name E-event-raised. This event is detected by another entity of type Entity-type-2. The purpose for raising the event by Entity-type-1 is to trigger the receiving entity of type Entity-type-2 to perform some action. Entity Entity-type-1 is sensitive to one event type E-external-event. E-external-event is raised by an Entity-type-3. The action done by Entity-type-1 is, send data via a data flow. In the tables of the respective entities below, when there is a data flow constructed based on the detection of an event, there is also a reference (a Roman number) to the corresponding data flow in Figure 5.1.

5.1.1 Constrainable

Table 5.2 shows all events that can be raised and observed by a constrainable entity. For the entity names in this table, the following holds. With Constraint is meant a constraint that is imposed on the

Entity-type-1		
Raised events		
Name	*Receivers*	*Purpose*
E-raised-event	Entity-type-2	Trigger entity
Observed events		
Name	*Source*	*Action*
E-external-event	Entity-type-3	Send data

Table 5.1: Raised and observed events for an entity.

Constrainable		
Raised events		
Name	*Receivers*	*Purpose*
E-changed	Constraint	Notification of state change
Observed events		
Name	*Source*	*Action*
E-solving	Constraint, Manager	Switch to solving mode
E-normal-mode	Constraint, Manager	Switch to normal mode
E-state-dispatch	Solver	Receive new state (II)
E-state-request	Constraint, Solver	Send current state (I, III)

Table 5.2: Raised and observed events for constrainables.

constrainable. Solver means a solver that is calculating a new state for the constrainable. Manager means the manager that operates on the constraints which are imposed on the constrainable.

A constrainable raises the E-changed event to indicate that its state has changed due to a message call. The event is only detected by the constraints that are imposed on the constrainable.

A constrainable is sensitive to four event types. The E-solving and E-normal events can be raised by a constraint that is imposed on the constrainable or the manager that operates on the constraints. When the event E-solving is detected, the constrainable switches to solving mode and disables message passing. Upon detection of the E-normal event, the constrainable *can* switch back to normal mode again. The E-state-dispatch event is raised by a solver that has computed a new state for the constrainable. Upon detection, the constrainable receives a new state via data flows (State Flow II in Figure 5.1). Event E-state-request can be raised by a constraint that is imposed on the constrainable or a solver that needs the object's state. When the event is detected, the constrainable sends it state via data flows (Flows I and III in Figure 5.1).

5.1.2 Constraint

Table 5.3 shows all events that can be raised and observed by a constraint entity. For the entities in this table, the following holds. With Operand is meant a constrainable on which the constraint is imposed. Solver means a solver that is solving the constraint. Manager means the manager that operates on the constraint.

A constraint can raise events of four different types. The E-violation event is raised when the constraint is violated. It is detected by the manager that operates on the constraint. E-solving and E-normal are raised by the constraint to put its operands in solving mode and normal mode, respectively. These two events are only detected by the constrainables that are operands of the constraint that raised it. The E-state-request event is raised to retrieve the state of an operand via data flows (Flow I in Figure 5.1). This event is a *targeted* event and is only detected by the constrainable which was aimed at by the constraint. After it is raised, the constraint waits until it has received the state of the operand.

The constraint is sensitive to three event types. The E-changed event can be raised by constrainables. A constraint will detect the event if the constrainable that raised it is one of its operands. The E-param-request event can be raised by a solver that has to solve the constraint or the manager that operates on it. Upon detection, the constraint will send its parameters via data flows (Flows IV and V in Figure 5.1). Parametric data is data that gives specific details concerning the constraint. For example, parametric data for a `FixPosition` constraint could be the coordinates of the point that is

Constraint		
Raised events		
Name	*Receivers*	*Purpose*
E-violation	Manager	Notification of violation
E-solving^T	Operands	Put in solving mode
E-normal^T	Operands	Put in normal mode
E-state-request^T	Operand	Receive state data (I)
Observed events		
Name	*Source*	*Action*
E-changed	Operand	Check validity
E-param-request	Solver or Manager	Send parameter data (IV, V)
E-normal	Manager	Put Operands in normal mode

Table 5.3: Raised and observed events for constraints.

used to fix the position. The last event that can be detected is the E-normal event. When it is detected, the constraint puts its operands in normal mode by raising the E-normal event. The need for this event will be explained in Section 5.2.2.

5.1.3 Solver

Table 5.4 shows all events that can be raised and observed by a solver entity. For the entities in this table, the following holds. With Constrainable is meant an operand of a constraint the solver has to solve. Constraint is a constraint the solver has to solve. Manager means the manager that is managing the solver. A solver has always exactly one manager that guides it.

A solver can raise events of four event types. E-state-request and E-state-dispatch are raised to receive and send object states, respectively (State Flows III and II in Figure 5.1). These events are targeted to specific constrainables that should send or receive a state. After the solver has raised one of these events, it waits until it has received or sent the state of the constrainable. E-param-request is raised by the solver to request parameter data of a constraint. Also this event is targeted and causes the solver to wait until it has received the data. The E-finished event is raised by the solver to notify the manager that initially triggered the solver that its calculations have finished.

A solver is sensitive to five event types. The first is E-start. This event is raised by a manager and triggers the solver to start its solving algorithm. The E-entity-dispatch event is raised by the manager to send identifiers of the constraints that the solver has to solve (Flow VIII in Figure 5.1). If the solver transforms constraints, the manager uses E-entity-request to receive the identifiers of the newly created constraints (Entity Flow IX). E-param-dispatch is used by the manager to send parameter data to the solver (Parameter Flow VI). The parameter data received by the solver is data concerning specific details of constraints or parameters for the solving process. For example, the number of iterations for a relaxation solver or the starting point for local propagation. Event E-param-request is used by the manager to retrieve the outcome of the solving process (Parameter Flow VII). For example, the process ended successfully, the constraints are conflicting, or more solutions exist.

Solver		
Raised events		
Name	*Receivers*	*Purpose*
E-state-request^T	Constrainable	Receive state data (III)
E-state-dispatch^T	Constrainable	Send state data (II)
E-param-request^T	Constraint	Receive parameter data (V, VI)
E-finished	Manager	Notification of termination
Observed events		
Name	*Source*	*Action*
E-start	Manager	Start solving process
E-entity-dispatch	Manager	Receive entity identifiers (VIII)
E-entity-request	Manager	Send entity identifiers (IX)
E-param-dispatch	Manager	Receive parameter data (VI)
E-param-request	Manager	Send parameter data (VII)

Table 5.4: Raised and observed events for solvers.

5.1.4 Constraint Manager

Table 5.5 shows all events that can be raised and observed by a constraint manager entity. For the entities in this table, the following holds. With Constrainable is meant an operand of a constraint the manager operates on. Constraint is a constraint that the manager operates on. Solver means a solver that is managed by the manager. Manager means another manager that is cooperating with this manager.

A constraint manager can raise events of eight different types to communicate with other entities of **CODE**. A manager can raise the E-start event to trigger a specific solver, or another constraint manager, to start solving a set of constraints. This event is targeted to the specific solver of manager that is triggered. A manager raises the E-finished event if it finished its network solving algorithm and was triggered itself by another manager (see below). It raises the E-solving and E-normal events to put all constrainables of the constraints it operates on in solving and in normal mode, respectively. The receivers of the E-solving event are all constrainables that have constraints imposed on them which are being solved by the manager. The E-normal event is also received by the constraints themselves. The reason for this is explained in Section 5.2.2.

E-entity-dispatch is raised to send identifiers to a solver or another manager (Entity Flow VIII in Figure 5.1 for solvers, Entity Flow XII for managers). E-entity-request is raised to receive identifiers from a solver that has applied a transformation to a set of constraints (Entity Flow IX). E-param-dispatch is raised to send parameter data to a solver or manager that has to solve a set of constraints (Flows VI and XI for solvers and managers, respectively). E-param-request is raised to receive parameter data from a constraint or the results of the solving process from a solver or manager (Parameter Flows IV, VII, and X for constraints, solvers, and managers, respectively). All these four events are targeted events and, after it has raised one of them, the manager waits until it has received or can send the data that is under concern.

A manager is sensitive to six event types. The E-violation event is raised by a constraint that is operated on by the manager. Upon detection, the manager starts its solving algorithm. If it detects an E-start event, this is raised by another manager that can cooperate with this one. The E-finished event is raised by a solver or another manager that was previously triggered by this manager. When

Constraint manager		
Raised events		
Name	*Receivers*	*Purpose*
E-start^{T}	Solver or Manager	Trigger
E-finished	Solver or Manager	Notification of termination
E-solving^{T}	Constraints	Put in solving mode
E-normal^{T}	Constraints	Put in normal mode
E-entity-dispatch^{T}	Solver or Manager	Send entity identifiers (VIII, XII)
E-entity-request^{T}	Solver	Receive entity identifiers (IX)
E-param-dispatch^{T}	Solver or Manager	Send parameter data (VI, XI)
E-param-request^{T}	Constraint, Solver, Manager	Receive parameter data (IV, VII, X)
Observed events		
Name	*Source*	*Action*
E-violation	Constraint	Start solving strategy
E-start	Manager	Start solving strategy
E-finished	Solver of Manager	Evaluate result
E-entity-dispatch	Manager	Receive entity identifiers (XII)
E-param-dispatch	Manager	Receive parameter data (XI)
E-param-request	Manager	Send parameter data (X)

Table 5.5: Raised and observed events for constraint managers.

this event is detected, the manager will evaluate the results of the solver or manager entity and will determine if the overall solving process is finished or not. When the manager detects E-entity-dispatch, it will receive constraint identifiers from another manager which it will have to solve (Flow XII). When E-param-dispatch is detected, specific parameters for the solving process will be received (Flow XI). Upon detection of E-param-request, the manager will send the outcome of its solving process to another manager via data flows (Flow X).

5.2 Mode Diagrams

Each entity is described by means of a mode diagram. A mode diagram describes the modes an entity can be in and the transitions between the modes. During each transition and in each mode, an entity can perform actions, for example, raising an event. Transitions between modes are invoked by events and guarded by conditions.

In a diagram, modes are depicted as black dots. The mode that an entity starts in is encircled. The names of modes are written close to the black dots in small capitals. An optional rectangle under the name of a mode indicates the actions that the entity takes in that mode. An entity can take actions upon *Entry* of the mode, upon *Exit*, or *Throughout*. Actions specified under *Entry* are executed when the mode is entered via a transition. Actions specified under *Exit* are executed just before the mode is exited via a transition. Actions that are performed *Throughout* are executed continuously between entry and exit of the mode. When a transition to another mode can be made, for example, when an event occurs, the *Throughout* actions are interrupted and are continued upon re-entrance of the mode.

Mode transitions are depicted as curved arrows and are labeled by a string that indicates the event that causes the transition, under which conditions the transition can be done, and actions that are

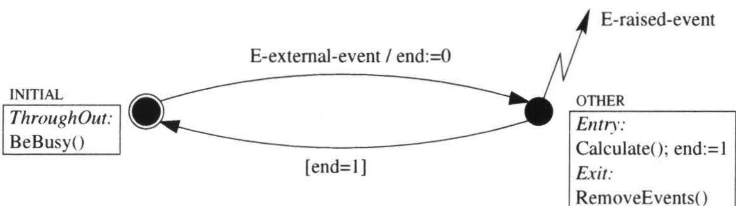


Figure 5.2: Mode diagram of an entity. The entity starts in the INITIAL mode. The transition to OTHER is caused by the event E-external-event. If, when in OTHER, the condition [end=1] becomes true, the entity performs the transition back to INITIAL.

performed during the transition. The format of this string is as follows,

Event [Condition] / Actions

Event is the event name that can trigger the transition. There can be zero or one events. If there is no event, then only the condition determines when the transition is done. The condition is written between brackets ([and]). After the slash (/), the actions are specified that are performed during transition from one mode to the other. If there are more than one actions, they are separated by semi-colons (;).

Events that arrive in a certain mode but do not trigger any transitions, either because there is no transition labeled with the event name or the associated condition is not true, remain in the event memory of the entity. An event is removed from event memory when it has successfully caused a transition or when it is explicitly removed by the entity itself.

All events that an entity raises are depicted in the mode diagram. When an event is raised during a transition, this is specified by the action `Raise(e_name)`. When an event is raised in a certain mode, this is represented by a zigzag arrow labeled with the event name. A zigzag arrow in a certain mode, thus, indicates that any function that is executed in that mode *can* (but not necessarily *has to*) raise the event represented by the arrow.

The notation for the mode diagrams in this section is based on the description of State Charts in [Harel, 1987]. State Charts provide a graphical notation for specifying state-transition diagrams. The mode diagrams have a somewhat different representation and there are additional symbols, such as the zigzag arrow for events. The name 'mode diagram' is used instead of 'state chart' or 'state transition diagram' in order to avoid a name clash between a *state*, or *mode*, that an entity can be in and the state which refers to the values of the variables of a constrainable.

In Figure 5.2, an example of a mode diagram is depicted. The entity in the figure has two modes, INITIAL and OTHER. While it is in INITIAL mode, it is executing the function `BeBusy()`. When the event **E-external-event** is detected, which is raised by another entity, execution of `BeBusy()` is suspended and a transition to the OTHER mode is made. During this transition, a variable, `end`, is set to zero. Upon entrance of the mode OTHER, the entity starts executing the function `Calculate()` and during this execution it might raise the event **E-raised-event**. After the function, it will execute the statement `end:=1`. This statement validates the guard `[end=1]` and enables the transition from OTHER to NORMAL. Before the transition is done, the function `RemoveEvents()` is executed which, in this example, removes all events from event memory. In INITIAL mode, the same process starts from the beginning.

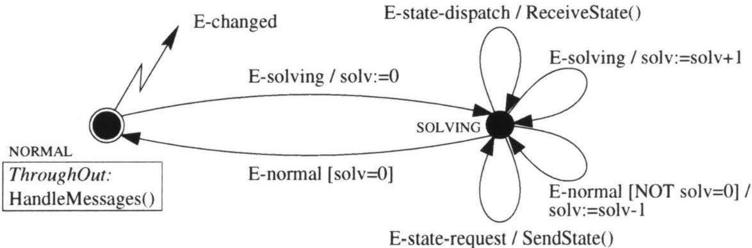


Figure 5.3: Mode diagram of a constrainable.

5.2.1 Constrainables

In Figure 5.3, the mode diagram for a constrainable is depicted. A constrainable has two modes, NORMAL and SOLVING mode. It starts in NORMAL in which it continually executes the function `HandleMessages()`. This function indicates that the entity is busy with communicating with other objects via message passing. When a change in the state has occurred, `HandleMessages()` raises the event E-changed to notify the entities in the environment.

Upon detection of the E-solving event, the constrainable will switch to SOLVING. This event can be raised by each constraint that operates on the constrainable and the constraint manager. A constrainable switches to SOLVING as soon as it detects this event, but it should not switch back to NORMAL until all constraints and the manager have raised the E-normal event. Therefore, the constrainable maintains a counter `solv` to count the number of E-solving events it has detected. During the transition from NORMAL to SOLVING, it sets this variable `solv` to 0.

In SOLVING mode, the constrainable is sensitive to four event types:

1. E-state-dispatch,
2. E-state-request,
3. E-solving,
4. E-normal.

Upon detection of E-state-dispatch, the object performs the transition that is labeled with this event name. During the transition, it executes the function `ReceiveState()` in which it receives new values for its internal variables via data flows. After the transition, it comes back into SOLVING mode. Similarly, when E-state-request is detected, the constrainable executes `SendState()` in which it exports the values of its variables via data flows.

When in SOLVING mode the E-solving event is detected, the constrainable increases the counter `solv` with 1. If the E-normal event is detected, there are two options depending on the value of the counter `solv`. If `solv` is not equal to zero, the constrainable performs the transition in which the counter is diminished by 1. If the counter is zero, the transition back to NORMAL is performed.

5.2.2 Constraints

In Figure 5.4, the mode diagram for a constraint is depicted. A constraint has two modes, NORMAL and CHECK. It starts in NORMAL mode. In this mode, it continually executes the function `HandleMessages()`. This implies that a constraint, provided it is represented as an object in the object-oriented application, can communicate with other objects via message passing. These messages can

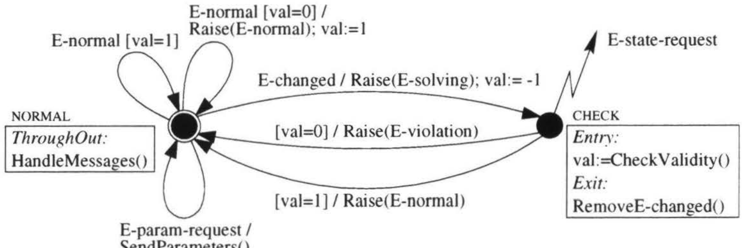


Figure 5.4: Mode diagram of a constraint.

be inquiries by the application concerning the validity, operands, or parameters of the constraint. However, when the constraint detects the **E-changed** event, which is raised by one of its operands, it suspends message passing and switches to CHECK mode. During the transition from NORMAL to CHECK, the constraint raises the **E-solving** event, to put all constrainables it operates on in solving mode and it sets a variable `val`, which is used to determine the transition to be taken after the CHECK mode, to -1.

Upon entrance of the CHECK mode, the constraint executes the function `CheckValidity()` in which it determines its own validity. The constraint retrieves the values of its operands by raising the event **E-state-request**. This event is aimed at the constrainables which are the operands of the constraint and will trigger these entities to send their states. The function `CheckValidity()` returns the value 1 if the constraint is valid and zero, otherwise. When the function is finished, the return value is assigned to the variable `val`.

When variable `val` becomes either zero or 1, the transition back to NORMAL mode can be made. Before the transition is performed, first the function `RemoveE-changed()` is executed. This function removes all **E-changed** events that arrived while the constraint was checking its validity. Removing these events prevents that the constraint will repeatedly check its validity if several of its operands changed at the same time.

If the constraint is violated, the transition labeled with condition `[val=0]` is selected. During this transition, the constraint raises the event **E-violation** to indicate to the environment that it is violated. If the constraint is valid, the transition labeled with condition `[val=1]` is selected. During this transition, the constraint raises the event **E-normal** to put the constrainables it operates on back in normal mode.

Besides the **E-changed** event, a constraint entity is sensitive to two other event types. These are **E-param-request** and **E-normal**. When **E-param-request** is detected, the constraint executes the function `SendParameters()` in which it sends parameter data via data flows.

In order for a constrainable to return to normal mode, all constraints that raised **E-solving** should also raise **E-normal**. However, a constraint that has raised **E-solving** when switching from NORMAL to CHECK does not always raise **E-normal** when it switches back from CHECK to NORMAL. In particular (see Figure 5.4), a constraint that is violated only raises **E-violated** when it switches back to NORMAL. Thus, its operands are still in solving mode.

This implies that a constraint which is violated still has to raise **E-normal** when the solving process has ended. This process is ended when the constraint manager raises the **E-normal** event. Therefore, a constraint is sensitive to the **E-normal** event raised by the constraint manager. When this event is detected and the constraint was violated, it executes the transition with event **E-normal** and condi-

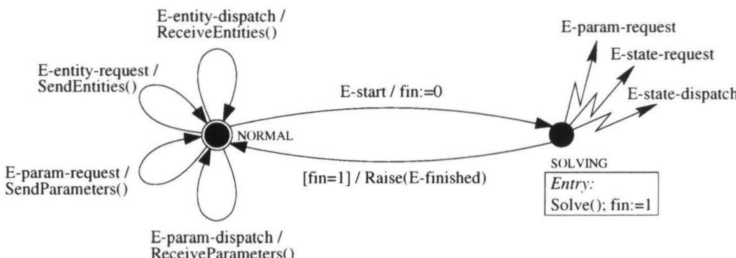


Figure 5.5: Mode diagram of a solver.

tion `[val=0]` and executes `Raise(E-normal)` to release the constrainables. Subsequently, it sets variable `val` to 1. If it was not violated, it performs the transition labeled with event **E-normal** and condition `[val=1]`. The transition removes the **E-normal** event from event memory without taking further action.

5.2.3 Solvers

In Figure 5.5, the mode diagram for a solver is depicted. A solver entity has two modes, NORMAL and SOLVING. The solver entity starts in NORMAL in which it is sensitive for the events **E-start**, **E-entity-dispatch**, **E-param-dispatch**, **E-entity-request**, and **E-param-request**.

The solver switches to SOLVING when the **E-start** event is detected. When the transition labeled with the event **E-entity-dispatch** is performed, the solver executes function `ReceiveEntities()` in which it receives identifiers of the constrainables and constraints it has to solve via data flows. When the transition labeled with event name **E-param-dispatch** is performed, the solver executes function `ReceiveParameters()` in which it receives parameter data via data flows.

During the transition from NORMAL to SOLVING, the solver sets a variable `fin`, used to indicate the end of the solving process, to zero. Upon entrance of SOLVING mode, the function `Solve()` is executed in which the solver calculates values for the constrainables such that the constraints are valid or translates the set of constraints it has to solve into another set of constraints. During this solving procedure, the solver can communicate with constraints and constrainables via events and data flows. The solver can raise three kinds of events:

1. **E-param-request**, to request parameters from a constraint,
2. **E-state-request**, to request the state of a constrainable,
3. **E-state-dispatch**, to send a new state to a constrainable.

After the function `Solve()` ends, the variable `fin` is set to 1, which enables the transition from SOLVING to NORMAL to be done. During this transition, the solver executes `Raise(E-finished)` to indicate to the manager that the solving procedure has ended.

There are two more events that a solver is sensitive to in NORMAL mode. These are **E-param-request** and **E-entity-request**. When **E-param-request** is detected, the function `SendParameters()` is executed which sends the outcome of the solving process via data flows to the constraint manager. If the solver transforms constraints into other (simpler) constraints, the constraint manager will ask for identifiers of these constraints by raising the **E-entity-request** event. Upon detection, the solver executes `SendEntities()` which sends the constraint identifiers.

5.2.4 Constraint Managers

In Figure 5.6, the mode diagram for a constraint manager is depicted. A manager has three modes:

1. NORMAL,
2. SOLVING,
3. EVALUATION.

It starts in NORMAL mode and, like constrainables and constraints, a manager can communicate with other objects and handles these messages in NORMAL mode in the function `HandleMessages()`. These can be messages for inquiry, but also to add to or remove constraints from the network of the constraint manager. In SOLVING and EVALUATION mode, it tries to solve a set of constraints. There is a variable `coop` which indicates if the manager is cooperating with another manager. It has initial value zero.

When the constraint manager detects **E-violation**, raised by a constraint, it suspends message passing and switches from NORMAL to SOLVING. During this transition the event **E-solving** is raised to put all constrainables in solving mode. Upon entry of SOLVING, the function `InvokeSolvers()` is executed in which the constraint network is analyzed in order to decide which constraints will be solved by which solvers and which part of the network is delegated to another constraint manager. In the function `InvokeSolvers()`, a manager can raise four kinds of events to trigger solvers and managers:

1. **E-start**,
2. **E-entity-dispatch**,
3. **E-param-dispatch**,
4. **E-param-request**.

These events are used to communicate with solvers and managers alike.

By raising **E-param-request**, it can request the constraint that raised the **E-violation** event for further details concerning the violation. For example, operands that have changed or the size of the error. By raising the events **E-entity-dispatch** and **E-param-dispatch**, the manager requests a solver or manager to receive identifiers of the constraints and parameter data, respectively. Next, it raises **E-start** to trigger the specific solver or manager.

When execution of `InvokeSolvers()` has finished, the manager waits until one of the triggered solvers or managers ends its solving process and raises the **E-finished** event. When the event is detected, the manager switches to EVALUATION. During the transition, it sets a variable `fin`, used for determining the next transition after EVALUATION, to a value unequal to zero or 1 (here, −1).

Upon entry of EVALUATION mode, the function `Evaluate()` is executed. In this function, the manager retrieves the results of the triggered entity. It can use the events **E-param-request** to request the outcome of the solving process of a solver or manager. In case a solver translated one set of constraints into another, **E-entity-request** is raised to request constraint identifiers. When all necessary data have been received, these are evaluated by the function. If solving has to continue, for example, because the solver did not solve all constraints, the function returns the value zero which is assigned to `fin`. If all constraints have been solved, the function returns 1.

When the variable `fin` receives value zero or 1, the transition out of EVALUATION can be made. Before the mode is exited and a transition is done, the function `RemoveE-violation` is executed which removes all **E-violation** events from event memory that were received during solving. This is done to prevent the manager from immediately switching to SOLVING after NORMAL, in case it had received multiple **E-violation** events from a number of constraints that simultaneously raised it.

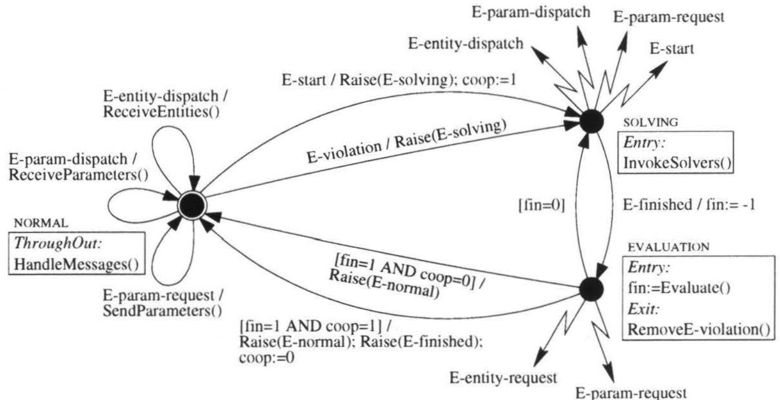


Figure 5.6: Mode diagram of a constraint manager.

If `fin` equals zero, the transition is done back to SOLVING and the solving process is continued. If `fin` equals 1, the transition is done to NORMAL (since variable `coop` was initially zero and hasn't changed). During this last transition, the manager raises the event E-normal to put all constrainables back to normal mode.

In NORMAL mode, a manager is also sensitive to four other events:

1. E-start,
2. E-entity-dispatch,
3. E-param-dispatch,
4. E-param-request.

All these events are raised by another manager that wants to use this manager for solving a set of constraints.

When the event E-entity-dispatch is detected, the manager executes the function `ReceiveEntities()` in which it receives identifiers of the constrainables and constraints it has to solve from another manager. When the event E-param-dispatch is detected, the manager executes function `ReceiveParameters()` in which it receives parameter data. The parameter data received is data concerning specific details of constraints or parameters for the solving process.

Upon detection of E-start, the manager is triggered to solve a set of constraints. When the event is detected, a transition is made to SOLVING. During the transition, E-solving is raised to put all constrainables in solving mode and variable `coop`, which indicates the manager is now communicating with another manager, is set to 1.

In SOLVING mode, the manager follows the same procedure as described above. That is, it switches back and forth several times between SOLVING and EVALUATION after which the variable `fin` becomes 1. As soon as `fin` becomes 1, the transition back to NORMAL is done via the transition labeled `[fin=1 AND coop=1]`. During the transition, E-normal is raised to put all constrainables back in normal mode, E-finished is raised to notify the cooperating manager of the termination of the solving process, and `coop` is set to zero.

In NORMAL mode, there is one more event the manager can detect, which is the E-param-request event. It is raised by the other manager to inquire for the result of the solving process. During the transition, the manager sends parameter data in `SendParameters()`.

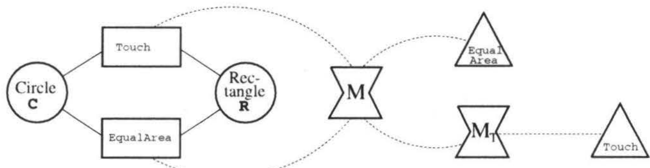


Figure 5.7: Circle C and rectangle R have two constraints which dictate that they have to touch and should have equal areas. Manager M cooperates with solver EqualArea and manager M_T to solve the constraints.

5.3 Operational Description

In this section, we will present a detailed operational description of the model that was specified in Section 5.2. The example that was used for the operational description in Section 4.8 is used with a slight extension. Instead of two solvers, the main manager M uses an EqualArea solver to solve EqualArea constraint and cooperates with another manager M_T to solve the Touch constraint. In Figure 5.7, the constraint network, together with the managers and solvers, is depicted.

Suppose, the right side of rectangle R is moved due to the execution of a message (see Figure 4.5). That is, the value of R.r is changed. Similar to the description in Section 4.8, we will examine the developing scenario point by point. Again, each point of the scenario has a corresponding point in the previous operational descriptions. When necessary, we refer to the mode diagrams of Section 5.2.

0. Initially, all entities start in their NORMAL modes, which are represented in the Figures 5.3 through 5.6 as black encircled dots.
1. Rectangle R raises the **E-changed** event due to the change in its internal state (the value of R.r). The raising of the event is represented by the zigzag arrow in Figure 5.3.
2. The **E-changed** event is (only) detected by the Touch and the EqualArea constraints (that is, not by the managers, solvers, or any other constraints). Both constraints suspend the actions they are doing in NORMAL mode (Figure 5.4) and make the transition to CHECK.
3. During the transition, the constraints raise the **E-solving** event to put their operands in solving mode (and initialize the value of val to 2). The two **E-solving** events (one raised by Touch, the other by EqualArea) are picked up by the circle and the rectangle. Both objects suspend the actions in NORMAL and use the first **E-solving** event they detect to switch to SOLVING. In SOLVING, the second **E-solving** event is detected. Upon detection, both objects perform the corresponding transition and increase the value of solv (which is initially zero) to 1.
4. Upon entrance of CHECK, the constraints start executing CheckValidity(). The constraints use the **E-state-request** event to retrieve the internal states of the rectangle and the circle via data flows. Since **E-state-request** is a targeted event, once raised, it will only be picked up by one object. For example, constraint Touch can request the state of circle C by raising the **E-state-request** event and indicating C as its target. Upon detection, C makes the corresponding transition and executes SendState() (State Flow I in Figure 5.1). If both constraints have received the states, they check their validity.
5. The EqualArea constraint is violated and, thus, its CheckValidity() function returns value

zero, which is assigned to the variable `val`. When `val` equals zero, the transition is made from CHECK to NORMAL via the arrow labeled `[val=0]`. During this transition, the constraint raises the **E-violation** event.

6. `Touch` is not violated and its `CheckValidity()` returns value 1, which is assigned to `val`. Consequently, the transition to NORMAL is done via the `[val=1]` transition. During this transition, `Touch` raises the **E-normal** event. The event is picked up by the circle and the rectangle and they perform the transition labeled `E-normal/[NOT solv=0]`. In this transition, variable `solv` is decreased to 0.
7. **E-violation** event is detected by the manager M. See Figure 5.6.
8. The **E-violation** event causes the manager to suspend the actions it is performing in NORMAL and to make the transition to SOLVING. During the transition, it raises **E-solving**. The event is picked up by the circle and the rectangle which perform the transition labeled `E-solving` starting in SOLVING. In this transition, variable `solv` is increased to 1.
9. Upon entrance of SOLVING, the manager starts executing the function `InvokeSolvers()`. It raises the targeted event **E-entity-dispatch** to send the identifier of the to-be-solved `EqualArea` constraint to the `EqualArea` solver. When solver `EqualArea` detects **E-entity-dispatch**, it executes the function `ReceiveEntities()` and receives the constraint identifier and the identifiers of circle C and rectangle R (via Entity Flow VIII, Figure 5.1) from the manager M.
10. Next, the manager raises **E-start** and the solver performs the transition from NORMAL to SOLVING. During this transition, the solver sets a variable `fin` to zero. See Figure 5.5.
11. Upon entrance of SOLVING, the solver starts executing `Solve()`. It raises the targeted event **E-state-request** to receive (via Data Flow III) the states of the circle and the rectangle. Upon detection, the circle and rectangle execute `SendState()`.
12. The solver requests parameter data from the constraint by raising the targeted event **E-param-request**. The requested parameter consists of an identifier of the object that changed. Upon detection, the constraint executes `SendParameters()` and sends the identifier of rectangle R (via Parameter Flow V).
13. The solver now calculates a new value for circle by changing its radius of the circle in such a way that C and R have equal areas. When succeeded, it raises the targeted event **E-state-dispatch**, aimed at the circle, and assigns the new state to C (Data Flow II). Upon detection of **E-state-dispatch**, the circle executes the function `ReceiveState()` and receives new values for its internal variables.
14. Next, the function `Solve()` of the solver finishes and value 1 is assigned to the variable `fin`. When `fin` becomes 1, the transition from SOLVING to NORMAL is done by the solver and, during the transition, the **E-finished** event is raised.
15. The **E-finished** event is detected by the manager M, which makes the transition from SOLVING to EVALUATION. During the transition, variable `fin` is initialized to 2.
16. Upon entrance of EVALUATION, the manager executes the function `Evaluate()`.

    (a) M raises **E-param-request** targeted at the `EqualArea` solver to receive (via Data Flow VII) the result of the solving process. The solver detects the event and executes `SendParameters()` and indicates that solving was successful and that C has changed. From this, M concludes that the `Touch` constraint might have become violated and, thus, that solving is not yet finished. After this, function `Evaluate()` returns the value zero which is assigned to variable `fin`.

    (b) The value of `fin` causes the transition back to SOLVING. Here, manager M starts executing `InvokeSolvers()`, again, and will now trigger manager M_T to solve the `Touch`

constraint. It raises **E-entity-dispatch** to send the identifier of the `Touch` constraint to M_T (Entity Flow XII) and **E-param-dispatch** to inform M_T that rectangle `R` has changed (Parameter Flow XI). Next, it raises **E-start** to activate the solving process of M_T.

(c) Upon detection of the events **E-entity-dispatch** and **E-param-dispatch**, M_T executes the functions `ReceiveEntities()` and `ReceiveParameters()`, respectively. When **E-start** is detected, M_T switches to SOLVING. During this transition, M_T raises **E-solving** and assigns value 1 to the variable `coop`. Next, it starts to execute the same algorithm as is described above for the manager M (using Data Flows VIII, VI, and VII to communicate with solver `Touch`).
During solving, circle `C` is moved by the `Touch` solver such that the `Touch` constraint is validated.
Finally, M_T arrives in EVALUATION where function `Evaluate()` returns value 1. When `fin` becomes 1, the transition back to NORMAL is done via the transition with guard `fin=1 AND coop=1`. During the transition, M_T raises **E-normal** to put the constrainables back in normal mode, raises **E-finished** to indicate it has finished the solving process, and sets variable `coop` back to zero.

17. Manager M receives the **E-finished** event and switches from SOLVING to EVALUATION where it executes `Evaluate()`. There, it raises **E-param-request** to request the result of the solving process from manager M_T (Data Flow X). M_T executes `SendParameters()` and indicates that circle `C` has moved. Based on this information, M determines that no `EqualArea` constraints were violated and that solving has finished. The function `Evaluate()` returns value 1 and the transition back to NORMAL is made via the transition labeled `[fin=1 AND coop=0]`. During the transition, M raises **E-normal**. This event is detected by the circle and the rectangle and they perform the transition labeled `E-normal/[NOT solv=0]`. In this transition, variable `solv` which has value 1 (see point 8, above) is decreased to 0. The **E-normal** event is also detected by the `Touch` and `EqualArea` constraints. For the `Touch` constraint holds that the value of variable `val` equals 1 (see point 6, above). Consequently, it performs the transition labeled `E-normal [val=1]` which does nothing besides removing the **E-normal** event from event memory. In the `EqualArea` constraint, `val` equals 0 (see point 5, above). Therefore, it performs transition `E-normal [val=0]`. During this transition, `EqualArea` raises **E-normal** (which it had not raised when it made the transition from CHECK to NORMAL, see point 5) and assigns value 1 to `val`. The event is detected by circle `C` and rectangle `R`. Both constrainables perform the transition labeled `E-normal [solv=0]` and switch back from SOLVING to NORMAL in which they continue the message passing activities with other objects.

5.4 Correctness

In the above sections, we have described the conceptual model, **CODE**, and also explained its operation. Although in the scenario listed above, the communication proceeds without problems, the question remains whether this is the case for all possible scenarios. That is, is the model is correct?

To verify these aspects of the system, formal methods need to be applied. In particular, model checking techniques are very appropriate to fulfill the task. Model checking is a formal verification technique for determining whether a given system satisfies a given specification [Clark et al., 1986]. A system is represented as a finite-state machine and specifications about the system are expressed as temporal logic formulas. For example, an important temporal requirement concerning constrainables could be the following, "A constrainable that has raised the **E-changed** event. should eventually return

to NORMAL mode".

In order to make the above specification of **CODE** suitable for model checking, several modifications should be made to the presentation of the protocol. First of all, the specification has to be simplified such that only the bare communication scheme remains without redundant details. For example, details like the variables in the mode diagrams and functions executed are not directly needed to check the communication protocol. Next, depending on the specific model checking technique that is used, the specification of **CODE** might have to be rewritten such that it can serve as input for the verification tool to be used. Finally, a set of temporal requirements has to be specified that guarantees absence of dead-lock, absence of life-lock, and continuous progress of all entities.

In this thesis, a complete formal verification of **CODE** is not given. The main reason is that the amount of work had to be balanced against the benefits of such a proof. The benefit might be difficult to determine since the model that is verified has gone through the modifications mentioned above and, moreover, an implementation will in general not be a precise implementation of **CODE**. Furthermore, in this research we decided to focus on the usability of **CODE**. It was one of the requirements of the conceptual model to develop a method for building systems that combine constraints and objects. Before proving such a method to be formally correct, it is first and foremost important to determine whether the method is workable. Therefore, instead of constructing a formal proof, a prototype system was built using **CODE** as the guideline.

5.5 Conclusion

In this chapter, we specified **CODE**, a conceptual model for combining constraints and objects. The elements that exists in the model are events, data flows, and the entities. The entities are constrainables, constraints, solvers, and managers. The characteristics of the entities are described and the way in which they interact. **CODE** is an elaboration of the requirements that were stated in Chapter 4.

What the model does not describe is how to apply it to build an actual application. This will be the subject of the next chapter, in which we present an application in which graphical objects can be subjected to constraints. The underlying system that deals with the constraints and the communication between the objects and the solvers is based on **CODE**.

Chapter 6

Prototype Implementation

In this chapter, we describe the first prototype implementation of the conceptual model, called PROTOM (contraction of Prototype Implementation). It is the fourth step in the software engineering trajectory that was presented in Chapter 1. The aim of this implementation is to build a system that shows all the concepts and elements of the previous chapter and enables us to verify them. Next to this, the aim is to determine which software components can be distinguished in an actual implementation.

PROTOM is built after the first version of **CODE** had been put on paper. As a result of the implementation process, the conceptual model evolved. The model presented in the previous chapter is the final version of this process.

In Section 6.1, the overall structure of the system and its components are presented. Several design decisions concerning this structure are explained. Section 6.2 treats the components of PROTOM and how the translation from the conceptual model to the implementation is done. Section 6.3 describes a constraint handling engine that provides several specific constraint types and solving techniques. In Section 6.4, a graphical application is shown that uses the constraint handling engine of Section 6.3. Finally, Section 6.5 concludes the chapter.

6.1 Overview of System Components

In order to verify the conceptual model, we build PROTOM to resemble **CODE** as closely as possible. That is, the elements that occur in PROTOM are directly mappable to the elements (entities, events, and data flows) in **CODE**.

The underlying system that provides the functionality to enable such an implementation is **MANIFOLD** (see [Arbab et al., 1993], [Arbab, 1998]), a language for managing interconnections among cooperating processes. The processes communicate via events and data flows. Using the language constructs of **MANIFOLD** (see Section 6.1.1), the communication primitives of **CODE** can be directly implemented. Furthermore, since the language is different from a programming language that implements the object-oriented application or constraint engine, it creates the physical separation between the application and the engine. For these two reasons, we choose **MANIFOLD** as a starting point in the design of PROTOM.

MANIFOLD enables us to implement the abstract conceptual model. To actually demonstrate this implementation, we need an interactive application that applies constraints on graphical objects and a constraint handling engine that solves the constraints. For this reason, an application, called Drawtool, is built that uses the constraints provided by a constraint handling engine, called the GA-CHE (see below).

In Figure 6.1, an overview of all systems is presented. The programming language MANIFOLD is based on the IWIM model (see Section 6.1.1). PROTOM is based on CODE. It is partly implemented in MANIFOLD to provide the conceptual functionality and partly in C++ to provide the interface to the outside world (see Section 6.2). Based on PROTOM, a constraint system, the GA-CHE (Graphics Application Constraint Handling Engine), is built that implements specific constraint types and solving methods. Next to the GA-CHE, there is the GA-CI (Graphics Application Class Interface) which provides the classes for an object-oriented application (see Section 6.3). In the application, Drawtool, graphical objects such as rectangles, circles, lines, can be modified. On the graphical objects, constraints can be imposed to specify relations (see Section 6.4).

The implementation language for implementing the object-oriented application is C++. C++ provides object-oriented concepts and it can cooperate easily with MANIFOLD. The constraint handling engine is also implemented in C++, for convenience. It could have been any other language (provided it can communicate with MANIFOLD).

Related to the components in Figure 6.1, we distinguish three programmers that build software components (see also Section 4.1).

1. The object-oriented application programmer. This programmer implements an object-oriented application in which constraints can be imposed on objects, in this case, the application is Drawtool. He has only access to the GA-CI which provides the classes for constrainable objects and constraint objects. He can create objects of these classes, create subclasses, and communicate with these objects via message passing. The application programmer cannot make direct modifications to the classes in the GA-CI.
2. The constraint handler programmer. This programmer creates a constraint engine which supplies constrainable and constraint types and solvers and managers that solve the constraints, here, the GA-CHE. He uses the functionality offered to him by PROTOM to implement the entities as described by CODE. Next to this, the constraint handler programmer creates the interface, the GA-CI, to provide the constrainable and constraint types to the object-oriented application. Which constrainable and constraint types are provided in the GA-CI depends on the types that are implemented in the GA-CHE.
3. The PROTOM system programmer. The PROTOM programmer builds the system that implements the basic functionality for the GA-CHE and the GA-CI to communicate with each other via events and data flows.

In subsequent sections, we will use these programmers to illustrate design decisions. We will refer to the types of programmers as the application engineer, the constraint engineer, and the PROTOM engineer, respectively.

The GA-CHE and the GA-CI are both developed by the constraint engineer. At first sight, it would be obvious to combine the two into one component. However, although the constrainable and constraint types in the GA-CI depend on the implementation of the GA-CHE, they are separate components. The GA-CI is an interface of classes to the application and, in this sense, is a part of the application. The GA-CHE is an engine that implements constraints and solvers. In accordance with CODE, the application and constraint handling engine have to be separated and should communicate via events and data flows. This separation and communication is carried out by PROTOM.

Before describing each component, we first introduce MANIFOLD.

6.1.1 Manifold

MANIFOLD is a coordination language for managing complex, dynamically changing interconnections

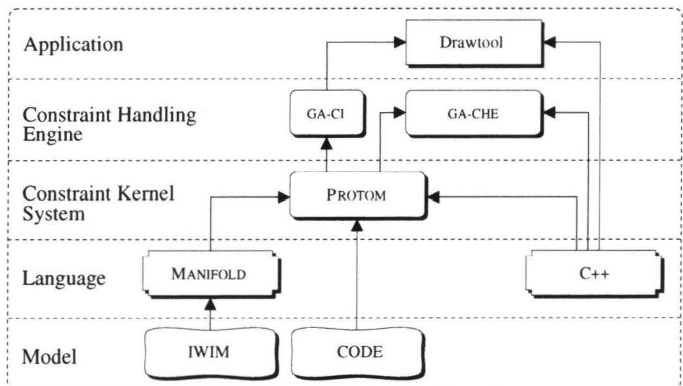


Figure 6.1: Systems overview. The language **MANIFOLD** is based on the IWIM model. The system PROTOM is based on the **CODE** model and is implemented in **MANIFOLD** and C++. The GA-CHE is a constraint handling engine. The object-oriented application, Drawtool, applies the constraints that are provided by the GA-CHE via the GA-CI.

among sets of independent, concurrent, cooperating processes [Arbab et al., 1993]. The language is based on the IWIM (Idealized Worker Idealized Manager) model of communication [Arbab, 1996]. This model describes a communication protocol which makes a distinction between *worker* processes, that is, processes that perform a computational task, and *manager* processes, that is, processes that manage communications. The IWIM model separates computation from communication concerns and establishes that no process is responsible for its own communication with other processes. It is always the task of a manager process to arrange necessary communication among a set of worker processes. Furthermore, a manager process may itself be considered as a worker process by another manager process, which allows hierarchies of communicating processes to be built.

The basic concepts in the IWIM model are *processes*, *events*, *ports*, and *channels*. A *process* is a black box with well-defined *ports* of connection through which it exchanges units of information with other processes in its environment. Interconnections between ports are made through *channels*, which are communication links that carry units of information.

Independent of the communication via channels, there is an *event* mechanism for information exchange in IWIM. There are *external* and *internal* events. External events are said to be *raised*, which means that they are broadcasted into the environment. External events can be picked up by any process, except by the process that raised the event (the *event source*). Internal events are said to be *posted*. They are not broadcasted, but are only detectable by the event source itself. Both internal and external events are typed. Contrary to **CODE**, IWIM has no concept of *targeted* events. Except in the case of internal events, a process never (directly) knows who are the receivers of the events it raises.

MANIFOLD is an implementation of the IWIM model and each of the basic concepts of process, event, port, channel in IWIM corresponds to an explicit language construct in **MANIFOLD**. Every process has an individual set of ports which serve as connection points for the channels (called *streams* in **MANIFOLD**). A process can raise events and can also decide for which event types it is sensitive.

A manager process, called a *manifold*, is always written in the **MANIFOLD** language. Worker processes can be written in any other language, such as C, C++, Fortran, etc. Corresponding to the IWIM

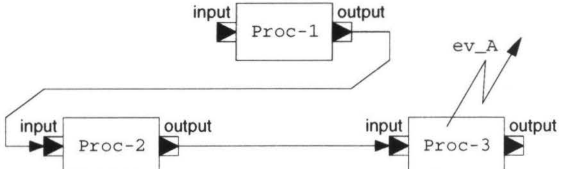


Figure 6.2: Example of **MANIFOLD** *processes*, *streams*, *ports* and *events*. Data which process `Proc-1` puts on its output port is redirected to the input port of `Proc-2`. Similarly, output of `Proc-2` is redirected to the input of `Proc-3`. Event `ev_A` which is raised by `proc-3` can be detected by any process that is sensitive to the event.

model, a manager process does not differentiate between worker processes and manager processes. In the remainder of this thesis, we will refer to (**MANIFOLD**'s) manager processes as "manifolds", to avoid name confusion with the (constraint) managers of the conceptual model. In Figure 6.2, a graphical representation is given of some arbitrary processes, streams, ports, and events.

6.2 Structure of PROTOM

In this section, we describe the design and implementation of PROTOM. PROTOM implements all functionality which is described in Chapter 4 as accurately as possible. Specific information concerning the types of constrainables, constraints, or solvers should not be present in PROTOM. It should only provide abstract entities that behave as described by **CODE**.

The following things should be present in the design of PROTOM:

1. The entities, events and data flows as described by **CODE**,
2. An interface to object-oriented applications,
3. An interface to systems that implements specific constraint and solver types.

These requirements lead to the structure of PROTOM as it is depicted in Figure 6.3. The three components are:

- PROTOM Kernel,
- OO-Application API (OO-API),
- Constraint Handler API (CH-API).

The PROTOM Kernel creates and removes processes that represent the conceptual entities and enables the communication among the entities. The OO-API is a library of classes that provides abstract constrainable and constraint types that can be used in an object-oriented application. It is the interface to the GA-CI in Figure 6.1. The CH-API encompasses an interface to a constraint solver or solvers that implement specific constraint and solver types. In Figure 6.1, it interfaces with the GA-CHE.

Below, we will treat each component of PROTOM in detail.

6.2.1 Kernel

The PROTOM Kernel is the part of the system that implements the communication protocol of **CODE**. It is built in **MANIFOLD**. This implies that data units flow via **MANIFOLD** streams as depicted in Figure 5.1,

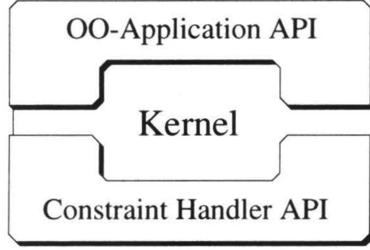


Figure 6.3: Structure of PROTOM. The OO-Application API (OO-API) is separated from the Constraint Handler API (CH-API) by the PROTOM Kernel.

and events are raised and detected as described in the tables of Section 5.1. The tasks of the Kernel are the following:

1. Process management, which entails creating and removing **MANIFOLD** processes,
2. Initializing event sensitivity of created processes, such that they can detect events that they should be sensitive to,
3. Data flow management, which entails creating and removing channels among processes.

Process Management

One task of the PROTOM Kernel is to create and remove **MANIFOLD** processes and let these processes communicate according to **CODE**.

Design Decision 6.1
A decision has to be made concerning the relation between **MANIFOLD** processes and **CODE** entities.

1. An option is to represent every entity in the conceptual model as a **MANIFOLD** process in the Kernel. The advantage is that it is possible to model **CODE** very accurately. The disadvantage is that it can lead to an exceedingly large number of processes and accompanying increasing burden for managing the overhead concerned with process creation, removal, and intercommunication.
2. The alternative is to combine entities in processes. This option would decrease overhead effort, but would obscure the precise event and data flow communication among entities.

In PROTOM, a middle way is chosen. Constrainable, constraint, manager entities are all represented by separate processes. That is, there are constrainable, constraint, and manager processes, respectively. This is done to be able to implement and verify all event and data flow communication among these entities. To reduce extra overhead, for the solver entities a less strict approach (that is, less strict according to **CODE**) is taken. □

Solvers that calculate a local solution for a constraint are combined with that constraint into one process. The resulting process is then equipped with event and data flows for constraint functionality and for solver functionality. For example, the process can detect E-changed events from the operands it is imposed on (thus, acting as a constraint entity) and it can be triggered by a constraint manager to calculate a local solution (thus, acting as a solver entity).

A solver entity that solves a collection of constraints (instead of calculating a local solution for only one constraint) is combined with a constraint manager entity into one process. The resulting

process has to be equipped with solver and with manager functionality. For example, the process is sensitive to the E-violation events raised by constraint entities (thus, acting as a manager entity) and it is able to send and receive states of constrainables via data flows (thus, acting as a solver entity).

Event Sensitivity

In PROTOM, not all events that are raised should be detected by all processes. Some processes should never detect events from certain other processes, while some processes should sometimes detect events from another process.

Design Decision 6.2
The sensitivity rules that specify which process type should be sensitive to which events of which other process type are specified in the event tables in Section 5.1.

1. One way to implement these sensitivity rules is to let all processes be sensitive to all event types and let each process find out for itself which events are of interest to it. However, if every process has to check all events that are raised, this results in needless system activity since most raised events are specifically aimed at another process or processes, but never at all processes together.
2. A better strategy is to let entities be selectively sensitive for events. **MANIFOLD** offers the functionality to make processes sensitive and insensitive to event types.

□

The sensitivity rules for the entity types do not vary during operation of an application. Therefore, it is convenient to have sensitivity initialized when a process is created. Since the PROTOM Kernel is the component that creates the processes it is also the most suitable place to initialize event sensitivity. The sensitivity rules for non-targeted events can be summarized as follows.

- A constrainable process is sensitive to the E-solving and E-normal events from the constraint processes that are imposed on it.
- A constraint process can detect the E-changed event from the constrainables it is imposed on, and the E-solving and E-normal events from the manager process that is concerned with solving that constraint.
- Finally, a manager process can detect E-violation events from the constraint processes it operates on.

In **MANIFOLD**, a newly created process is, by default, only sensitive to events of the process that created it (in this case, the PROTOM Kernel). A process can be made sensitive to the event of another process if it has a reference to this process. This means that by passing the proper process references around, the Kernel can make one process sensitive to events of another process. For the above listed non-targeted events, this means the following.

- The constrainable processes on which a constraint is imposed receive the reference to that constraint process.
- A constraint process receives the references to the constrainables on which it is imposed and receives a reference to the constraint manager.
- The constraint manager receives the reference to all constraint processes.

In this way, the non-targeted events can be initialized.

For the targeted events, a different approach has to be taken. Like IWIM, **MANIFOLD** does not have *targeted* events, while in **CODE** targeted events are used to trigger specific entities. However, the language does provide the distinction between internal and external events and has a mechanism called *guards*. A guard is a special process that is connected to a port and waits for some condition concerning this port to become true. When the condition becomes true, the guard posts an event in the event memory of the process that installed the guard. For example, a process `A` can install a guard on its `output` port, with condition `connected` and event `ev_output`. This means that, as soon as a connection is made to the output port of process `A`, the guard posts the specified event in the event memory of `A`.

The two mechanisms of internal events and guards can be used to simulate targeted events in PROTOM. For each targeted event that an entity has to be sensitive to in **CODE**, it has a port and a guard on that port in PROTOM. On the event raising side, the process that wants to *raise* a targeted event, raises this event and outputs the identifier of the target process via its standard output port. The raised event and the identifier are detected by the PROTOM Kernel who creates a stream to the targeted process.

For example, a constrainable has to be sensitive to the (targeted) **E-state-request** event from a constraint (see Section 5.2.1). Therefore, a constrainable has a port, called `p_state_out`, and a guard on this port that posts an event `ev_state_request` as soon as a connection is made. When the constraint requests the state from a specific constrainable, it raises the event **E-state-request** and sends the identifier of the constrainable through its output port. The event and identifier are picked up (only!) by the PROTOM Kernel who creates a stream between the `p_state_out` of the constrainable and the `p_state_in` of the constraint. When the connection is made to the `p_state_out` of the constrainable, the guard posts `ev_state_request` in its event memory. And, finally, when the constrainable detects this event, it sends the data the constraint requests.

Note that the constrainable does not know who is requesting the data. It only detects a connection to its output port and sends the data. This clearly demonstrates the strong separation between the constrainables and the solvers.

Data Flow Management

The PROTOM Kernel takes care of the data flows that exist among entities. The channels, called *streams* in **MANIFOLD**, that transport the data are connected to the *ports* of a process. Through these ports the processes send or receive data.

Design Decision 6.3
There are basically two strategies that can be followed for stream creation and removal.

1. The first is to create all streams such that all data flows that are depicted in Figure 5.1 are implemented.
2. The second option is to create a stream when a process requests it, and remove the stream again after data has flown through it.

The first is more efficient in CPU time, since streams do not have to be created and removed each time data is sent. Creating all streams in advance means that all processes are constantly connected and, thus, additional information has to be sent with the data to route it properly. In **MANIFOLD**, the number of ports that a process can have is static. The number of processes that a process communicates with is dynamic. Therefore, it will be necessary to connect multiple streams to a single port. If a process sends data through a port to which multiple streams are connected, this data is copied on all streams. Since the data in most cases will be intended for just one other process, this is a waste of resources.

The second option will be less efficient since stream creation and removal for each (little) part of data is time consuming. However, it has the advantage that streams are connected only between the two processes that communicate with each other. There is no routing information required, nor is data copied unnecessarily. Moreover, the stream creation can be used to implement targeted events as is described in the previous section. Therefore in PROTOM, this last alternative is chosen. □

The actual implementation of the part of the PROTOM Kernel that creates streams when a process raises a targeted event is done as follows. For every (constrainable, constraint, or manager) process, there is a separate manifold that takes care of stream creation requests for that process. If a process raises a targeted event to request or send data, this event is detected by its accompanying manifold. Based on the type of the event, the manifold determines which ports have to be connected. For example, if a process `P` raises the **E-state-request** event to get the state of a process `T`, its manifold, say `manif-P`, creates a stream between **P-state-in** port of process `P` and the **P-state-out** port of the targeted process `T`.

For each type of data that a process can send, it has an output port. For each type of data that a process can receive, it has an input port (see Figure 5.1). For example, a constrainable has an input and an output port for receiving and sending a state. A constraint has an input port to receive states and an output port to send parameter data. The alternative of having just one input port and one output port for all data for each process would require again extra information to identify the kind of data that is received or sent. Moreover, having just one port would disable a process sending or receiving data concurrently.

6.2.2 OO-Application API (OO-API)

The OO-API is the interface to an application that applies constraints. Since this application is object-oriented, the interface consists of a library of classes that can be used and inherited from within the application.

The OO-API should provide functionality to the GA-CI interface. At this level, we are not concerned with how or when the constraints are solved. The underlying system should satisfy and maintain the constraints that are specified. We need three types of classes:

1. Classes to create constrainable objects,
2. Classes that enable to impose constraints on those objects,
3. Classes for activation or deactivation of constraints.

Seen from the OO-API, the underlying constraint handling engine is invisible. It is therefore not needed to add more classes for creating solvers or managers.

In the OO-API, there are three C++ classes that enable the constraint engineer to build an interface for an application. These classes are:

1. `Constrainable`,
2. `Constraint`,
3. `MFHandler`.

These classes are abstract superclasses that provide necessary functionality for specific subclasses. The constraint engineer can subclass from `Constrainable` and `Constraint` to create constrainable and constraint classes, respectively. The class `MFHandler` is a dedicated class to handle communication with the PROTOM Kernel. The classes are explained below.

Class Constrainable

Class Constrainable provides the functionality for constrainable objects to the application. From Section 4.4, we can determine that this includes the following.

- A constrainable provides a *normal* mode in which it only communicates via message passing and a *solving* mode in which it only communicates via data flows.
- A constrainable raises an event when a state change has taken place.

When a new class is created for objects on which constraints will be imposed, this new class has to inherit from Constrainable. The new class, say Sub_Constrainable, has to provide information to the class Constrainable to ensure proper constraint solving. This entails identifying the above mentioned aspects.

1. The variables that constitute the state of the constrainable that can be subjected to constraints,
2. The messages that should be blocked when the constrainable is in solving mode,
3. An indication of when a state change is considered to have taken place.

In the OO-API, we choose these aspects to be identified by implementing methods of class Constrainable.

1. Methods of Sub_Constrainable that should be blocked during constraint solving must call a method, solving_mode(), of class Constrainable to check if the object is in solving mode. Based on the value of the method it can decide, whether it is executed or not.
2. The constraint engineer indicates the variables that constitute the state of an object of class Sub_Constrainable by implementing the methods frame_data() and unframe_data() of Constrainable. These methods, basically, pack and unpack the variables in order to be transported by **MANIFOLD**.

The third point, determining how to indicate a state change, will be considered separately below.

Indicating State Change

In Section 4.4, it was stated that a constrainable object indicates when its state has changed. In Chapter 5, it was determined that this is done by raising the E-changed event.

Design Decision 6.4
In PROTOM, it has to be decided who should initiate the raising of the E-changed event. There are three options.

1. The PROTOM engineer. The PROTOM engineer implements the abstract superclasses. At the level of the abstract superclass it can not be decided when exactly an event should be raised. For example, should an event be raised (and consequently solving be triggered) if one internal variable of the object changes or if more variables change? If an event is raised each time a single variable changes, solving might be triggered too often. For example, in the case where a vector is added to the position of an object and each time one coordinate of the position is updated also solving is triggered. In this example, one would like to treat the addition as one atomic action after which constraint solving is triggered. The PROTOM engineer only provides the facility to raise the event, but he cannot not decide when it has to be raised.
2. The constraint engineer. A better option would be to let the constraint engineer raise the event. This person implements constraint types and constrainable types and thus has knowledge about the states of constrainables.

3. The application engineer. It might still be the case that the event is raised to often. For example, consider the case where the application engineer has built a system to move graphical objects around. He furthermore has decided that constraints should not be solved while moving the objects, but only when a moved object has reached its end destination. If we let the constraint engineer decide to raise the event, constraint solving will also be triggered while moving the object.

Because we want to offer the application engineer the possibility to fine-tune the triggering of constraint solving, in PROTOM, we have chosen for the third option. It is implemented by providing a special message `commit()` to the application engineer. □

When an object has changed to such an extend that it might be necessary to solve the constraints, the application engineer calls the message `commit()` of that specific object. To preserve the state at the moment of committing, execution of the method locks the object and raises the E-changed event.

Constrainable Objects and Processes

The constrainable and constraint objects that are created in an application relate to the constrainable and constraint processes in PROTOM Kernel. They both represent the entities of **CODE** and should have full access to the object-oriented techniques, such as inheritance, message passing, information hiding, and should be able to communicate via events and data flows. In order to use object-oriented techniques, an entity should be implemented as an object. On the other hand, in order to have access to events and data flows, the entity should be implemented as a **MANIFOLD** process.

Design Decision 6.5
These two aspects lead a number of possible alternative implementations.

1. An entity can be implemented as a **MANIFOLD** process with some functionality for message passing. However, in that case, characteristics of object-oriented programming, such as inheritance, might be lost and the entity would not have full access to all object-oriented techniques. Since one of the strengths of **CODE** is that an application can be fully object-oriented, this option is not acceptable.
2. Alternatively, the entity could be implemented as an object with some **MANIFOLD** functionality. In that case, however, the entity could not take full advantage of functionality that **MANIFOLD** processes offer, such as being sensitive to a specific type of events. Since the aim of PROTOM is to be an *ideal* implementation, this alternative, too, is not satisfactory.
3. The adopted solution is to implement an entity both as an object in an application and as a process in the Kernel. In this way, an entity is a complete object, while it can also communicate via events and data flows. Furthermore, an application programmer is more explicitly shielded off from the constraint handling engine.

□

A consequence of having a constrainable entity represented as a C++ object and as a **MANIFOLD** process is that there are two states for one entity (one in the object and one in the process) that have to be kept up-to-date. That is, if a message call changes the state of the C++ object, the state of the **MANIFOLD** process has to be updated. And, if a solver changes the state of the process, the state of the object has to be updated. However, since message passing and data flow communication do not interfere with each other, the state update is only necessary when the entity switches its communication protocol.

A constrainable entity communicates via message passing when in normal mode and via events and data flows when in solving mode. Therefore, the state of the MANIFOLD process has to be updated when the constrainable switches to solving mode and the state of the C++ object has to be updated when the constrainable switches to normal mode.

Having a dual state for a constrainable has the advantage that while the constraint solver is modifying the state of the constrainable process, the state of the C++ object can serve as a back-up state in case the constraint solver cannot find a solution to the constraint problem.

Furthermore, the dual state can introduce some flexibility for an actual application. For example, consider the case where a constraint solver has to solve a large number of constraints on a large number of objects which can take a considerable amount of time. During solving, all constrained objects will be put in solving mode. This means that they are 'frozen' while the solver is active because communication via messages is not allowed. In the case of an interactive graphics program, if the solver would need, for example, 2 seconds to calculate a solution and, consequently, the objects are frozen for 2 seconds, this could have an enormous impact on the user interface of the program. Since the solver is calculating with a state different from the one that the C++ object has, one could still allow (some) messages of the constrained objects to be executed to prevent 'blocking' of the user interface. In that case, the user could be notified about the solver's activity, for example, by changing the color of the objects in solving mode.

Class Constraint

Class `Constraint` provides the functionality for imposing constraints on constrainable objects. This implies providing the following:

1. Communication with the Kernel for creating and removing the constraint processes,
2. Methods for inspecting characteristics of the constraint such as its arity or its operands.

Like a constrainable entity, an actual constraint entity consists of two parts; one part resides as an object in the C++ application and the other part is a MANIFOLD process in the PROTOM Kernel. The constraints in the application specify the relation among a set of constrainables. When a constraint is specified, the corresponding MANIFOLD processes are created in the constraint system. Next, the parameters of the constraint (its arity, operands) are communicated to the MANIFOLD process via data flows.

Once the constraint processes are set up in the Kernel, only these are used during constraint solving. That is, the constraint processes communicate via events and data flows with their constrainable operands. The constraint objects in the application are only an interface to the application and communicate via messages with (any) other objects. Consequently, a constraint object in the application does not detect state changes of its operands and it does not perform functions like checking its validity or calculating a local solution.

Specific constraint classes are implemented by the constraint engineer. Based on the implemented types in the constraint handling engine (the GA-CHE), he can provide the corresponding constraint classes in the GA-CI.

Class MFHandler

The final class is `MFHandler`, which is called this way because its main activity is handling the communication with MANIFOLD. That is, it implements all functionality that is needed for exchanging information with the Kernel. application engineer. It is used by the abstract superclasses `Constrainable` and `Constraint` to be able to communicate with the MANIFOLD processes.

In an application, there is one object of class `MFHandler` and objects of the classes `Constrainable` and `Constraint` have a pointer to this one object. When a constrainable object, or constraint object, needs to communicate with the Kernel, it calls messages of the `MFHandler` object to accomplish it. When in control, `MFHandler` object handles requests from **MANIFOLD** processes. When constraints are created or removed, this involves requesting the constraint object to send its type and operands. In other cases, it involves requesting constrainables to send or receive states.

The class `MFHandler` also acts as the part of the constraint manager that has to communicate with the application (see Section 4.7). For example, created constraints are registered to the `MFHandler` object which communicates them to the Kernel. But, like the constraint objects, there is no functionality within the class itself. All messages are forwarded to the manager entity in the Kernel which performs the actual operation.

6.2.3 Constraint Handler API (CH-API)

The CH-API is the interface to the constraint handling engine that implements specific constraint and solver types. Similar to the OO-API, the interface of the GA-CHE is offered in PROTOM, by means of abstract C++ superclasses. There is a class `CHE_Constrainable`, `CHE_Constraint`, `CHE_Solver`, and `CHE_Manager`.

Note that the classes `CHE_Constrainable` and `CHE_Constraint` are similar to but not the same as the classes `Constrainable` and `Constraint` in the OO-API. The classes of the GA-CHE interface implement functionality to communicate via events and data flows among entities. The classes of the OO-API implement functionality for the OO-application to communicate with the Kernel.

A constraint engineer that creates a constraint system, inherits from the abstract superclasses of the interface and implements several methods of these classes. For example, in subclasses of `CHE_Constraint` and `CHE_Solver` methods have to be implemented for packing and unpacking data that has to be communicated to other entities.

The classes are used by the PROTOM Kernel to create objects thereof. In each process that is created by the Kernel, an object of a GA-CHE interface class is created which is put inside the process.

6.3 Constraint Handling Engine

The constrainable and constraint types that are provided by the GA-CHE are circles, rectangles, points, and lines. Examples of constraint types are *touch* or *equal area* between circles and rectangles, *intersect* between points and lines, and *perpendicular* between lines. Via the GA-CI, these constrainables and constraints are made accessible for the object-oriented application.

In this section the solving techniques of the GA-CHE constraint handling engine are described. In Section 6.3.1, we first present a local propagation manager to solve an acyclic network of constraints. It illustrates the operation of one manager conducting a number of constraint solvers.

Next, in Section 6.3.2, a second manager is added to solve cyclic networks. The manager in the former section is then adapted to cooperate with the second manager to solve networks that may contain cycles.

6.3.1 Local Propagation Manager

The network that is formed by the objects and constraints is solved by a manager that uses a simple local propagation algorithm, similar to the one described in Section 2.2.3. This means that for every

constraint a local solution is calculated until every constraint in the network is validated. Each constraint has a local solver that can compute a solution for that constraint. A characteristic of the network is that it contains no cycles to prevent that the local propagation algorithm winds up executing infinite loops.

In PROTOM, a constraint and its local solver are combined into one **MANIFOLD** process. This means, that one identifier identifies both a constraint and its associated solver. The algorithm that the manager uses to trigger local solvers is given below.

```
 1. LP()
 2.   C     := ''constrainable that raised E-changed''
 3.   Cstrs := ''constraints on C''
 4.   while (Cstrs ≠ ∅) do
 5.     for (∀ Cr ∈ Cstrs)
 6.       ''Trigger local solver S of Cr''
 7.     rof
 8.     Cstrs_tmp := ∅
 9.     for (∀ Cr ∈ Cstrs)
10.       Cstns := ''constrainables that were changed
11.                 by solver S that solved Cr''
12.       for (∀ Cn ∈ Cstns)
13.         Cstrs_tmp := Cstrs_tmp + ''constraints on Cn''
14.         Cstrs_tmp := Cstrs_tmp - Cr
15.       rof
16.     rof
17.     Cstrs := Cstrs_tmp
18.   od
19. PL
```

The algorithm starts at the constrainable that first raised the E-changed event (line 2). The constraints on this constrainable are determined from the network (line 3). In the first propagation step, these constraints are solved (line 6). Next, the constrainables that were changed due to this step are determined (line 10). This information is requested from the solvers. Using the received constrainables, it is determined from the network which is the next set of constraints that have to be solved (line 13). Next, the constraint that was solved in the previous step is removed again, since it is not necessary to solve it again (line 14). This continues until there are no more constraints left to solve.

Since the network is acyclic, it can always be modeled as a tree. When taking the changed constrainable as the root of the tree, residing at level 0, the nodes on level 1 are the constraints on this constrainable. The nodes at level 2 are the constrainables that are the remaining operands (besides the operand in the root) of the constraints at level 1. At level 3, there are constraints again, and so on. The local propagation algorithm solves the constraints which reside at the odd numbered levels. Considering this algorithm, the tree is traversed in a breadth-first manner. That is, first all constraints on level 1 are solved, next, the constraints at level 3, etcetera.

An alternative to this strategy is to solve the tree in a depth-first manner. In that case, the algorithm can be written as a recursive algorithm and has the advantage that the system (that runs the software) can take care of some data storage. However, a depth-first approach forces the algorithm to solve all constraints sequentially. Constraints deeper down the tree cannot be solved before the previous constraints higher up the tree are solved (this is a property of local propagation) and the branches of the tree are dealt with one after the other. Taking a breadth-first approach, all constraints in one level can be solved in parallel. Since **MANIFOLD** processes run concurrently, this is the most appropriate solution.

A local solver computes one solution that satisfies the constraint. In case more solutions are possible (for example, a constraint that specifies that a circle and rectangle should touch, has an infinite number of solutions, even if one of the objects is fixed), the local solver picks one. In the GA-CHE, the solution chosen obeys the 'principle of least astonishment'. That is, the solver tries to find a solution that the user would expect. For example, in case of a touch constraint, the solver calculates the shortest vector between the two constrainables, and translates one of them along this vector.

If a solver calculates a solution which validates the constraint, the constrainables are updated by the solver and the identifiers of the changed operands are sent to the manager. If a local solver does not change any operands (either because it cannot find a solution, or the constraint is not violated), a notification is sent to manager. In that case, the manager will not propagate to constraints that come after the corresponding constraint.

The constraint manager has no knowledge of how any solver calculates a local solution for its constraint. Moreover, the constraint manager does not have any knowledge about the type of the constraints. The constraint network can contain any kind of constraints (which can be imposed on any kind of constrainables). As long as each constraint has a local solver and the network of the constraints is acyclic, the manager can successfully apply local propagation to solve the network.

The drawback of the simple local propagation manager is that it cannot handle cycles. In case there are cycles in the network, the manager can wind up in a loop, continuously triggering constraints around the cycle. In the GA-CHE, endless looping is prevented by marking each constraint that is solved. If a constraint is marked more than a certain number of times (currently, this number is arbitrarily set to 5), it will not be triggered again. However, this implies that the solving of the network possibly is ended before all constraints are solved. In the next section, cooperating managers are introduced to solve a network that contains cyclic constraints.

6.3.2 Cooperating Managers

An extension of the previous implementation demonstrates the cooperation of two managers. The two managers are the local propagation manager and a *cycle* manager. The first manager tries to solve the network by local propagation, as described in Section 6.3.1. The second manager applies a numerical method to find a solution to a set of numerical equations which are extracted from constraints in a cycle. The local propagation manager can solve networks that do not contain any cycles and, therefore, can be ordered as a tree. The cycle manager can solve cycles and, more general, other non-tree-like structures.

If we have a network that can be structured as a tree apart from some subparts that contain non-trees, we would like to have one manager that uses local propagation to solve the tree and one manager that solves the non-trees. In this way, large networks in which the non-tree-like structures are relatively small can be solved efficiently. A local propagation algorithm traverses the network only once, while a cycle solver treats a cycle, for example, by traversing it multiple times or solve all constraints in a cycle at once (see [Sannella, 1994] for an overview of cycle solvers). In the remainder of this text, we will use the word cycle to indicate cycles as well as any other non-tree-like structures.

This is the strategy that is taken in the GA-CHE. The local propagation manager is the *main* manager. It maintains the complete constraint network and initially starts the solving algorithm when constraints get violated. The cycle manager is triggered by the local propagation manager to solve constraints in a cycle.

The problem remains how to detect a cycle. One strategy is to traverse the network until a cycle is encountered which, for example, could be detected by marking the solved constraints, and let the encountered cycle be solved by a cycle manager. However, this will be inefficient since the constraints

are solved twice. First, by the local propagation manager and, next, by the cycle manager. Moreover, if cycles within cycles occur, this strategy has to be enhanced to be able to determine the constraints that have to sent to the cycle manager.

In order to detect cycles as early as possible, a better strategy is to determine them before starting local propagation. If, in that case, the local propagation algorithm encounters a constraint that is marked as being in a cycle, all constraints in that cycle can be collected and sent to the cycle manager.

The local propagation algorithm is very similar to the one described in Section 6.3.1. It only differs in two points.

1. Before the actual algorithm is started, first, a pre-processing step is performed in which trees and cycles are determined.
2. If, while propagating through the network, a constraint in a cycle is encountered, all constraints in that cycle are collected and are sent to the cycle manager.

To detect whether a constraint is in a tree or not, the local propagation manager does a pre-processing step before starting the actual local propagation algorithm. In this pre-processing step, the constraint network is analyzed and for each constraint is indicated whether it is part of a tree or part of a cycle.

Before discussing the network analysis algorithm, we first present the adapted local propagation algorithm. The lines that are different from the algorithm in Section 6.3.1 are prefixed by an asterisk (*).

```
   1. LP_cycl()
*  2.   analyse_network()
   3.   C     := ''constrainable that raised E-changed''
   4.   Cstrs := ''constraints on C''
   5.   while (Cstrs ≠ ∅) do
   6.     for (∀ Cr ∈ Cstrs)
*  7.       if (''Cr is in a cycle'') then
*  8.         Crs := ''constraints in cycle of Cr''
*  9.         ''Trigger other manager M to solve Crs''
* 10.       else
  11.         ''Trigger local solver S of Cr''
  12.       fi
  13.     rof
  14.     Cstrs_tmp := ∅
  15.     for (∀ Cr ∈ Cstrs)
  16.       Cstns := ''constrainables that were changed
* 17.                 by entity (S or M) that solved Cr''
  18.       for (∀ Cn ∈ Cstns)
  19.         Cstrs_tmp := Cstrs_tmp + ''constraints on Cn''
  20.         Cstrs_tmp := Cstrs_tmp - Cr
  21.       rof
  22.     rof
  23.     Cstrs := Cstrs_tmp
  24.   od
  25. lcyc_PL
```

In line 2, `analyse_network()` is called to mark which constraints in the network are in a cycle and which ones are in a tree. This information is later used in line 7 to check if the constraint is in a cycle. If a constraint is in a cycle, all constraints in that cycle are collected (line 8) and then sent to the numerical manager (line 9).

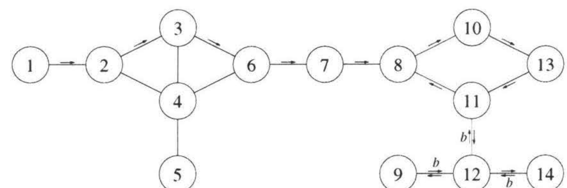


Figure 6.4: A network of nodes. Nodes 1, 5, 7, 9, 12, and 14 are tree nodes. Nodes 2, 3, 4, 6 are in one cycle. Nodes 8, 10, 11, 13 are in another cycle. The arrows next to the edges indicate an arbitrary, incomplete traversal path starting at node 1. A *b* next an arrow indicates a backtracking step.

In the `for` loop of line 6, an `if` statement is added (line 7) to determine if a collection of constraints has to be solved by a manager or one constraint has to be solved by a local solver. Note that, in the `for` loop of line 15, this distinction is not made. The algorithm requests all triggered entities to send the constrainables they changed. Since the interface for solvers and managers is the same for this type of information, it is not necessary to make the distinction between a solver or a manager entity.

Network Analysis

We want to analyze the network as a preprocessing stage prior to actually solving the network. This stage identifies the constraints that can be solved by the local propagation manager and find the constraints that have to be solved by the cycle manager. This means that we are looking for constraints that are in trees and constraints that are part of a cycle (that is, a non-tree-like structure). Furthermore, the local propagation manager should be able to determine which constraints belong to a certain cycle. Therefore, all constraints that are in the same cycle should have a common attribute, so that they can be identified.

The problem that has to be solved consists of three sub-problems:

1. Traversing the network,
2. Identifying cycles and trees,
3. Identifying nodes that together belong to a certain cycle.

The network under consideration is built up of nodes, where a node is either a constrainable or a constraint. Between each node, there is either zero or one undirected edge. Two nodes that are connected by an edge are neighbors. A collection of nodes $\{v_1, \ldots, v_k\}$ is in a cycle if $(\forall v_i : 1 \leq i \leq k)$ a traversal path $(v_i, \ldots, v_i)$ can be found without traversing any edge more than once. If a collection of nodes forms a cycle, all the constraints that are represented by these nodes are also in that cycle. The same holds for a collection of nodes that forms a tree. See Figure 6.4, for an example.

The network analysis is done by traversing the network. That is, given a current node, a neighbor is selected. The current node is marked visited, the edge is traversed, and the neighbor is also marked visited. If a current node has no neighbors or all neighbors have been visited before, backtracking to the previous node is done. Backtracking is not interpreted as an edge traversal. This algorithm visits all nodes in a depth-first traversal manner with backtracking. It continues until all edges have been traversed exactly once (not counting the backtrack steps).

We will call all nodes that have been visited previously to a current node c, the *ancestors* of c. The last ancestor that was visited before c was visited is the *parent* of c. All nodes that will be visited

after c are the *children* of c. The children of c that can be visited directly after c has been visited are called the *next neighbors* of node c.

While traversing the network, we alter each undirected edge by giving it a direction, namely the direction in which it is traversed. After having traversed every edge exactly once, this will lead to the following situation concerning the nodes.

- Every node, except the first node, has one incoming edge from its parent node.
- Every node that has p next neighbors ($1 \leq p \leq m-2$, where m is the number of nodes in the network) has p_o ($1 \leq p_o \leq p$) outgoing edges to these next neighbors.
- Every node that has p next neighbors ($1 \leq p \leq m-2$, where m is the number of nodes in the network) has p_i ($0 \leq p_i \leq p-1$) incoming edges from these next neighbors ($p_o + p_i = p$).

While backtracking, it can be determined whether a node is part of a tree or a cycle. There are some trivial cases (between brackets examples are given which refer to Figure 6.4).

- If a certain node c, has no children, the status of c is a tree node (nodes 9 and 14).
- If all next neighbors of node c have status tree, then also the status of c is tree (node 12).
- If there is a next neighbor n of c that already is marked visited when c selects it (that is, node n is *revisited* by one of its children c), the status of c is cycle (node 11, when it visits node 8).

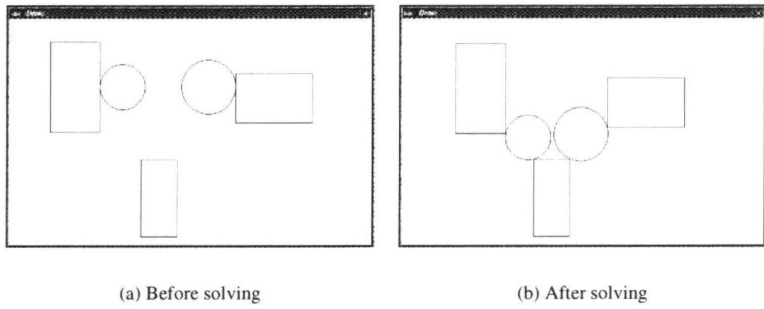
It is more difficult to determine the status of c when there is a next neighbor n that has status cycle while n was not revisited by c. For example, consider nodes 7 and 13 in Figure 6.4. The status of nodes 8 and 11, next neighbors of nodes 7 and 13, respectively, are cycle. Furthermore, both node 8 and node 11 were not revisited by nodes 7 and 13, respectively. Yet, node 7 is part of a tree, whereas node 13 is part of a cycle.

The difference between nodes 8 and 11 is that there is a node that has revisited node 8 (namely, node 11), while no node has revisited node 11 (only backtracking occurred). This information will be used to determine that node 7 is not in a cycle.

The algorithm is presented below. At each point in time, a current node c is visited. The set N_c contains all next neighbors of c. There is a variable cyc_c that indicates whether c is in a cycle or not. Initially, cyc_c is 0, indicating c is in a tree. If cyc_c is larger than zero, c is in a cycle. Finally, there is a variable rev_c that counts the number of times a node revisits c. That is, rev_c equals the number of incoming edges p_i.

```
 1. Int visit(Node c)
 2.     ``Mark c as visited.''
 3.     for (∀n ∈ N_c)
 4.        N_n := N_n \ {c}
 5.        if (``n is not marked visited'') then
 6.           cyc_c := cyc_c + visit(n)
 7.        else
 8.           cyc_c := cyc_c + 1
 9.           rev_n := rev_n + 1
10.        fi
11.     rof
12.     return(cyc_c − rev_c)
13. tisiv
```

The algorithm consists of execution of the recursive function `visit()`, which is called the first time by `analyse_network()` in `LP_cycl()`. The function takes as arguments a `Node` and returns an integer which is used to determine if the node is in a cycle or not.



(a) Before solving (b) After solving

Figure 6.5: Snapshots of Drawtool. Touch constraints are solved sequentially by a local propagation manager starting at the object that moved most recently (the middle rectangle).

In `visit()`, the current node c, is investigated. This node is marked as visited (line 2). After this, the next neighbors of c are investigated (line 3). For a selected $n \in N_c$, c is removed from N_n (line 4). That is, the current node c is removed from the set of next neighbors of node $n \in N_c$. This implies, that the edge between node c and n is traversed only once.

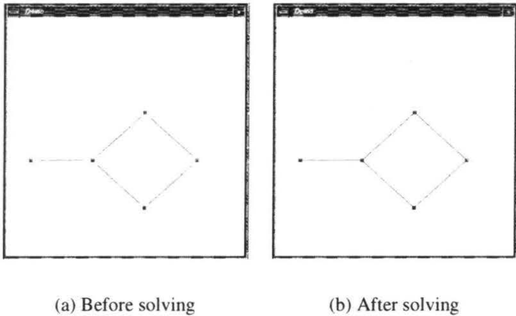
If n was not visited before, the function `visit()` is called recursively for node n and the return value of `visit()` is added to cyc_c (line 6). If n has already been visited, c is in a cycle and its cyc_c is increased (line 8). Furthermore, c has revisited node n and, thus, rev_n is increased (line 9). After all children of node c have been visited, node c backtracks to its parent. Finally, node c returns the difference between cyc_c and rev_c to the previous node (line 12).

Just before backtracking, node c can determine if its parent is or is not part of the cycle that c is in. As can be seen in the algorithm, a node c that is removed from a set N_n (where $n \in N_c$) is a node that is always marked visited. Therefore, no unvisited nodes are removed from the neighbor set, N_n, of any node n. This implies that all nodes of the network will eventually be marked visited, thus, that the number of `visit()` calls equals at least m, where m is the number of nodes in the network. It can also be seen that the function `visit()` is never called for a node that has already been marked visited (line 5). Since the number of nodes in a network is finite and a node is marked visited when `visit()` is called for that node, this implies that the algorithm always terminates and that the number of function calls that execute `visit()` is at most m. Together with the previous conclusion, this leads to exactly m calls of function `visit()`.

In each execution of `visit()`, the `for`-loop is executed for all next neighbors of a current node c. In a maximally connected graph, the number of next neighbors equals $(m - 1)$. Since the removal of node c from set N_n ($n \in N_c$) in line 4 of the algorithm means that the edge between node c and n is traversed only once, the average number nodes visited by c equals $((m - 1)/2)$. Since the number of `visit()` calls is m, the time-complexity of the algorithm equals $\mathcal{O}(m(m - 1)/2) = \mathcal{O}(m^2)$.

6.4 Object-Oriented Application Drawtool

In the context of PROTOM, a typical Object-Oriented application (OO-application) is a program that is operated by an end-user, who communicates with the system via a graphical user-interface (and might be totally unaware of constraints and objects). The programmer of the application, the application



(a) Before solving (b) After solving

Figure 6.6: Snapshots of Drawtool. Coincidence constraints between points and lines are solved sequentially by a local propagation manager. Constraints in a cycle are solved by a cycle manager.

engineer, builds the system in an object-oriented way and puts constraints on objects.

For PROTOM and the GA-CHE, a simple application, called Drawtool, was built in which graphical objects could be manipulated. Relations between the objects can be specified using constraints that are provided by the GA-CHE. Examples of graphical objects are circles, rectangles, lines, points. Examples of constraint types include touch constraints, equal area, constraints, and intersection constraints.

All objects in the OO-application communicate with each other via message passing. Some of these objects, the ones on which constraints can be imposed, are extended with additional functionality (inherited from class `Constrainable` of the OO-API), which allows them to communicate with PROTOM. When no constraint solving is at hand, the constrainables behave as ordinary objects. During constraint solving, message passing is locked and communication takes place via events and data flows.

In the Figures 6.5 and 6.6, some screen shots of Drawtool are taken. A user can modify the graphical objects that are displayed to him by selecting them using a mouse. After a modification operation, such as moving or resizing an object, has taken place, the object executes its `commit()` messages which causes the GA-CHE to solve the constraints.

6.5 Conclusion

The aim of the implementation of PROTOM was twofold. First, to investigate **CODE**'s approach to circumvent information hiding for solvers by separating the two paradigms. Second, to study and verify **CODE**'s communication protocol.

The implemented system is a simple application for manipulating geometrical objects in which constraints can be imposed on the objects. When an object is adapted, the constraints are solved by a combination of local propagation and a cycle manager.

In the system, a strict separation between the OO-application and the constraint framework is maintained. The objects only communicate via messages, while, on the other hand, constraint and solver entities have direct access to the object's internal state via data flows. The communication pro-

tocol is implemented as accurately as possible using the MANIFOLD language. Using MANIFOLD, it was possible to implement every entity as a separate process and to define communication and communication patterns among the processes indepedent of the data that was transfered. Similar separation between communication and computation can be found in middleware systems (CORBA, DCOM) and the TOOLBUS architecture [Bergstra et al., 1998]. In the latter, a number of *tools* can (only) cooperate with each other via a communication bus. This allows heterogeneous software components to be integrated with each other. Unlike MANIFOLD, the TOOLBUS is not a language (it is controlled by scripts) and does not have the same notions of events and data flows. However, it has a formal basis and can formally be analyzed (for example, by using discrete time algebra).

Evaluating the first aim of the PROTOM system, it turns out that the approach is very effective for avoiding conflicts with information hiding. Furthermore, the separation of the paradigms leads to a separation of concerns, where object-oriented modeling does not interfere with solving the constraints. The extra work that is required by an application programmer to use the system is minimal (inheriting from some classes and implement some functions).

In order to study CODE's communication protocol, all elements of the conceptual model, such as the entities, data flows, and events, are explicitly implemented as MANIFOLD processes. In this way, it was possible to check, adjust, and restructure the communication protocol of CODE.

Having implemented the system, it turns out that a major drawback for using it as an interactive drawing tool is the long delay time needed each time the constraints have to be solved. This delay is caused by the fact that for each constrainable, constraint, and manager entity, several processes are instantiated which have to communicate with each other via streams that are constantly being created and removed. These activities produce an overhead that makes the system take several seconds to solve the constraints.

A solution to reduce the overhead can be achieved by reducing the number of independent processes that need to communicate during constraint solving. However, doing this would obscure the 'one-to-one' mapping between the system and the conceptual model. As was indicated before, efficiency concerns were not taken into consideration when the system was designed. Rather, the goal was to study the behavior of the individual entities of the model and to demonstrate that the proposed model of communication indeed separated the message passing activity from the data flow communication used for solving the constraints.

Chapter 7

Application Implementation

This chapter describes the application implementation that was developed for the VR-DIS project and the GDP project which were mentioned in Chapter 1. This application implementation encompasses a constraint system for the object-oriented application in the two projects. The aim is to build the system as an independent software library. That is, it is independent of the applications for which it is developed.

The constraint system is composed of two main components. the first is the SCAFFOLD system (Solver for Constraints And Fabric For Object-oriented Library Design). SCAFFOLD is the kernel that implements all functionality of the model **CODE**. The second component is called the VD-CHE (VR-DIS Constraint Handling Engine), which implements specific constrainable, constraint, solver, and manager types. The VD-CHE offers these types to the OO-applications in the VR-DIS project and the GDP project. The VD-CHE uses the SCAFFOLD system to implement the communication among constrainable, constraints, solvers, and managers.

In the next section, a short summary of the backgrounds of the projects in which the OO-applications are developed is given. In Section 7.2, the top-level design of SCAFFOLD and the VD-CHE is treated. Section 7.3 describes the SCAFFOLD system. Section 7.4 describes the VD-CHE constraint handling engine. In Section 7.5, the two OO-applications are presented. Finally, Section 7.6 concludes the chapter.

7.1 Project Backgrounds

The constraint system was built for the VR-DIS project and the GDP project. The VD-CHE engine was initially developed for the VR-DIS project. The constrainable objects and constraint types that are provided by the VD-CHE are determined here. The background and the need for constraints in this project are described in Section 7.1.1, After this, the SCAFFOLD system was incorporated in the GDP animation system. The background of the GDP and its relation to constraints is the subject of Section 7.1.2.

7.1.1 VR-DIS Project

The VR-DIS project was started at the Architecture department of the Eindhoven University of Technology in order to investigate the combination of a new medium like virtual reality (VR) with the architectural design process. In architectural design, much effort has been spent on systematically describing the design process (see, for example, [Roozenburg et al., 1995], [Akin, 1986], [Bax, 1989]).

Although many parts of this process, especially the creative part, are difficult to describe formally, it has proven to be very useful to let specific tasks of the process be supported by computers.

Whereas current software packages for architectural design are based on the traditional design technologies, the medium of VR offers new ways for building architectural design systems. VR relies on three-dimensional, stereo scopic, head-tracked displays, hand/body tracking, and binaural sound. It enables an architect to design a building as if he or she is creating a full scale model, provided the designer has the appropriate tools to manipulate the design and to easily change the view. The architect can use VR to judge a design on its aesthetic and its functional qualities.

In the VR-DIS project, feature based modeling (see [Shah et al., 1995], [Leeuwen et al., 1998]) is used to support dynamic information modeling. Feature types can be associated with building components, but they enclose the additional information that a designer wishes to be captured. To specify and maintain geometric relations within feature type modeling, also the use of constraints has been investigated [Dohmen, 1998].

In the VR-DIS project, it is investigated whether constraints can be used to describe the behavior of the design. Using geometric constraints, a designer can specify relations among building components that are related to each other while a constraint solver maintains the relations. This facilitates the task of the designer and makes the maintenance of the relations less error-prone. A prerequisite is that constraint solvers need to be powerful engines to satisfy the constraints. Especially in situations where constraints conflict with each other (over-constrained, there is no solution) or situations in which constraints do not specify a unique solution (under-constrained, there is more than one solution), there is not one general approach to solve the problem.

In Section 7.5.1, the OO-application is described that is built for the VR-DIS project and which uses SCAFFOLD and the VD-CHE. In the system, constraints can be imposed on three-dimensional objects and a constraint system maintains these relations whenever a constraint gets violated due to actions of the designer.

7.1.2 GDP Animation System

The GDP (Generalized Display Processor, see [Peeters, 1995]) is a system for creating and manipulating computer animations. In the system, animated objects display autonomous behavior that can interactively be controlled by a user through direct manipulation (a running animation can be controlled by means of a hardware device, for example, a mouse) and direct modification (a running animation can be controlled by means of script fragments).

Direct modification is done through an object-oriented script-language, called Looks. Looks code is interpreted at run-time and offers full object-orientation, dynamic binding and garbage collection. Furthermore, the language supports concurrency that allows objects to perform operations concurrently and to perform multiple operations simultaneously.

The GDP continuously executes the Looks script to create a sequence of frames that together make up the animation. Basically, the GDP main loop is summarized as follows.

```
1. while TRUE do
2.     ''Parse Looks script''
3.     ''Interpret parsed script''
4.     ''Render the frame''
5. od
```

During parsing, Looks script fragments are read and checked whether they conform to the Looks syntax rules and context conditions such as correctness and uniqueness of class definitions. Based

on the script, the parser creates abstract syntax trees (ASTs) that can be read by the interpreter. The interpreter executes the statements that are stored by the parser in the ASTs. This encompasses the evaluation of methods and the update of object states due to their velocity, acceleration, or applied forces (for dynamic objects, see below).

Finally, the renderer creates a view of the 3D scene as it is seen from one or more virtual cameras. The scene can be viewed using one of several rendering techniques (flat shading, Gouraud shading, Phong shading) in combination with different light sources (ambient, diffuse, specular).

Looks contains an elaborate collection of default classes. This includes classes for input and output, for data structures like arrays, collections, classes for geometrical objects, widgets, cameras, and so on. Since Looks is an object-oriented language (supporting, for example, inheritance and information hiding), it can easily be extended by new classes. Besides building new classes, Looks can dynamically bind pre-compiled C or C++ code. This enables to incorporate functionality which is not (yet) available in Looks or which would be too inefficient if it is implemented in Looks.

The object-orientedness of the script-language promotes the well-structured design of the *data model* of an animation, since the animation entities map intuitively to objects that exist in an object-oriented system. Constraints, on the other hand, can be applied to facilitate the *behavior* of the animation entities. Using constraints to simulate reality is convenient since reality often seems to behave as if it obeys constraints.

Looks provides several types of constraints that can restrict the translational or rotational motion of *rigid* objects or that can be used to connect rigid objects to each other. Rigid objects are objects whose local 3D points remain fixed with respect to a body-fixed frame. Examples of constraints that restrict an object's motion are the prism constraint, which restricts translation to a certain vector, and the revolute constraint, which restricts rotation to one axis. Examples of connection constraints are point-to-point constraints, which specify that two points have the same coordinates, or point-to-line and point-to-curve constraints, which specify that a point has to lie on a line or curve, respectively.

Depending on the specific class of the objects, the constraints are maintained in two different ways. The first one is by using *kinematics*, that is, adjusting position or orientation of objects using kinematic operations such as translate or rotate. The second one is by using *dynamics*, that is, applying *forces* to objects in order to maintain constraints. Furthermore, kinematic objects can be instructed to follow the movements of a target object, which can be used to easily model, for example, a camera that has to follow an object. For dynamic objects, sophisticated constraints can be used to connect objects to each other, to simulate different kinds of springs, or to simulate rolling wheels [Barenbrug, 1999].

Although Looks provides a rich set of constraints, these constraint types can only be applied to rigid objects. Besides this, the Looks constraint solving techniques (inverse kinematics and relaxation) are focussed on constraint maintenance which requires a valid initial solution as opposed to more general constraint solving techniques. The SCAFFOLD system is added to the GDP to alleviate these problems. The functionality of SCAFFOLD provides a general structure for modeling constraint types, solvers, and managers that do not necessarily only deal with constraint maintenance. The integration of the constraint system in the GDP project is the subject of Section 7.5.2.

7.2 Constraint System Top Level Design

The design of the constraint system is based on the experiences with the PROTOM implementation. This leads to the first design decision.

Design Decision 7.1

The system is divided into two main components, a *Kernel* part, called SCAFFOLD, and a *Constraint Handling* part, called the VD-CHE. □

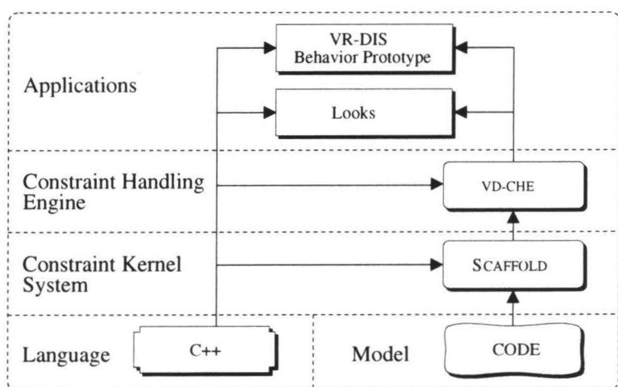


Figure 7.1: The constraint system SCAFFOLD is based on the **CODE** model. The constraint engine VD-CHE is built on top of SCAFFOLD. Two OO-applications, the VR-DIS Behavior Prototype and Looks, use the constraints of VD-CHE. The applications and systems are implemented in C++.

Like in PROTOM, the kernel implements the main functionality of the conceptual model. The constraint handling engine implements specific constraints and solvers that are accessible by an application.

In Figure 7.1, an overview is given of all the systems involved. The structure of SCAFFOLD is based on the **CODE** model. The VD-CHE uses the functionality of SCAFFOLD to implement specific types. Two applications, the *VR-DIS Behavior Prototype* of the VR-DIS project and the Looks language of the GDP project, use the constraint system implemented by the VD-CHE. In the picture, one implementation language is indicated for all systems.

Design Decision 7.2
Since OO-application of the VR-DIS project and the GDP project are implemented in C++, it is decided that this language is also used for the implementation of SCAFFOLD and the VD-CHE. By choosing C++, problems that can arise by coupling two different languages are evaded. □

The SCAFFOLD Kernel (see Section 7.3) provides the basic functionality of the conceptual model. This implies providing the four entity types (constrainable, constraint, solver, and manager) and allowing them to communicate via events and data flows.

Design Decision 7.3
Like in PROTOM, the entity types are provided by means of abstract super classes. The functionality can be used by creating new classes that inherit from the abstract superclasses. □

In Figure 7.2, an object diagram is depicted which presents the four entity classes `Constrainable`, `Constraint`, `Solver` and `Manager`. Functionality that these classes provide is, for example, solvers and constraints get the internal state of a constrainable via data flows (which is implemented as specific messages), a constrainable an raise an event to indicate a state change which is picked up

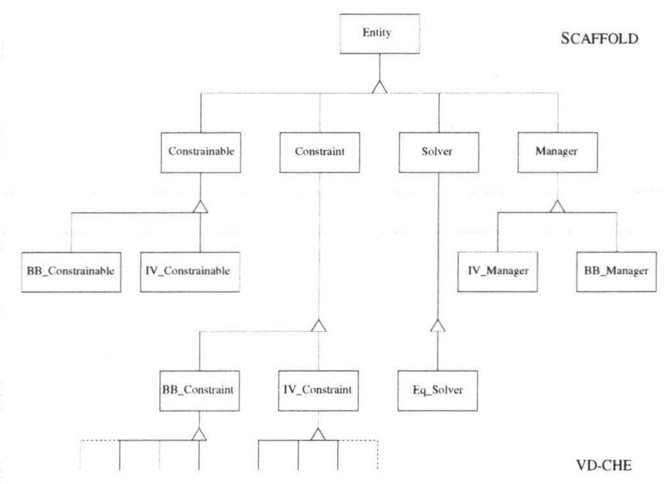


Figure 7.2: Object diagram of SCAFFOLD and VD-CHE inheritance structure.

by the imposed constraints. The entity classes inherit from one common class `Entity`. This class enables that all entities share some common attributes.

The classes of the VD-CHE (see Section 7.4) inherit from the four entity classes. There are two specific constrainable classes, `BB_Constrainable` and `IV_Constrainable` which represent bounding boxes and intervals, respectively. Two specialized constraint classes, `BB_Constrainable` and `IV_Constrainable`, provide constraints that can be imposed on the constrainable objects. Subclasses of `BB_Constrainable` provide constraint types that can be imposed on `BB_Constrainable` objects. For example, `touch`, `align`, `distance`.

The VD-CHE has two manager classes, `BB_Manager` and `IV_Manager`, that deal with the respective constraint classes. A solver class, `Eq_Solver` is dedicated to solve numerical equations.

In the next sections, we will take a closer look at SCAFFOLD and the VD-CHE.

7.3 SCAFFOLD System

The SCAFFOLD system provides the basic functionality for dealing with constrainables, constraints, solvers, and managers. It is similar to the kernel of PROTOM (see Chapter 6). The PROTOM kernel was built using the **MANIFOLD** language, enabling us to implement entities, events, and data flows. In the SCAFFOLD system, the aim is to implement the same functionality using the C++ language only.

Below, we will consider the main aspects of **CODE** and describe how they are implemented in the SCAFFOLD system.

7.3.1 Entities

The implementation of constrainable, constraint, solver, and manager entities is done in a straightforward way. Namely,

Design Decision 7.4
A single entity type is represented by one *entity* class. □

This leads to the entity classes `Constrainable`, `Constraint`, `Solver`, and `Manager`. Objects of class `Constrainable` can act as operands of an object of class `Constraint`. Objects of `Constraint` are added to an object of class `Manager` and are also managed by this object. A `Manager` object triggers objects of class `Solver` to calculate a valid solution for the constraints.

In Section 4.1, the functionality of all entities is described. This functionality is provided by the respective classes as follows.

- Class `Constrainable` provides methods to put an object in solving mode or in normal mode A method `commit()` can be called to indicate a state change. It has features to enable direct access to its internal data for solvers and constraints but not for objects of other types.
- Class `Constraint` provides virtual methods `check()`, to check a constraint's validity, and `solve()`, to calculate a local solution. These methods are implemented in subclasses.
- Class `Solver` provides methods to trigger solving, set the constraints to be solved, and methods for passing parameter data. The actual implementation of the methods is done in the subclasses.
- Class `Manager` has internal data structures to store a network of constraints. Furthermore, there are methods to add or remove `Constraint` objects, to trigger constraint solving, and methods to retrieve information concerning the constraint network.

7.3.2 Events

Events that are raised by one entity should be detected by other entities for which these events are meaningful. In the PROTOM kernel, forwarding events was carried out by **MANIFOLD**. The handling of these event now has to be carried out by C++. The language does not have a built-in event mechanism. Therefore, the events of **CODE** are implemented as specific message calls.

We want these message calls to have the same functionality as the events of **CODE** (see Section 4.3). In particular, an entity should be able to indicate its sensitiveness to events. An entity that detects an event should be able to find out who the source is. And for untargeted events, the source of the event should not be concerned with its receivers.

Design Decision 7.5
The class that implements messages to supply events is called `CommOrch`, the communication orchestrater. In an application, there is exactly one (automatically generated) object of this class that handles all communication via events (and data flows, see below). □

In this respect, raising an event means sending a message to the object of class `CommOrch`.

In **CODE**, there are two kinds of events, targeted and untargeted. If an entity raises an untargeted event, it does not know which entities will detect it. In case of targeted events, the source indicates which entity should receive the event.

The class `CommOrch` provides methods for the untargeted events. An object that calls one of these methods passes an identification of itself and the event. Based on these parameters, the `CommOrch` object determines to which objects the "event" has to be "forwarded". In the current version of SCAFFOLD, the `CommOrch` class determines which events are forwarded to which entities. It does not provide methods for an entity to indicate to which events it wants to be sensitive or insensitive. Forwarding events is done according to the specification in **CODE**. This means the following.

- In **CODE**, the E-changed event is raised by a constrainable entity to indicate a state change. This event is received by the constraints that are imposed on the constrainable.

In SCAFFOLD, to raise this event, a constrainable object calls the method `cstn_event()` of class `CommOrch`. The `CommOrch` object then calls the `detect_event()` method of class `Constraint` for all constraints that are imposed on the object and passes the event type and identification of the source as parameters.

- In **CODE**, the event **E-violated** is raised by a constraint to indicate that it is violated. The constraint manager detects this event.
  In SCAFFOLD, if a constraint wants to raise this event, it calls the method `cstr_event()`. The `CommOrch` object will then call the method `detect_event()` of the constraint manager object that manages the constraint.
- In **CODE**, the **E-solving** and **E-normal** are raised by constraints and managers to change the modes of a constrainable.
  In SCAFFOLD, class `CommOrch` provides the methods `set_solving()` and `set_normal()` which can be called by a constraint object or manager object. When one of these methods is called by a constraint, the `CommOrch` object will put the operands of the constraint in solving or in normal mode, respectively. When called by a constraint manager, constrainables that are operands of the managed constraints are put in solving mode or normal mode, respectively.

The methods `cstn_event()` and `cstr_event()` of class `CommOrch` are public methods. This implies that they can be called by any object. Undesirable situations that could occur if the method is called by "unauthorized" objects are prevented by the fact that the caller has to pass its identifier as parameter of the method. This identifier is used by the `CommOrch` object to check whether the caller is a registered constrainable or constraint object. Similarly, the `detect_event()` methods of the classes `Constraint` and `Manager` may not be called by any object, but only by an object of `CommOrch`. Thus, `Constraint` and `Manager` objects check if the caller is the `CommOrch` object that manages them.

Next to the untargeted events, there are the targeted events. If we would let these be handled by the class `CommOrch`, the following scenario can be constructed.

1. An object `A` that wants to raise a targeted event calls a method of class `CommOrch` indicating the type of the event and the target object, `B`.
2. The `CommOrch` object calls the method of the target object `B`.

In this scenario, the event raising object and the receiver are decoupled, which can have some advantages. For example, the event raising object deals only with an identifier for the target object and the `CommOrch` object is concerned with re-routing the event to the exact address location of the target object.

In the targeted event situation, the source of the event knows the target and the action that has to be performed. In this respect, an object `A` raising a targeted event for an object `B` is similar to object `A` calling a method of object `B`. However, the objects are not decoupled.

Design Decision 7.6

There is no need to decouple the objects since they all exist as C++ objects in one application. For this reason, targeted events are not implemented as methods of the `CommOrch` class. They are provided as methods by the entity classes of the SCAFFOLD System themselves. □

An example of a targeted event which is implemented as a message is the **E-solve** event. It is raised by the manager to trigger a solver entity. The class `Solver` provides a method `solve_cstrs()` that is called by an object of class `Manager`.

7.3.3 Data Flows

The data flows that exist in **CODE** are the following (see Figure 5.1):

- State flows from constrainables to constraints and between constrainables and solvers,
- Parameter flows and entity flows among constraints, solvers, and managers.

The state flows to and from the constrainables are essential. They guarantee that information hiding is ensured among objects while constraints and solvers have direct access to the object's state. The other data flows are not concerned with protection of constrainable data.

Design Decision 7.7
To deal with the data flows, there are three options.

1. All data flows are handled via the class `CommOrch`.
2. All data flows are provided as messages by the respective classes.
3. Some data flows are handled by the class `CommOrch`, while others are implemented as messages.

Like with targeted events, it is not efficient to handle all data flows via the `CommOrch`. However, if direct access to a constrainable's internal state is provided by messages of class `Constrainable`, this implies that any object can approach this data. Therefore, state flows to and from constrainables are managed by the `CommOrch` object, while the other flows are implemented as methods of the entity classes of the SCAFFOLD system. □

For example, the entity flow from managers to solvers (see Figure 5.1) is implemented as a method `set_constraints()` in class `Solver`. The entity flow and the method `set_constraints()` have the same functionality, namely, communicating the constraints that have to be solved from the manager to a solver.

Class `CommOrch` has two methods to implement the state flows for constrainables. These are `get_state()` and `set_state()`. An object that calls one of these methods passes a reference of itself. The `CommOrch` object checks if the calling object is allowed to execute the method, that is, whether the calling object is a constraint or solver object. If so, the `CommOrch` object calls the method `get_state()` or `set_state()` of class `Constrainable` which performs the actual state transfer.

The methods `get_state()` and `set_state()` of class `Constrainable` are private methods that can only be called by the `CommOrch` object. Access to this private method is granted to the orchestrater object using the `friend` construct of C++. Furthermore, the `get_state()` and `set_state()` methods check whether the caller is the `CommOrch` object, thus preventing that other `friends` can execute the method.

Finally, subclasses of `Constrainable` implement the `set_state()` and `get_state()` as private methods. This entails putting the constrainable variables in a special data structure that can be transported by the class `CommOrch`.

In Figure 7.3, the object diagram is depicted which shows the most important relations between the classes in SCAFFOLD. Besides the five entity classes, three additional classes are shown, `Comm-Orch`, `Network`, and `Node`. The arrows in the object diagram indicate which object has a reference (a pointer in C++) to another object. A plain arrow represents exactly one reference, an open circle at the end of the arrow means zero or more references, a closed circle means one or more references.

An object of class `Constraint` has references to the constrainables on which it is imposed. The number of references is at least one, because a constraint cannot exist without operands. An object of class `Manager` has a reference to one or more networks. An object of class `Solver` has exactly one network, which contains the constraints is has to solve. This network is initialized by a manager which

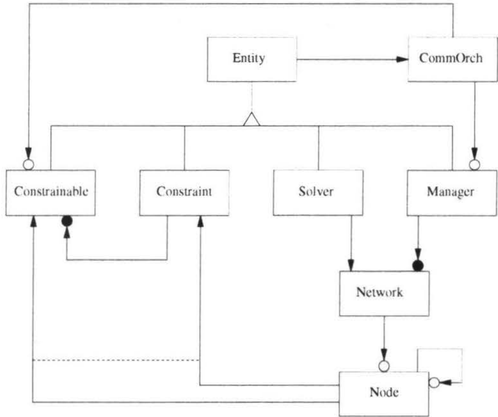


Figure 7.3: Object diagram of SCAFFOLD class relations.

triggers a solver to satisfy the constraints. A network contains zero or more objects of class `Node`. A node contains one reference either to a constrainable or to a constraint. This exclusive-or relationship is indicated in the figure by the dotted line. Furthermore, a node has one or more references to other nodes which are its neighbors.

The class `CommOrch` has references to constrainables and managers. The references to constrainables are needed to have quick access to all constrainables that are constrained, for example, to put them in solving mode or in normal mode. The references to the managers are used to get information about relations between constrainables and constraints, for example, in order to forward events that are raised by constrainables or constraints. The class `Entity` has one reference to the `CommOrch` object. This reference is used by all entity classes to raise events or to communicate via data flows.

7.4 VD-CHE Constraint Handling Engine

The constraint handling engine VD-CHE is initially built to support constraint handling in the VR-DIS project. The acronym VD-CHE stands for the *VR-DIS Constraint Handling Engine*. Typically, constraints should be available that can be used to define relations among architectural elements that constitute a house. For example, there should be constraints to specify the positions and relations of the components of the room as it is depicted in Figure 7.4.

The VD-CHE is an engine that provides one constrainable class, several constraint classes, one manager class and several solving techniques. All of its classes are subclasses from the abstract superclasses of the SCAFFOLD system. In this section, we will describe the different object and constraint types, the solving techniques that are applied, and the management mechanism that exists.

7.4.1 Object and Constraint Types

Constrainable Types

In architecture, many different shapes occur that represent the elements to build a house. For example, there are walls, doors, windows, roofs, floors, pillars, and so on. In order to have one general shape

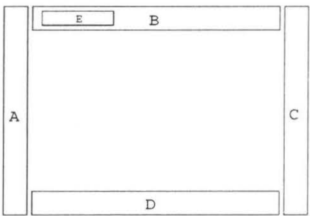


Figure 7.4: A room. The rectangles A, B, C, and D represent walls. Rectangle E represents a door in wall B. Connections between walls, the dimensions of walls, and the position of the door (relative to wall B) are specified by constraints.

that can be used to represent a broad class of architectural elements and that can be applied to a large class of constraint types, the following choice is made.

Design Decision 7.8
The objects that are constrainable are axis-parallel bounding boxes. These boxes can be used to represent different kinds of architectural elements, such as walls, floors, doors, or windows. □

In order to be able to determine the side of a box, that is, left, right, top, bottom, front, back, the following choice is made.

Design Decision 7.9
The bounding boxes have an orientation. That is, a box has a local coordinate system which defines the three dimensions (*width*, *height*, and *depth*) independent of its alignment relative to the axes of the global coordinate system. □

The three axes of the local coordinate system are called the *lateral* axis, the *vertical* axis, and the *frontal* axis. The lateral axis is the axis through the left side and right side of the box. The difference between left and right defines the width of the box. The vertical axis goes through bottom and top. The difference between bottom and top defines the height of the box. The frontal axis goes through the front and back. The difference between front and back defines the depth. Each local axis is always parallel to one the world coordinate axes (X, Y, or Z). For example, if the vertical axis is parallel to the Y-axis, the top and bottom sides of the box are situated on the Y-axis.

All the sides of a bounding box are uniquely determined by two points, its lower left corner (`llc`) and its upper right corner (`urc`), and an axis (`vaxis`), which indicates the vertical axis of the box. See Figure 7.5 for an example, where all coordinates of `llc` are smaller than those of `urc` and `vaxis` = Y-axis.

The VD-CHE constraint handling engine provides one class for constrainable bounding boxes (`BB_Constrainable`). This class inherits from the SCAFFOLD class `Constrainable` and implements the methods for state transfer.

Constraint Types

The following constraint classes are provided.

Design Decision 7.10

1. Unary constraints to limit the shape of a box, in particular, the position and dimensions of a box.

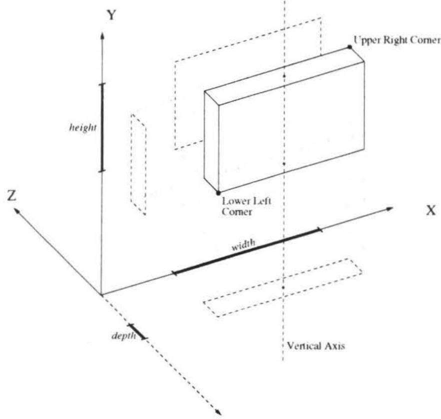


Figure 7.5: An axis-parallel bounding box. The position and orientation of the box are uniquely determined by two corner points, `llc` and `urc`, and an axis, `vaxis`. The thick lines segments show the projections of the box on the global coordinate axes.

2. Binary constraints to specify relations between two boxes, in particular, distances, connections, and intersection relations.

□

Unary constraints on boxes put bounds on the position or the dimensions of the box. There are position constraints that fix the center point of a box to a certain position or let it vary within a certain limit. Dimension constraints can fix the dimension of a box or specify a minimum and/or maximum dimension for a certain axis of the box.

Binary constraints are used to specify relations between boxes. Connection constraints specify that two boxes have to touch or have to be aligned. See Figure 7.6 for an example of a touch and an align constraint, respectively.

With respect to distance constraints, there are different sorts that can be distinguished. For example, consider the situation as depicted in Figure 7.7. In the left picture, box B_1 and box B_2 should have a fixed distance to one another. In the right picture, the left hand side of box B_3 should have a fixed distance to the left hand side of box B_4. For this reason, we define *opposite side* distance constraints and *same side* distance constraints. Opposite side distance constraints specify a fixed distance between two opposite sides of two boxes. For example in Figure 7.7, the distance between the right hand side of box B_1 and the left hand side of box B_2 is equal to d_o. Same side distance constraints specify a fixed distance between two same sides of two boxes. For example, the same side distance between boxes B_3 and B_4 is equal to d_l.

Next to the fixed distance constraints, also minimum and maximum distances can be specified. Other binary constraints are constraints that specify that one box has to contain another box or that two boxes may not intersect one another.

The general constraint class for all bounding box constraints is `BB_Constraint`, which inherits from the SCAFFOLD class `Constraint`. This class provides some basic methods that are necessary of all bounding box constraint types. Each specific constraint type inherits from the class `BB_Con-`

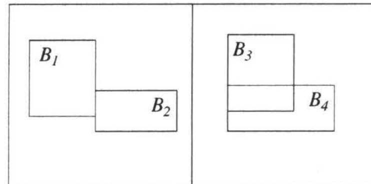


Figure 7.6: Connection constraints. The left picture shows a touch constraint. The bounding boxes B_1 and B_2 reside at opposite sides of the touch plane. The right picture shows an align constraint. The bounding boxes B_3 and B_4 reside at the same side of the touch plane.

`straint`. These subclasses specify the arity and parameters of the constraint and implement the method for validity checking. A summary of the constraint types is listed below.

- FixPosition(`BB_Constrainable` b, `Point` p)
  The constraint dictates that the center point of bounding box b must be equal to the specified point p.
- {Min/Max/Fix}Dimension(`BB_Constrainable` b, `LocAxis` ax, `float` x)
  The constraint MinDimension dictates that dimension ax of bounding box b must be at least minimum value x. The constraint MaxDimension dictates that dimension ax must be at most minimum value x. The constraint FixDimension dictates that dimension ax must be equal to value x.
- Touch(`BB_Constrainable` b_1, `BB_Constrainable` b_2, `Axis` a)
  The constraint dictates that the farther side of box b_1 (seen from the origin) must touch the closer side of box b_2 on axis a.
- Align(`BB_Constrainable` b_1, `BB_Constrainable` b_2, `Axis` a, `int` n)
  The constraint dictates that operand b_1 should be aligned with operand b_2 on axis a. The integer n indicates if the closer sides of the boxes (seen from the origin) must be aligned or the farther sides.
- OppDistance(`BB_Constrainable` b_1, `BB_Constrainable` b_2, `Axis` a, `float` d)
  The constraint dictates that the farther side of operand b_1 must have a specified distance d to the closer side of operand b_2 on axis a.
- LatDistance(`BB_Constrainable` b_1, `BB_Constrainable` b_2, `Axis` a, `float` d)
  The constraint dictates that operand b_1 should have lateral distance d to operand b_2. The sides of the boxes that are closest to the origin on axis a will have distance d, and b_1 is closer to the origin than b_2.
- Contains(`BB_Constrainable` b_1, `BB_Constrainable` b_2)
  This constraint dictates that box b_1 contains box b_2.
- NonIntersect(`BB_Constrainable` b_1, `BB_Constrainable` b_2)
  This constraint dictates that the two boxes b_1 and b_2 may not intersect or contain each other.

7.4.2 Constraint Management and Solving

The constraint types that are specified in the previous section can be used to model the room in Figure 7.4. The next step is to build a solver that can maintain the relations that are imposed by constraints. A solution to the constraints should meet the following requirements.

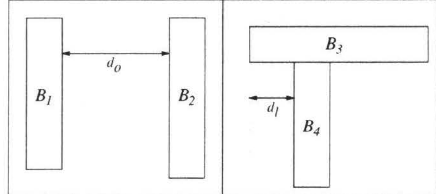


Figure 7.7: Distance constraints. The left picture shows an *opposite side* distance constraint. The bounding boxes B_1 and B_2 reside at opposite sides of the separating area. The right picture shows an *same side* distance constraint. The distance, d_l, between bounding boxes B_3 and B_4 is specified between two equal sides of the bounding boxes.

1. One solution has to be calculated which validates all constraints.
2. In case no solutions exist, an error message has to be generated.
3. In case multiple solutions exist, one solution has to be chosen that does not confuse or surprise the user (the architect). (In practice, an acceptable approach to this requirement is to calculate values that displace the boxes as little as possible.)

In order to validate the constraints, they are first decomposed or rewritten into a form that can easily be solved. Next, the values for the boxes are calculated such that the constraints are valid.

The listed constraint types all operate on complete bounding boxes. An axis-parallel bounding box can also be represented by its projections on the axes of the global coordinate system. See for example Figure 7.5, where the thick line segments on the axes represent the projections of the bounding box. Each projection occupies a certain *interval* on an axis. As a consequence, the constraints on bounding boxes can be rewritten to constraints on the intervals on the axes. These interval constraints can then easily be rewritten to numerical equations and inequalities which can be solved by a numerical solver. Thus, the outline for the solving strategy is as follows.

- First, the bounding box constraints are rewritten by a solver to constraints on intervals.
- Next, the interval constraints are sent to a solver that rewrites the constraints to equations and inequalities.
- Finally, the equations and inequalities are fed to a solver that can find a solution for them.

Design Decision 7.11

To map the solving strategy onto a solver–manager structure, several alternatives are possible.

1. There is only one manager that maintains the bounding box constraints and controls a number of solvers to rewrite the constraints and to calculate solutions.
2. There is a manager that maintains the bounding box constraints and controls a number of solvers to rewrite the constraints and communicates with other managers to solve the rewritten constraints.

For the VD-CHE, an intermediate solution is chosen. If a certain type of constraints is rewritten into another type, another manager is used to handle this other type of constraints. If a constraint type is rewritten into a form that can directly be used by a solver to calculate a solution, no other manager is used. □

For the VD-CHE, this leads to the following scenario (see Figure 7.8).

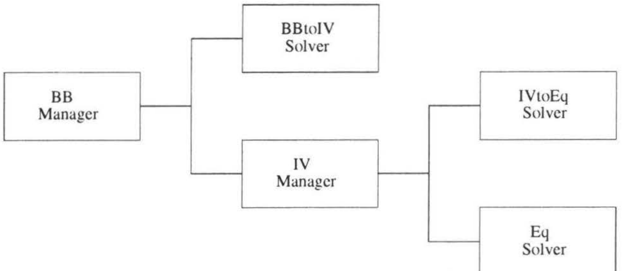


Figure 7.8: Overview of Solver and Manager structure. The rectangles represent objects. The lines represent *uses-a* relationships between the objects. That is, an object on the *left* side of a relationship *uses* the objects on the *right* side of the relationship.

1. A BB manager that maintains the bounding box constraints feeds the constraints to an BBtoIV solver that rewrites the bounding box constraints to interval constraints.
2. Because now there is a new type of constraints specified on a new type of constrainables, the interval constraints are sent to an IV manager that can handle these constraints.
3. The IV constraint manager feeds the constraints to a IVtoEq solver that rewrites the constraints to equations and inequalities.
4. Because the output of the IVtoEq solver is not a set of constraint objects, but a set of numerical equations, these equations are directly fed to the Eq solver that can calculate a solution.

The BB manager is an object of class `BB_ConstraintManager` (see Figure 7.3). This class inherits from the SCAFFOLD class `Manager`. The IV manager is an object of class `IV_ConstraintManager`, which also inherits from `Manager`. The Eq solver is an object of class `Eq_Solver`. In the current implementation, both solvers that rewrite constraints are not implemented as actual separate classes. Since these types of solvers need all information of the constraints they have to rewrite, they are incorporate with these constraint types for efficiency. However, it must be mentioned that separation of the rewrite solvers from the constraints is a more modular solution. It will become necessary to design separate classes for the rewrite solvers once the structure becomes more complex.

In the sections below, we will consider rewriting the constraints and solving the equations.

Constraint Rewriting

All constraints types, defined in Section 7.4.1, that specify relations on or between bounding boxes can be rewritten to specify relations on or between the projections of the boxes on each axis. For unary and binary bounding box constraints holds,

$$C_b(B_1) \leftarrow C_{iv}(x \cdot B_1) \wedge C_{iv}(y \cdot B_1) \wedge C_{iv}(z \cdot B_1) \tag{7.1}$$

$$C_b(B_1, B_2) \leftarrow C_{iv}(x \cdot B_1, x \cdot B_2) \wedge C_{iv}(y \cdot B_1, y \cdot B_2) \wedge C_{iv}(z \cdot B_1, z \cdot B_2) \tag{7.2}$$

In the formulas, C_b denotes a bounding box constraint and C_{iv} denotes an interval constraint. B_i $(i = 1, 2)$ are two bounding boxes and $k \cdot B_i$ denotes the interval that is obtained by projecting box B_i on axis k $(i = 1, 2$ and $k = x, y, z)$. The arrow ($\leftarrow$) denotes that the constraint on the left hand side of the arrow is rewritten by the constraints on the right hand side.

Relation	Illustration	Inverse
x equal y	x y	
x before y	x y	y before x
x meets y	x y	y meets x
x overlaps y	x y	y overlaps x
x during y	x y	y during x
x starts y	x y	y starts x
x finishes y	x y	y finishes x

Table 7.1: The 13 interval relations as defined by Allen.

Interval constraints can be described using Allen's relations [Allen, 1981]. Allen defined thirteen elementary relations to describe the relative positioning of two segments along an axis. See Table 7.1. However, using these relations it is not possible to specify sizes of or distances between intervals. Therefore, some additional interval constraints are defined in Table 7.2.

As an example, we consider the touch(B_1, B_2, x) constraint which specifies that the sides of B_1 and B_2 that reside on the x-axis should touch in such a way that B_1 is closer to the origin than B_2. This can be rewritten to interval constraints that specify that the projections of B_1 and B_2 on the x-axis have to *meets* (see Table 7.1) and that the projections on the y-axis and the z-axis have to *coincide* (see Table 7.2). Thus,

$$\begin{aligned} \texttt{Touch}(B_1, B_2, x) \leftarrow\ & x \cdot B_1 \text{ meets } x \cdot B_2 \ \wedge \\ & y \cdot B_1 \text{ coincide } y \cdot B_2 \ \wedge \\ & z \cdot B_1 \text{ coincide } z \cdot B_2 \end{aligned} \tag{7.3}$$

For the constraint types in Section 7.4.1 holds that every bounding box constraint C_b can be rewritten to a conjunction of (at most) three interval constraints C_{iv}. Each interval constraint can be rewritten to a set of linear equations and inequalities. Since we only deal with conjunctions, all interval constraints together can be rewritten to a single matrix which can be solved by a numerical solver (see below).

The VD-CHE does not deal with disjunctions in the interval constraints. However, we briefly investigate this situation. If the bounding box constraints introduce disjunctions in the interval constraints, the resulting equations and inequalities cannot be put in a single matrix. For example, consider a constraint type $\texttt{Touch}_d$ which, as opposed to the above mentioned type Touch, specifies a distance, but does not specify which of the boxes has to be closer to the origin. $\texttt{Touch}_d$ rewrites to the following interval constraints,

$$\begin{aligned} \texttt{Touch}_d(B_1, B_2, x) \leftarrow\ & (x \cdot B_1 \text{ meets } x \cdot B_2 \ \vee \ x \cdot B_2 \text{ meets } x \cdot B_1) \ \wedge \\ & y \cdot B_1 \text{ coincide } y \cdot B_2 \ \wedge \\ & z \cdot B_1 \text{ coincide } z \cdot B_2 \end{aligned} \tag{7.4}$$

Relation	Description
f fix x	Value f is middle of interval x.
sz size x	Value sz is size of interval x.
mn min x	Value mn is minimum size of interval x.
mn max x	Value mx is maximum size of interval x.
x eqBegin y	Intervals x and y have equal beginnings (x starts y, or y starts x).
x eqEnd y	Intervals x and y to have equal ends (x finishes y, or y finishes x).
x dist y, d	Difference between end of interval x and beginning of interval y is value d.
x distBegin y, d	Difference between beginning of interval x and beginning of interval y is value d.
x distEnd y, d	Difference between end of interval x and end of interval y is value d.
x coincide y	Intervals x and y coincide (x meets y, or x overlaps y, or x equal y, x during y, or x starts y, or x finishes y, or their inverses).

Table 7.2: Additional interval constraints.

$\texttt{Touch}_d$ does not specify the positions of the bounding boxes relative to each other. That is, either the first argument (B_1) is closer to the origin, or the second one (B_2) is. This introduces the *meets* constraint disjuncts.

A practical solution is to rewrite the collection of interval constraints into DNF (Disjunctive Normal Form) notation in which the set of constraints is expressed as a series of disjuncts and each disjunct is either a single constraint or a conjunction of constraints. In order to solve the constraints, only one disjunct has to be validated. The problem is to choose the right disjunct and, in case a chosen disjunct cannot be satisfied, determine another one.

The DNF notation for the $\texttt{Touch}_d$ constraint will be,

$$
\begin{aligned}
\texttt{Touch}_d(B_1, B_2, x) \leftarrow\ & (x \cdot B_1 \text{ meets } x \cdot B_2\ \wedge \\
& y \cdot B_1 \text{ coincide } y \cdot B_2\ \wedge \\
& z \cdot B_1 \text{ coincide } z \cdot B_2) \\
\vee\ & (x \cdot B_2 \text{ meets } x \cdot B_1\ \wedge \\
& y \cdot B_1 \text{ coincide } y \cdot B_2\ \wedge \\
& z \cdot B_1 \text{ coincide } z \cdot B_2)
\end{aligned}
\tag{7.5}
$$

In the VD-CHE constraint handling engine, the algorithm or heuristic for deciding which disjunct has to be chosen, would be typically situated in the IV Manager (see Figure 7.8). In case the choice for a disjunct is based on the current values of the bounding boxes, the IV manager would need an additional solver to do this (it would be an IVtoEq solver). This is because there are no state flows between a constrainable and a constraint manager, that is, a manager cannot retrieve a constrainables state (see Figure 5.1). A solver can inspect the state of a constrainable and can use this information to rewrite the interval constraints.

In SCAFFOLD, the practical complications have been avoided by only allowing conjuncts of constraints. However, adding disjuncts would not change the structure of the constraint handling engine.

The final rewrite step is to rewrite the interval constraints into linear equations and inequalities. This results in a matrix which will be solved by a numerical solver. In this matrix the columns represent the numerical variables, which are internal variables of constrainables, and the rows represent the constraints. That is, a constraint `Cr` on a constrainable `Cn` occupies a number of rows in the matrix, while the constrainable `Cn` occupies a number a columns. Since no constraints have an arity higher than 2 (they are only imposed on one or two constrainables), this results in sparse matrices.

As an example, we show here how the *meets* and *coincide* constraints rewrite to linear equations and inequalities.

$$I^{B_1} \text{ meets } I^{B_2} \leftarrow I_b^{B_1} \leq I_e^{B_1} \wedge I_b^{B_2} \leq I_e^{B_2} \wedge I_e^{B_1} = I_b^{B_2} \tag{7.6}$$

$$I^{B_1} \text{ coincide } I^{B_2} \leftarrow I_b^{B_1} \leq I_e^{B_1} \wedge I_b^{B_2} \leq I_e^{B_2} \wedge I_b^{B_1} \leq I_e^{B_2} \wedge I_b^{B_2} \leq I_e^{B_1} \tag{7.7}$$

I^{B_1} and I^{B_2} are two intervals belonging to bounding boxes B_1 and B_2, respectively. Variable $I_b^{B_i}$ $(i = 1, 2)$ is a scalar (with continuous domain) which has the value of the *beginning* of interval I^{B_i}. Variable $I_e^{B_i}$ is a scalar which has the value of the *end* of interval I^{B_i}.

Solving Linear Equations and Inequalities

The system of interval constraints that is rewritten into a set of linear equations and inequalities can be represented as follows.

$$\begin{aligned} \mathbf{A_1}\,\mathbf{x} &= \mathbf{b_1} \\ \mathbf{A_2}\,\mathbf{x} &\geq \mathbf{b_2} \end{aligned} \tag{7.8}$$

$\mathbf{x} \in \mathbb{R}^n$ is a vector containing n scalar variables. Each variable is associated with either the beginning or the end of an interval. The dimensions of the matrices $\mathbf{A_1}$ and $\mathbf{A_2}$ are $m_1 \times n$ and $m_2 \times n$, respectively, where m_1 is the number of equal-to $(=)$ equations and m_2 equals the number of greater-or-equal-than $(\geq)$ inequalities. $\mathbf{b_1}$ and $\mathbf{b_2}$ are the right hand side variables written as column vectors having size m_1 and m_2, respectively. Less-or-equal-than $(\leq)$ constraints are expressed in terms of rows in $\mathbf{A_2}$ and elements of $\mathbf{b_2}$ after multiplying all coefficients by -1.

For the variables in $\mathbf{x}$, a solution has to be calculated that satisfies the constraints, which means solving the system in Formula 7.8. The problem is modeled as a constrained optimization problem, where a proper objective function has to be optimized subject to the constraints. The optimization function is found by considering the requirement that a solution to the constraints should displace the boxes as little as possible. This can be achieved by minimizing the change of each variable. It leads to the following objective function f_1 that has to be minimized.

$$f_1(\mathbf{x}) = \frac{1}{2} \parallel \mathbf{x} - \mathbf{c} \parallel^2, \tag{7.9}$$

where $\mathbf{c}$ is a constant vector containing the current values of all variables. The constraints on this function are stated in Formula 7.8.

In the field of (non-linear) constrained optimization, the problem of optimizing a quadratic function subject to linear constraints is known as the *Quadratic Programming* problem (QP). A QP prob-

lem is defined as follows (see [Fletcher, 1987], [Luenberger, 1984]).

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{G}\mathbf{x} + \mathbf{g}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} & \mathbf{a}_i^{\mathrm{T}}\mathbf{x} = b_i, \quad i \in E \\ & \mathbf{a}_i^{\mathrm{T}}\mathbf{x} \geq b_i, \quad i \in I \end{array} \tag{7.10}$$

$\mathbf{G}$ is a symmetric and semi positive definite matrix. That is, the dimension is $n \times n$ and $\forall \mathbf{x} \neq \mathbf{0}$: $\mathbf{x}^{\mathrm{T}}\mathbf{G}\mathbf{x} \geq 0$. E and I are index sets for the equality and inequality constraints.

To be able to apply these methods, we first have to rewrite the constraints in Formula 7.8 and the function in Formula 7.9 into a QP problem. If we expand f_1, we get,

$$f_1(\mathbf{x}) \quad = \quad \frac{1}{2}(\mathbf{x}^{\mathrm{T}}\mathbf{x} - 2\,\mathbf{c}^{\mathrm{T}}\mathbf{x} + \mathbf{c}^{\mathrm{T}}\mathbf{c}) \tag{7.11}$$

In the minimization function, we can ignore the constant term ($\mathbf{c}^{\mathrm{T}}\mathbf{c}$). It can then be seen that $\mathbf{G} = \mathbf{I}$ (the identity matrix) and $\mathbf{g} = -\mathbf{c}$. The constraints in Formula 7.8 can be rewritten into the shape of Formula 7.10 as follows. Each vector $\mathbf{a}_i$ equals the i-th row in matrix $\mathbf{A_1}$ and b_i equals the i-th element of $\mathbf{b_1}$, for $1 \leq i \leq m_1$ ($|E| = m_1$). Similarly, each vector $\mathbf{a}_{m_1+i}$ equals the i-th row in matrix $\mathbf{A_2}$, and b_{m_1+i} equals the i-th element of $\mathbf{b_2}$, for $1 \leq i \leq m_2$ ($|I| = m_2$).

Several techniques are known to solve this problem. For example, the *Lagrange Method* can be used to solve the equality constraints and an *Active Set Method* to deal with the inequalities. The Lagrange methods optimizes some function which is subject to linear equality constraints. Active Set Methods select certain inequality constraints that are solved as equalities. When these methods are combined, the above described constrained optimization problem can be solved. For the theoretical background of these and other techniques, we refer to [Fletcher, 1987] and [Luenberger, 1984].

In the current implementation, another approach is taken. In the formulation of the problem in the beginning of Section 7.4.1, it was stated that a calculated solution should not "confuse" or "surprise" the user. A good heuristic is to calculate a solution that is as close as possible to the current situation, which can be done using QP as is described above. However, the minimum displacement is not a hard requirement. Calculating a solution that is *near* the current situation instead of the *nearest* solution would also be acceptable. The constraint solver in the VD-CHE calculates such a solution. Below, we describe how this is done.

The equality constraints are solved directly by using the *conjugate gradient* method which minimizes the function f_2 [Press et al., 1992].

$$\text{minimize } f_2(\mathbf{x}) \quad = \quad \frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} - \mathbf{b}^{\mathrm{T}}\mathbf{x} \tag{7.12}$$

where $\mathbf{A}$ is symmetric and positive definite. Any non-singular matrix can be made symmetric and positive definite by multiplying it by its transpose. Since in general, matrix $\mathbf{A_1}$ in Formula 7.8 will not be symmetric or positive definite, we define $\mathbf{A} = \mathbf{A_1}^{\mathrm{T}}\mathbf{A_1}$. Furthermore, $\mathbf{b} = \mathbf{A_1}^{\mathrm{T}}\mathbf{b_1}$. Consequently, the sizes of $\mathbf{A}$ and $\mathbf{b}$ are $n \times n$ and n, respectively.

A minimum is found when the gradient of f_2, the vector of first partial derivatives, ∇f_2, equals zero. As can be seen from Formula 7.12, $\nabla f_2 = \mathbf{A}\mathbf{x} - \mathbf{b}$, which means that minimizing f_2 solves the equalities (not the inequalities!) in Formula 7.8.

The conjugate gradient method is an iterative method to find the minimum of a quadratic function. In each iteration, the current solution is improved by selecting a descent direction vector $\mathbf{v}$ along which is searched for a better one. Vector $\mathbf{v}$ is determined using the gradient at the current position. The conjugate gradient method is similar to steepest descent, but it uses a more sophisticated technique to

determine the descent direction which leads to faster conversion to the minimum. The minimum of the quadratic function is found in n steps.

To deal with the inequalities of $\mathbf{A_2}$, a heuristic is applied that is similar to Active Set methods. That is, there is a set of inequality constraints that is treated as active and there is a set that is treated as inactive. Active constraints of $\mathbf{A_2}$ are added to matrix $\mathbf{A_1}$, resulting in matrix $\mathbf{A_1^*}$, and corresponding elements of $\mathbf{b_2}$ are added to $\mathbf{b_1}$, resulting in vector $\mathbf{b_1^*}$. (Consequently, we redefine $\mathbf{A} = \mathbf{A_1^{*}}^{\mathrm{T}}\mathbf{A_1^*}$ and $\mathbf{b} = \mathbf{A_1^{*}}^{\mathrm{T}}\mathbf{b_1^*}$ in Formula 7.12.) The inactive constraints are ignored as far as solving is concerned. This implies that active inequality constraints are treated as if they were equality constraints. It is obvious that if an inequality constraint is validated as an equality constraint, it is solved.

The set of active and inactive constraints is updated as the conjugate gradient method is iterating towards a minimum. Which inequalities should be active and which inactive is determined by checking their validity. Active inequalities that become valid during the process are made inactive, while inactive constraints that become invalid are made active.

If there is a solution to the system in Formula 7.8, this approach will eventually converge to this solution. If the system of equations defined by matrix $\mathbf{A_1^*}$ is over-constrained, there is no solution. However, the algorithm iterates to some value, because matrix $\mathbf{A}$ will not be over-constrained (due to the $\mathbf{A_1^{*}}^{\mathrm{T}}\mathbf{A_1^*}$ multiplication).

If there are multiple solutions, the starting point of the algorithm, say point $\mathbf{c}$, determines which one is found. Usually, since the algorithm iterates from $\mathbf{c}$ towards the solution space, the solution will be close to $\mathbf{c}$. However, this cannot be guaranteed and, consequently, it cannot be proven that the boxes are displaced as little as possible. On the other hand, examples that were tried in practice did not show unexpected large displacement of the boxes and acceptable results were produced.

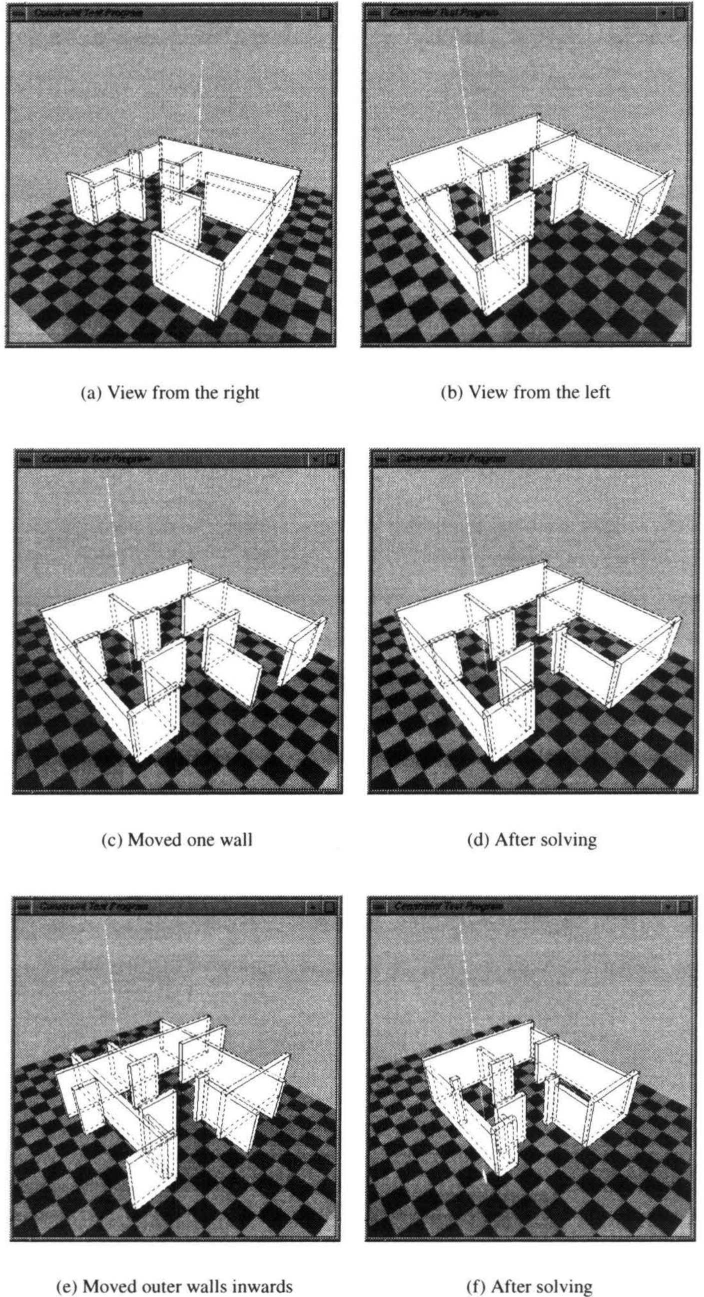
7.5 OO-Applications

There are two object-oriented applications that are based on the SCAFFOLD system. The first is the VR-DIS Behavior Prototype (see Figure 7.1) which is a three-dimensional modeling tool that is built for the VR-DIS project. The second is Looks which is the script language of the GDP. In the following sections, these two applications will be presented.

7.5.1 VR-DIS Behavior Prototype

The VR-DIS Behavior Prototype is an interactive, graphical system to create and manipulate architectural elements. It was developed in the VR-DIS group at the Architecture department. The aim of the prototype system was to investigate the role of geometrical constraints in architectural design. The prototype system provides axis-parallel bounding boxes that can be used to represent elements such as walls, doors, and windows. On the elements, constraints can be imposed to aid an architectural designer in modeling a scene. The user-interface of the application consists of a window which displays a three-dimensional view of the scene. Via this window, a user can manipulate the elements and the camera viewpoint (see Figure 7.9). Another window is a text window in which commands can be entered to create or remove elements and constraints and in which the status of the system can be inspected.

The VR-DIS Behavior Prototype starts one default constraint manager to which all created constraints are added. Constraint solving can be invoked explicitly by the architectural designer via the text window or it is invoked (automatically) by a constrainable that raises the E-changed event. In the latter case, the person who implements the VR-DIS Behavior Prototype indicates at which points an



(a) View from the right

(b) View from the left

(c) Moved one wall

(d) After solving

(e) Moved outer walls inwards

(f) After solving

Figure 7.9: Snapshots of the VR-DIS application.

element has changed to such an extend that constraint solving is necessary. At that point, the object calls its `commit()` message which raises the event and the constraints and the solving process.

The programmer of the VR-DIS Behavior Prototype only deals with the classes that are provided by the VD-CHE. The VD-CHE provides a class for constrainable bounding boxes (class `BB_Constrainable`) and the classes for bounding box constraints (the subclasses of `BB_Constraint`). There is one class that can be used to create constraint managers, `BB_Manager`. Via an object of this class, the programmer can maintain the constraints and set certain parameters for the solving process, for example, the number of iterations. The VR-DIS programmer has no access to the solvers that are created by the constraint manager. In this way, the constraint system itself is shielded from the programmer.

In Figure 7.9, some snapshots are depicted that give an impression of the system. In the pictures, the walls of one level in a house is modeled by constraints. Constraints specify that walls have to touch, specify distances between walls that contain doors, minimum sizes of walls, relative distances between walls, and so on. Figures 7.9(a) and (b) present the same house from two different viewpoints. In Figure 7.9(c), one wall is moved, while Figure 7.9(d) presents the solution that is calculated. Minimum dimension constraints prevent that the wall which size is decreased, becomes flat. In Figure 7.9(e), all outer walls are moved inwards. Figure 7.9(f) presents the solution. Note that in the figure, first all walls are moved and next solving is invoked. However, it is also possible to maintain the constraints as elements are moved. This option can be set by the architect who uses the prototype.

7.5.2 Looks

The language Looks is the scripting language to control the GDP animation system. Looks allows that compiled C++ code can be called from within the language. Since the SCAFFOLD library is implemented in C++, it can be linked to Looks. The coupling enables to create constrainable objects in a GDP animation on which the SCAFFOLD constraints can be imposed. In this way, the new constraints contribute additional functionality to the already existing constraints in Looks.

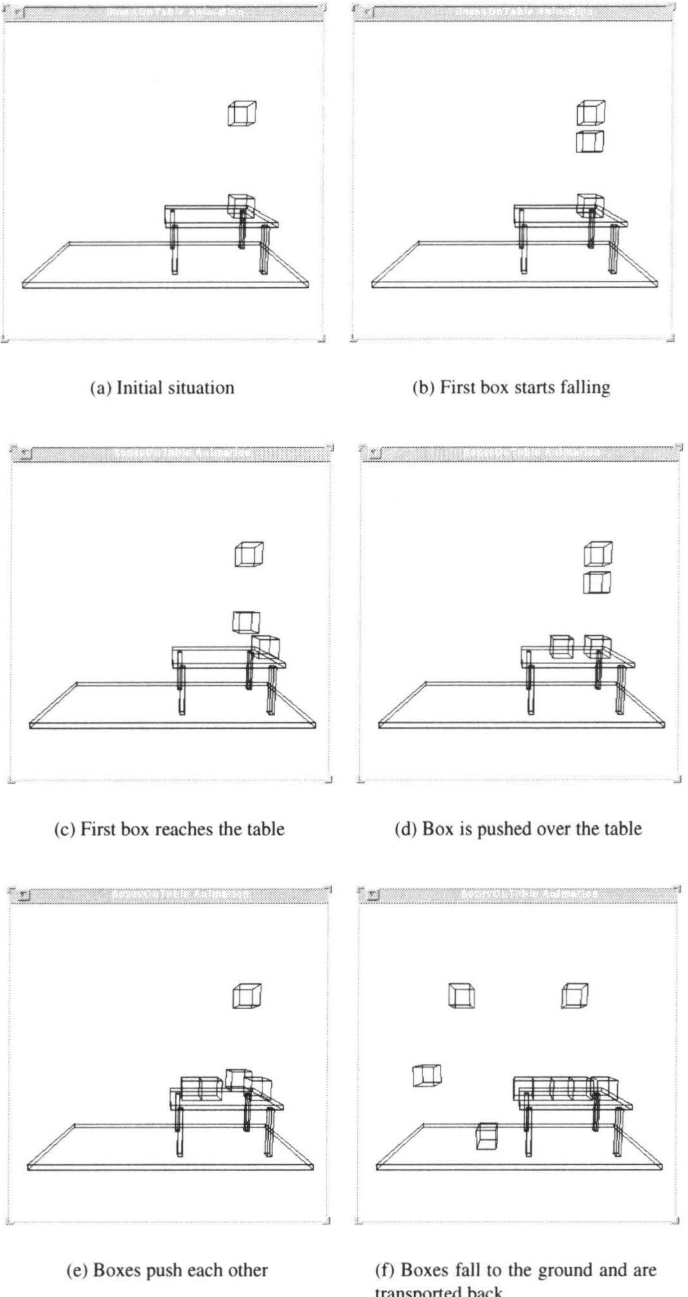
We require that no changes to either system have to be made in order to couple Looks with the SCAFFOLD library. The coupling should be realized by only creating the proper subclasses in Looks as well as in the SCAFFOLD library.

The script-language is extended with classes that a Looks programmer needs to impose constraints on constrainable objects. These include the classes to create constrainable objects, constraints, and constraint managers. There is one type of constrainable object provided which is the bounding box class from the VD-CHE library. There is a number of constraint classes, each class representing a specific constraint type that can be imposed on constrainable objects. The manager class can be used to create constraint managers to which constraints can be registered.

In the GDP main loop (see Section 7.1.2), a frame is drawn at the end of each iteration. It is preferable to have all constraints solved before the frame is rendered but after all objects performed their methods and have been updated. In order to trigger constraint solving, a Looks programmer has two options.

- The programmer calls the `solve_network()` method of a constraint manager object, whenever he wants constraints to be solved.
- A state change is indicated via the `commit()` method of constrainable class and the "system" handles efficient constraint solving.

A call `commit()` directly triggers constraint solving. Since, in general, more than one object will move in between two frames and an object can also move more than once, this can lead to solving



(a) Initial situation

(b) First box starts falling

(c) First box reaches the table

(d) Box is pushed over the table

(e) Boxes push each other

(f) Boxes fall to the ground and are transported back

Figure 7.10: Snapshots of the GDP animation.

the constraints multiple times before a frame is drawn. This is not necessary or desired. Instead, a better strategy would be to save up state-change indications and trigger constraint solving just before rendering is done. This option implies that the SCAFFOLD library needs to communicate with the GDP, since it has to know the rendering is at hand. Consequently, this requires an adaptation of either the GDP or the SCAFFOLD system.

To avoid this, we choose for the first option. It is the responsibility of the Looks programmer to call the `solve_network()` method of the constraint manager before a frame is rendered. It has a slight disadvantage, since the programmer has to take care of it, and has to be cautious not to trigger solving multiple times before a frame is drawn. On the other hand, there is the advantage that the programmer has more control over the solving process. For example, he can choose to call `solve_network()` once every five frames.

Currently, the VD-CHE is added to Looks, via the standard way of coupling a C++ library to the GDP. Integrating the constraint handling engine more with the Looks language could free the Looks programmer from creating a constraint manager himself. Also, special construct could be created in Looks that would trigger constraint solving in a way that would be transparent to the programmer.

In Figure 7.10, some snapshots of an animation are taken. Boxes fall from a certain point in space towards a table. One box which is already on the table makes an oscillating movement. The falling and oscillating movements are implemented using functionality of the GDP system. Boxes that land on the table are pushed by the oscillating box towards the edge of the table. When pushed beyond the edge, the boxes fall to the ground. Once reached the ground, each box is transported back to its original position after which it starts falling again. To prevent that boxes intersect the table or each other, and to model movement of objects pushing each other, constraint types are used of the VD-CHE.

7.5.3 Evaluation by Users

Within the VR-DIS project and the GDP project, the constraint system SCAFFOLD is used for solving constraints. In this section, we present the opinion of the users of the constraint system.

VR-DIS

Within the VR-DIS project, there are two users. The first one is the architect who used the VR-DIS Behavior Prototype to investigate constraints in the design environment. The second one the software engineer who implemented the VR-DIS Behavior Prototype. This person is called the developer of the object-oriented application in the **CODE** context. The software engineer used the SCAFFOLD system to solve constraints that are imposed by the architect on the elements. The conclusions of the architect concern observations related to the applicability of constraints in a design environment. The conclusions of the software engineer have to do with the usage of SCAFFOLD as a software library.

For the architect, geometric constraints are a powerful concept for capturing specific design knowledge. However, several limitations of the current prototype system have been observed. In the described geometric constraint solving mechanism the bounding box plays a central role. Although many building elements fit a bounding box quite well (for example, wall and floor), it imposes limitations on architectural design. For example, elements like walls, floors can only meet perpendicular, curved walls, roofs cannot be represented adequately. Thus, bounding boxes suit well for an important category of building elements, but for non-rectangular shapes, a different strategy must be developed.

The software engineer developed the prototype system that is used by the architect. The graphical user interface of the prototype system does not allow yet the direct creation and manipulation of design

constraints. A graphical representation of constraints gives the user better control of the design and its behavior. Widgets should allow the user to interactively attach and detach constraints appropriate for the building element at hand.

The main strength of the constraint system according to the software engineer was the well-designed structure of the software library. The library has been 'plugged-in' in several other small applications for testing purposes. For these test applications, no changes were necessary with respect to the constraint library. Based on this fact, it was relatively easy to create new elements (besides the bounding box classes) that can be used in architectural design.

A disadvantage of the constraint system is that the performance of the prototype systems decreases rapidly with an increasing number of constraints. The complete set of equations is solved each time when a system event that signals a change in the position of one of the bounding boxes is raised. The constraint solving algorithm could be optimized by recalculating only local changes.

GDP

The users in the GDP project are the developers of animation scripts in Looks. For a Looks programmer, the main requirement is whether constraints enable him to write less code. According to the programmer of the demonstration animation this was the case. The advantage was that constraints can be specified in a static manner, while in the main loop of the animation only the motions of the objects have to be prescribed.

The steps to arrive at a constraint-driven animation in Looks include the following.

1. Create a constraint manager.
2. Create the objects on which constraints have to be imposed.
3. Create the constraint objects themselves.
4. Initialize each constraint with the operands on which it is imposed.
5. Add the constraints to the constraint manager.
6. Trigger the constraint manager at the end of each frame to invoke constraint solving.

Currently, the first and the last step have to be carried out by the Looks programmer. However, they can also be made part of the GDP animation system so that they are performed automatically. Constraints can be added and removed dynamically from the constraint manager.

The Looks programmer was able to create his own constrainable and constraint objects by simply inheriting from the constraint classes that were provided in Looks. He was not able to write constraint solvers in Looks. The solvers only reside in the constraint system and are invisible to the Looks programmer. This was experienced as a positive point, since the programmer was not drawn into solver aspects.

The disadvantages of the constraint library were similar to the ones in the VR-DIS project. The provided bounding box classes were too limited to model a complex animation with various objects. Furthermore, as more constraints were added the animation would become slow. In an animation in which about 30 constraints were applied on 10 objects, the frame rate would drop to 5 to 10 frames per second.

The coupling of Looks and the constraint system was done by a software engineer in the GDP project. His task was to create an interface that would enable a Looks programmer to communicate with the SCAFFOLD library. His experience with SCAFFOLD were again similar to the software engineer in the VR-DIS project. The modular design enabled him to smoothly integrate Looks and SCAFFOLD. Updates and changes in the SCAFFOLD system itself had no effect on the created interface.

7.6 Conclusion

In this chapter, we described the SCAFFOLD system, the constraint handling engine VD-CHE, and two applications that use the engine. SCAFFOLD implements the functionality that is described in the **CODE** model. Based on SCAFFOLD, the VD-CHE provides constrainable and constraint types and solving mechanism. Two OO-applications, the VR-DIS Behavior Prototype and Looks, use the classes of the VD-CHE to apply constraints.

In contrast with the PROTOM implementation, SCAFFOLD and and the VD-CHE implemented the model **CODE** without the help of an extra language or mechanism to bring about the separation between the OO-application and the constraint world. As can be seen in this chapter, in both OO-applications there is still a strong separation between the OO-application and the VD-CHE library enabling separate development and updates of systems. Both the VR-DIS Behavior Prototype and Looks use exactly the same VD-CHE library.

From evaluation with users and software designers in the VR-DIS project and the GDP project, it can be concluded that the main advantage of the SCAFFOLD library is its modularity and flexibility for using it in a wide range of environments. SCAFFOLD owes its modularity to the fact that it is based on the **CODE** model. **CODE** prescribes the separation of the constraint world from the object world and identifies the elements that occur in an implementation.

The disadvantage of the SCAFFOLD library was its loss of performance when a large number of constraints had to be solved. This is due to the fact that the numeric constraint solver of the VD-CHE can be implemented in a more efficient way. First of all, the numerical method that is applied does not exploit all characteristics of the set of equations that have to be solved. Secondly, the constraint manager solves all constraints at once without exploring the possibility of calculating a local solution. However, due to the setup of SCAFFOLD, it is a relatively easy job to replace the numerical solver or the constraint manager by a more efficient one.

The main goal of **CODE** and of SCAFFOLD was to assist software engineers in creating object-oriented software systems that incorporate constraints. In the two application implementations in the VR-DIS project and the GDP project, it has been demonstrated that it is possible to let an object-oriented component cooperate with a constraint component by separating both paradigms.

Chapter 8

Conclusions

8.1 Design Process

As part of a designer's Ph.D. dissertation, we devote this last chapter to an evaluation of the design process that was carried out during the project. In Chapter 1, an outline was given of the software engineering process that was followed during the project. This was the following:

1. Domain analysis (Chapter 2),
2. Problem analysis (Chapter 3),
3. Outline of a solution (Chapter 4),
4. Specification of the solution (Chapter 5),
5. Prototype implementation (Chapter 6),
6. Application implementation (Chapter 7).

In the engineering discipline of software development, several process models exist that describe how to go through the different phases to develop a software product (see [Sommerville, 1989], [Winograd et al., 1996], [Jacobson, 1995], [Boehm, 1986]). Among the most familiar are the *waterfall approach*, *prototyping*, and the *spiral model*. The waterfall approach is the oldest one which was developed as a reaction to the software crisis in the late 1960s. The software engineering process, or life-cycle, is viewed as a number of phases, such as requirements definition, specification, design, implementation, and so on. The initial idea of this model was to end one phase before starting the next one. However, this principle was abolished relatively quickly and overlap of phases and iterations between subsequent phases were incorporated. The waterfall approach is still one of the most widely applied models. It is especially useful for management purposes to plan and estimate development costs, although the actual software engineering process in reality is often executed differently.

The prototyping approach involves developing a working system, the *prototype*, as quickly as possible, and modifying it until it performs in an adequate way. Based on the prototype, the system requirements are determined and the software is re-implemented for the final product. This approach is especially useful in the development of systems for which detailed requirements are difficult to specify, for example, user-interfaces. A disadvantage of this approach is that the prototypes are not always discarded once the requirements are determined. In practice, existing code of the prototype implementation is frequently used in the implementation of the final product. Since prototype implementations are usually not set up using sound software engineering principles, the final product often suffers from the anomalies that were introduced by the prototype. These anomalies, such as bugs, tricks, and hacks, and most of them usually undocumented, are hard to isolate and they can cause costly problems throughout the lifetime of the software product.

The spiral model overcomes some of the problems of the previous two models. It describes the different phases that are performed for the development of an initial product, and subsequent versions, using a waterfall model for each phase. It is especially intended to help in managing risks. In the beginning, only the highest priority features are defined and implemented. Next, feedback from users or customers is collected in order to develop the next version. The model describes how subsequent versions can be developed incrementally from a prototype to a complete product. The model owes the name to its graphical representation in which the subsequent phases are depicted in a spiral. Each time one round in the spiral is completed, a version of the product is finished.

For the development of the constraint system for the VR-DIS project and the GDP project, the prototyping approach was taken. This approach was chosen because the requirements for the constraint system were not clear initially. The projects in which the system was going to be used were themselves experimental research environments. Furthermore, the constraint system had to combine two different programming paradigms that conceptually conflicted with each other. Through the use of prototypes, a better insight could be gained in the development of a final system.

The prototyping approach has proven to be an appropriate way to perform the research. The first prototype implementation, called PROTOM, was built using the MANIFOLD language (see Chapter 6). During the creation of this implementation, insight was gained regarding the implementation aspects of the model. The prototype was discarded after it was finished, and the knowledge gained was taken to implement a new class structure for the final constraint system. This resulted in a constraint system called SCAFFOLD which had a clear structure in accordance with the conceptual model and enabled to combine object-oriented programming with constraint programming. Moreover, SCAFFOLD resulted in a software library that could easily be used and plugged-in into different systems, such as the design environment of the VR-DIS project and the animation system in the GDP project.

In Figure 8.1, an overview of the activities is given on an approximate time scale. The phases do not follow each other in a sequential order, but overlap. The overlap occurs because clear borders between the phases are hard to identify and iterations to previous phases were often necessary. At the end of the fourth year, part of the time was spent to write the Ph.D. thesis, which is not depicted in the figure.

Although the borders are not precisely demarcated, we can give a good indication of when a certain activity started to receive attention and when it was stopped. The problem analysis was started while analyzing the domain. When the problem analysis was nearing its end, the design of the conceptual model was already unfolding. At a certain point in time, the design, specification, and prototype implementation of the conceptual model were executed in parallel. As experience was gained due to the prototype implementation, iterations to previous phases were done to incorporate the new insights in the design and the specification. Figure 8.1 also shows that the specification still continued after the prototype implementation was finished. In this period, the experience gained by the prototype implementation were worked out to arrive at a final version of the conceptual model. After the conceptual model had crystallized into its final shape at the end of phases 3, 4, and 5, the practical implementation was started as a relatively separated activity.

In the figure, the sizes of the boxes give a rough indication of the effort that was spent on each activity. The approximate distribution of the effort over the different phases is given in Table 8.1.

1. The domain analysis (Chapter 2) gave an overview of the two background disciplines, object-oriented programming and constraint programming. In both the VR-DIS project and the GDP project, applications (the virtual reality design environment and the animation system) are built using object-oriented technologies. For these applications, a constraint system was needed that would enable to impose constraints on the objects.

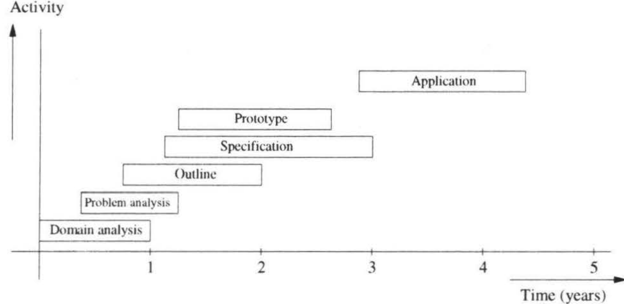


Figure 8.1: Global overview of the six project phases.

2. After this overview, the problem was described and analyzed (Chapter 3). The usefulness of objects and constraints in computer graphics justifies the investigation of combining these two approaches for the development of graphics software systems. However, two conceptual incompatibilities were encountered which obstruct a smooth and straightforward integration of the two paradigms. These incompatibilities were worked out and an initial solution was indicated.
   Already in this stage, a small prototype implementation had been built to experiment with the basic concepts of the conceptual model. This implementation is not mentioned in this thesis. However, it was used for demonstration purposes in [Veltkamp et al., 1996].
3. In the outline phase of the project (Chapter 4), a solution was developed that proposes a radical separation of the constraint system and the object-oriented framework. This is done by means of two orthogonal communication strategies for objects, messages on the one hand, and events and data-flows on the other hand. The solution is described in form of a conceptual model which identifies the entities and describes the communication among the entities. The entity types are constrainables, constraints, solvers, and managers. They communicate via events and data flows.
4. In the specification step (Chapter 5), the conceptual model was further worked out in detail. The model is named **CODE** and consists of the elements described in the outline.
   Commonly known notational techniques were used to specify the characteristics of the model. State charts are used to describe the modes that an entity can be in. A data flow diagram is used to describe the data exchange among the entity types. Event tables describe the sensitivity of entities for events.
5. To verify the conceptual model and to arrive at a system in which all software components are distinguished, a prototype implementation was built that resembles the **CODE** model as closely as possible (Chapter 6).
   The communication structure was built in a concurrent environment using the **MANIFOLD** language. The implementation consisted of an application for manipulating geometrical objects that allows constraints to be imposed on the objects. Several types of constraints are provided, such as `touch`, `equal_area`, that are solved by a local propagation solver and a cycle solver.
   The prototype implementation demonstrated the feasibility of the conceptual model and provided the "blueprint" for the application implementation.
6. In the final stage of the engineering process (Chapter 7), the constraint system SCAFFOLD was built, based on the **CODE** model and the experience with the prototype implementation. Spe-

Phase	*Chapter*	*Estimated effort*
Domain analysis	2	14 %
Problem analysis	3	12 %
Outline conceptual model	4	12 %
Specification conceptual model	5	17 %
Prototype implementation	6	20 %
Application implementation	7	25 %

Table 8.1: Estimates for the various design activities.

cific constraint types were developed in cooperation with the VR-DIS project group. These constraint types could be used to specify `touch`, `distance`, and other constraints among architectural elements. For the GDP project, SCAFFOLD was integrated with the Looks language, allowing Looks programmers to use and create their own constraint types.

8.2 Conclusions

The goal of the Ph.D. project was to build a constraint system for the GDP project and the VR-DIS project. In both projects, object-oriented systems had been developed or were under development that had to be combined with constraint technology. This resulted in a conceptual model **CODE** upon which the constraint system SCAFFOLD was built. Based on the experience with these systems, we draw the following main conclusions.

- The conceptual model demonstrates that radical separation of object-oriented programming and constraint programming is an effective way for combining the two paradigms.
  CODE separates a system in the object-oriented application module and the constraint system module. The communication between the two modules is described by means of events and data flows. The advantage of using events and data flows is that this communication strategy does not interfere with the message passing communication between objects. In the model, events are used to notify the constraint system of changes in the object-oriented application and data flows are used to access internal variables of objects directly.
  The implementations of the model clearly demonstrate the detachment between the object-oriented application and constraint system and the feasibility of implementing the functionality of the events and data flows.
- The strict definition of the entity types *constrainables*, *constraints*, *solvers*, and *managers* in **CODE** allows the definition of constraint types independently of the solvers or solving techniques that have to satisfy them. The constraints define only the relation that has to hold without maintaining information about how to solve them. The "solving" information is kept by the solver or manager that solves a network of constraints.
  The advantage is that constraints can be solved in different ways depending on various factors that can be surveyed by a constraint manager. For example, in a network which is solved by local propagation, a certain constraint is solved in such a way that only those variables are updated which have not been updated before by any other solver. Solving techniques that are discussed in the literature often maintain solving information within the constraints [Sannella, 1994] or

even within the objects themselves [Hoole et al., 1994]. This makes it more difficult to upgrade solvers or to experiment with different global solving techniques.

- The implementations of **CODE** show that multiple solvers can be made to cooperate to solve one network of constraints. This is desired if for a certain constraint problem there is not a single sufficiently powerful solver. When multiple solvers are present, each solver can work on a part of the network that contains a specific structure or that contains a specific type of constraints. Each solver can apply an efficient method to satisfy the constraints. The cooperating solvers are controlled by a manager which maintains the network. In literature, the benefits of solver cooperation have well been recognized. For example, in [Sannella, 1994], the SkyBlue solver cooperates with dedicated solvers to deal with cycles in the network. Also in the field of constraint logic programming, research is done regarding cooperation of multiple solvers (see [Kirchner et al., 1994], [Monfroy, 1996]).
- The **CODE** model and the implementations demonstrate the feasibility of modular design when combining objects and constraints. The use of constraints enabled the software engineers to specify relations among many objects in a declarative manner. This declarative style freed him from writing out these relations himself. Moreover, the maintenance of the relations was outside his scope. As a result less code had to be written, which resulted also in less errors that were made. Furthermore, in the implementations, the systems are clearly decomposed into *modules* that are designed and can be revised independently and have well-defined interfaces. This modularity enables software engineers to better design, implement, and maintain software systems.

8.3 Further Work

Based on the conclusions of the previous section, we suggest the following subjects for further research.

- To develop proofs that justify the design of the conceptual model. Such proofs can either be in terms of theoretical design principles or in terms of hypotheses and well-formulated experiments. They can show on a theoretical level that, for example, fewer lines of code were written under some controlled situation or that users were able to tackle complex tasks with greater speed.
- To develop a variety of satisfaction techniques that exploit specific features of constraint networks. Although developed in a computer graphics context, the **CODE** model is very general. However, different constraint domains give rise to specific constraint types, solving techniques, and structures of networks.

  For example, in the area of user interface design, linear inequality constraints are frequently used to express relations for the specification of the relative position of a certain object (for example, left, right, or above) with regard to another object [Borning et al., 1997]. In computer animation, constraints are often solved using kinematics or dynamics [Barenbrug, 1999]. Geometric modeling often deals with non-linear constraints [Kramer, 1992]. In multimedia, constraints are used to guarantee quality of service, such as coherency among the presented data, restrictions on the validity of data, and so on [Hintum, 1997].

  Sub-models of **CODE** can be developed that provide specific solving techniques for a certain application area. For example, specific solver-manager structures can be developed for for the combination of the local propagation algorithm with degrees of freedom propagation. Or constraint managers and solvers can be developed that deal with specific cyclic structures in the

network. A third option is to develop managers that can transform a constraint network into a *higher-order* constraint network (see below).

- As a specific instance of developing generic constraint satisfaction techniques, extensions to the conceptual model can be developed to explicitly define *higher-order constraints* and *meta-constraints*. Higher-order constraints impose conditions on other constraints and offer possibilities to deal with large and complex constraint networks. Meta-constraints put restrictions on the solving process.
  In cases that allow a constraint network to be decomposed into subproblems, the global structure of the network can be expressed in a so-called higher-order network. The structure of that network can be exploited to determine a global solving strategy while the solution of subproblems is treated in isolation. An overview of techniques that exploit structures in a network is given in [Freuder, 1994]. Meta-constraints can define restrictions on the constraint satisfaction mechanism which allow to select from different solvers.
  In environments where the resources are scarce and the variety of constraint variables and types is large (for example, in multimedia environments), higher-order constraints and meta-constraints solvers can improve efficient constraint solving. In [Hintum, 1997], a constraint-based quality-of-service management system is developed for a multimedia environment. By dynamically servicing requests for resources using constraints, it provides a specific kind of higher-order constraints and meta-constraints to fulfill the requests.
- To develop a software engineering method based on the conceptual model. In **CODE**, the design activities that a designer of an object-constraint application has to perform are under-exposed. **CODE** describes the separation of the constraint and object world and the communication between the two.
  However, it does not explicitly describe a phased methodology which treats the different steps in the design process of combined OO and constraint solving systems. Although design is an inherently "messy" process [Winograd et al., 1996], phasing could aid in the efficient development of applications based on **CODE**. For example, ideally, the identification of constrainable and constraint types should go before developing solver and manager types. In order for **CODE** to incorporate also a design method, tools and techniques have to be provided to structure the design process.

Appendix A

Abbreviations

API	Application Programmer's Interface
CH-API	Constraint Handler API (of PROTOM)
CODE	Constraints on Objects via Data flows and Events
GA-CHE	Graphics Application - Constraint Handling Engine
GA-CI	Graphics Application - Class Interface
GDP	Generalized Display Processor
IWIM	Idealized Worker Idealized Manager
OO-API	Object-Oriented API (of PROTOM)
PROTOM	Prototype Implementation
SCAFFOLD	Solver for Constraints And Fabric For Object-oriented Library Design
VD-CHE	VR-DIS Constraint Handling Engine
VR-DIS	Virtual Reality - Design Information System

Bibliography

[Akin, 1986] Oemer Akin. *Psychology of architectural Design*. Pion, ISBN 0-85086-120-9, 1986.

[Aksit et al., 1992] M. Aksit and L. Bergmans. Obstacles in Object-Oriented Software Development. *Proceedings OOPSLA '92*, ACM SIGPPLAN Notices, Vol. 27, No. 10, October 1992, pages 341–358.

[Aksit et al., 1993] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object-Interactions Using Composition-Filters. *Object-based distributed processing*, R. Guerraoui, O. Nierstrasz, and M. Riveill (editors), LNCS, Springer-Verlag, 1993, pages 152-184.

[Allen, 1981] J. F. Allen. An interval based representation of temporal knowledge. *Proceedings of the seventh International Joint Conference on Artificial Intelligence*, August 1981, pages 221–226.

[Apt, 1997] Krzysztof R. Apt. *From Logic Programming to Prolog*. Prentice Hall, ISBN 0-13-230368-X, 1997.

[Arbab, 1996] Farhad Arbab. The IWIM model for coordination of concurrent activities. *Coordination Languages and Models*, Lecture Notes in Computer Science, volume 1061, Springer, 1996, pages 34–56.

[Arbab, 1998] Farhad Arbab. MANIFOLD 2.0 Reference Manual.
URL: http://www.cwi.nl/ftp/manifold/refman.ps.Z

[Arbab et al., 1991] F. Arbab and B. Wang. A geometric constraint management system in Oar. *Intelligent CAD Systems III – Practical Experience and Evaluation*, Springer-Verlag, 1991, pages 231–252.

[Arbab et al., 1993] F. Arbab, I. Herman, and P. Spilling. An overview of MANIFOLD and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, February 1993.

[Argante, 1998] Erco Argante. *CoCa: a model for parallelization of high energy physics software*. Designer's Ph.D. thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1998.

[Baase, 1978] Sara Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, ISBN 0-201-06035-3, 1978.

[Barenbrug, 1999] Bart Barenbrug. *Designing a class library for interactive simulation of rigid body dynamics*. Designer's Ph.D. thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1999 (To appear).

[Bax, 1989] M.F.Th. Bax. Structuring Architectural Design Processes. *Open House International*, vol. 14, no. 3, 1989, pages 20–27.

[Bergstra et al., 1998] Jan A. Bergstra and Paul Klint. The Discrete Time TOOLBUS - A Software Coordination Architecture. *Science of Computer Programming*, 31(2-3): 205-229, 1998.

[Boehm, 1986] B.W. Boehm. A spiral model of software development and enhancement. *Software Engineering Notes*, 11(4), 1986.

[Blake et al., 1992] Edwin H. Blake and Quinton Hoole. Expressing Relations Between Objects: Problems and Solutions. *Proceedings of the Third Eurographics Workshop on Object-Oriented Graphics*, Chambery, Switzerland, 1992.

[Booch, 1986] Grady Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, Vol. 12, No. 2, February 1986, pages 211–221.

[Booch, 1991] Grady Booch. *Object-Oriented Design*. The Benjamin/Cummings Publishing Company, Inc., ISBN 0-8053-0091-0, 1991.

[Borning, 1981] A. Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 4, October 1981, pages 353–387.

[Borning, 1986] A. Borning. Classes versus Prototypes in Object-Oriented Languages. *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, Dallas, Texas, November 1986, pages 36–40.

[Borning et al., 1986] Alan Borning and Robert Duisberg. Constraint-Based Tools for Building User Interfaces. *ACM Transactions on Graphics*, Vol. 5, No. 4, October 1986, pages 345–374.

[Borning et al., 1997] Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving Linear Arithmetic Constraints for User Interface Applications. *Proceedings of the 1997 ACM Symposium on User Interface Software and Technology*, October 1997, pages 87–96.

[Budd, 1997] Timothy Budd. *An Introduction to Object-Oriented Programming*. Second Edition, Addison-Wesley, ISBN 0-201-82419-1, 1997.

[Choppy et al., 1989] C. Choppy, S. Kaplan, and M. Soria. Complexity analysis of term-rewriting systems. *Theoretical Computer Science*, 67(2/3):261–282, 1989.

[Clark et al., 1986] E.M. Clark, E.A. Emerson, A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *TOPLAS 1986*, 1986.

[Colmerauer et al., 1973] A. Colmerauer, H. Kanoui, R. Pasero, P. Roussel. *Un Système de Communication Homme-machine en Français*. Technical Report, Groupe Intelligence Artificielle, Université d'Aix-Marseille II, 1973.

[Cournarie et al., 1995] E. Cournarie and M. Beaudouin-Lafon. Alien: a prototype-based constraint system. [Laffra et al., 1995], pages 92–110.

[Davis, 1987] Ernest Davis. Contraint Propagation with Interval Labels. *AI*, 32:281–331, 1987.

[Davy, 1991] Jacques Davy. Go, a graphical and interactive C++ toolkit for application data presentation and editing. *Proceedings 5th Annual Technical Conference on the X Window System*, 1991.

[Dechter et al., 1988] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *AI*, 34:1–38, 1988.

[Dohmen, 1998] Maurice H. P. J. Dohmen. *Constraint-Based Feature Validation*. Ph.D. thesis, Delft University of Technology, ISBN 90-9011285-5, 1998.

[Donikian et al., 1995] Stéphane Donikian and Gérard Hégron. Constraint Management in a Declarative Design Method for 3D Scene Sketch Modeling. *Principles and Practice of Constraint Programming*, V. Saraswat et P. Van Hentenryck (editors), MIT Press, 1995, pages 427–443.

[Fletcher, 1987] R. Fletcher. *Practical Methods of Optimization*. Second Edition, John Wiley and Sons, ISBN 0-471-91547-5, 1987.

[Fowler, 1997] Martin Fowler. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, ISBN 0-201-32563-2, 1997. (The Addison-Wesley object technology series)

[Freeman-Benson, 1991] Bjorn N. Freeman-Benson. *Constraint Imperative Programming*. Ph.D. thesis, University of Washington, Department of Computer Science & Engineering, 1991. Published as technical report UW-CSE-91-07-02.

[Freeman-Benson et al., 1990] Bjorn N. Freeman-Benson, John Malony, and Alan Borning. An Incremental Constraint Solver. *Communications of the ACM*, Volume 33, Number 1, January, 1990.

[Freeman-Benson et al., 1992] Bjorn N. Freeman-Benson and Alan Borning. Integrating constraints with an object-oriented language. *Proceedings ECOOP'92–European Conference on Object-Oriented Programming, Utrecht, 1992*, O. Lehrmann Madsen (editor), Lecture Notes in Computer Science 615, Springer-Verlag, 1992, pages 268–286.

[Freeman-Benson et al., 1992] Bjorn N. Freeman-Benson and Alan Borning. The Design and Implementation of Kaleidoscope'90, a Constraint Imperative Programming Language. *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, 1992, pages 174–180.

[Freuder, 1982] E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, Volume 29, Number 1, 1990, pages 24–32.

[Freuder, 1994] E.C. Freuder. Exploiting structure in constraint satisfaction problems. [Mayoh et al., 1994], pages 51–74.

[Gamma et al., 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, ISBN 0-201-63361-2, 1995.

[Ginsberg et al., 1990] M.L. Ginsberg and W.D. Harley. Iterative Broadening. *Proceedings National Conference on Artificial Intelligence (AAAI)*, 1990, pages 216–220.

[Gleicher, 1994] M. Gleicher. Practical Issues in Graphical Constraints. *Principles and Practice of Constraint Programming*, V. Saraswat and P. Van Hentenryck (editors), MIT Press, 1994.

[Gleicher et al., 1994] M. Gleicher and A. Witkin. Drawing With Constraints. *The Visual Computer*, 11(1):39–51, 1994.

[Goldberg et al., 1983] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Güsgen et al., 1988] Hans-Werner Güsgen and Joachim Hertzberg. Some fundamental properties of local constraint propagation. *AI*, 36:237–247, 1988.

[Harel, 1987] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Prog. 8*, 1987, pages 231–274. (Preliminary version appeared as Technical Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, February 1984.)

[Hautus, 1997] Edwin Hautus. *Sybar, a Human Motion Analysis System for Rehabilitation Medicine*. Designer's Ph.D. thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1997.

[Heintze et al., 1987] N. Heintze, S. Michaylov, P. Stuckey. CLP($\Re$) and Some Electrical Engineering Problems. *Proceedings of the 4th International Conference on Logic Programming*, MIT Press, 1987.

[Hentenryck, 1989] Pascal van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, London, ISBN 0-262-08181-4, 1989.

[Herman et al., 1992] I. Herman and F. Arbab. More Examples in Manifold. Technical Report CS-R9214 1992, Centrum voor Wiskunde en Informatica (CWI), May 1992. *URL:* ftp://ftp.cwi.nl/pub/CWIreports/IS/CS-R9214.ps.Z

[Hintum, 1997] J.E.A. van Hintum. *Quality Constraints & Constrained Quality*. Ph.D. thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1997.

[Hoole et al., 1994] Quinton Hoole and Edwin Blake. OOCS–constraints in an object oriented environment. *Proceedings 4th Eurographics Workshop on Object-Oriented Graphics, Sintra, Portugal*, May 1994, pages 215–230, 9–11.

[Horn, 1991] Bruce Horn. *Siri: A constrained-object language for reactive program implementation*. Technical Report CMU-CS-91-152, Carnegie Mellon University, School of Computer Science, June 1991. *URL:* http://www.cs.cmu.edu/Reports/1991.html.

[Jacobson, 1995] Ivar Jacobson. *Object-oriented software engineering: a use case driven approach*. Revised 4th print, ACM Press Books, New York, ISBN 0-201-54435-0, 1993.

[Jaffar et al., 1987] J. Jaffar and J.L. Lassez. Constraint Logic Programming. *Proceedings POPL '87*, Munich, January 1987, pages 111–119.

[Kelleners et al., 1997] R.H.M.C. Kelleners, R.C. Veltkamp, and E.H. Blake. Constraints on Objects: a Conceptual Model and an Implementation. *Proceedings of the 6th Eurographics Workshop on Programming Paradigms in Graphics*, Budapest, September 1997.

[Kenneth et al., 1986] Kenneth Kahn, Eric Dean Tribble, Mark Miller, Daniel Bobrow. Objects in concurrent logic programming languages. *Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. Portland, Oregon, 1986.

[Kirchner et al., 1994] Hélène Kirchner and Christophe Ringeissen. Combining Symbolic Constraint Solvers on Algebraic Domains. *J. Symbolic Computation*, 18(2):113–155, 1994.

[Knudsen et al., 1994] J.L. Knudsen and J. Middelfart. Supporting Constraints in BETA. *Proceedings of the Nordic Workshop on Programming Environment Research*, Lund, Sweden, June 1994.

[Kowalski, 1974] R. Kowalski. Predicate Logic as Programming Language. *Proceedings of the IFIP Congress 74*, North Holland (editor), pages 569–574, 1974.

[Kramer, 1992] Glenn A. Kramer. *Solving Geometric Constraint Systems. A Case Study in Kinematics*. The MIT Press, ISBN 0-262-11164-0, 1992.

[Laffra et al., 1991] Chris Laffra and Jan van den Bos. Propagators and concurrent constraints. *OOPS Messenger*, 2(2):68–72, April 1991.

[Laffra et al., 1995] C. Laffra, E. H. Blake, V. de Mey, and X. Pintado, editors. *Object-Oriented Programming for Graphics*. Focus on Computer Graphics, Springer, 1995.

[Leeuwen et al., 1998] J.P. Leeuwen and H. Wagter. A Features Framework for Architectural Information, dynamic models for design. *Proceedings of Artificial Intelligence in Design '98*, J.S. Gero and F. Sudweeks (editors), Lisbon, Portugal, July 1998, pages 20–23. Kluwer Academic Publishers, Dordrecht.

[Leler, 1988] W. Leler. *Constraint Programming Languages, Their Specification and Generation*. Addison-Wesley, ISBN 0-201-06243-7, 1988.

[Li, 1995] Jiarong Li. *Object-oriented constraint programming for interactive applications*. Ph.D. thesis, Royal Institute of Technology, Department of Numerical Analysis and Computing Science, Sweden, 1995.

[Lipton, 1995] Richard J. Lipton. DNA Solution of Hard Comptutaional Problems. *Science*, vol. 268, 28 April 1995, pages 542–545.

[Lizkov et al., 1974] B. Lizkov and S. Zilles. Programming with abstract data types. *ACM Sigplan Notices*, 9(4):50–59, 1974.

[Lopez, 1997] Gustavo Lopez. *The design and implementation of Kaleidoscope, a constraint imperative programming language*. Ph.D. thesis, University of Washington, Department of Computer Science & Engineering, 1997. Published as technical report UW-CSE-97-04-08.

[Lopez et al., 1994] Gus Lopez, Bjorn Freeman-Benson, Alan Borning. Implementing Constraint Imperative Programming Languages: the Kaleidoscope'93 Virtual Machine. *Proceedings of the 1994 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, October 1994, pages 259–271.

[Luenberger, 1984] David G. Luenberger. *Linear and Nonlinear Programming*. Second Edition, Addison-Wesley, ISBN 0-201-15794-2, 1984.

[Mackworth, 1977] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8, 1977, 99–118.

[Mackworth et al., 1985] A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25, 1985, 65–74.

[Malony et al., 1989] John Malony, Alan Borning, and Bjorn Freeman-Benson. Constraint Technology for User-Interface Construction in ThingLab II. *Proceedings of the 1989 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, New Orleans, 1989.

[Mayoh et al., 1994] B. Mayoh, E. Tyugu, and J. Penjam (editors). *Proceedings NATO ASI on Constraint Programming*. LNCS ASI Series F: Computer and System Sciences, Vol 131. Springer-Verlag, 1994.

[Meyer, 1992] Bertrand Meyer. *Eiffel: the language*. Prentice Hall, London, ISBN 0-13-247925-7, 1992. (Prentice Hall object-oriented series)

[Meyer, 1997] Bertrand Meyer. *Object-oriented software construction*. Second edition, Prentice-Hall, London, ISBN 0-13-629155-4, 1997.

[Micallef, 1988] J. Micallef. Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages. *JOOP*, April/May 1988, pages 12–35.

[Mohr et al., 1986] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *AI*, 28:225–233, 1986.

[Monfroy, 1996] Eric Monfroy. *Solver Collaboration for Constraint Logic Programming*. Ph.D. thesis, Université Henry Poincaré – Nancy I, Formation Doctorale en Informatique, 1996.

[Myers et al., 1990] B.A. Myers, D.A. Giuse, R. B. Dannenberg, B. Vander Zanden, D.S. Kosbie, E. Pervin, A. Mickish, and P. Marshal. Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. *IEEE Computer*, 23(11):71–85, November 1990.

[Noort et al., 1997] A. Noort, M. Dohmen, and W.F. Bronsvoort. Solving over- and underconstrained geometric models. *Proceedings of the Workshop on Geometric Constraint Solving and Applications*, Ilmenau, 1997.

[Parnas, 1972] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(2):1053–1058, 1972.

[Peeters, 1995] Eric Peeters. *Design of an Object-Oriented, Interactive Animation System*. Ph.D. thesis, Eindhoven University of Technology, Mathematics and Computing Science, 1995.

[Press et al., 1992] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery. *Numerical recipes in C: the art of scientific computing*. Second edition, Cambridge University Press, Cambridge, ISBN 0-521-43108-5, 1992.

[Rankin, 1991] John R. Rankin. A graphics object oriented constraint solver. [Laffra et al., 1995], pages 71–91.

[Robinson, 1965] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Roozenburg et al., 1995] N.F.M. Roozenburg and J. Eekels. *Produktontwerpen, structuur en methoden*. Lema, Utrecht, ISBN 90-5189-067-2, 1995.

[Rumbaugh et al., 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented modeling and design*. Prentice Hall, ISBN 0-13-630054-5, 1991.

[Sannella, 1994] Michael Sannella. *Constraint Satisfaction and Debugging for Interactive User Interfaces*. Ph.D. thesis, University of Washington, Department of Computer Science & Engineering, 1994. Published as technical report UW-CSE-94-09-10.

[Saraswat, 1993] Vijay A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, ISBN 0-262-19297-7, 1993.

[Shah et al., 1995] J. J. Shah and M. Mäntylä. *Parametric and Feature Based CAD/CAM*. John Wiley & Sons, Inc., ISBN 0-471-00214-3, 1995.

[Sommerville, 1989] Ian Sommerville. *Software Engineering*. Third edition, Addison-Wesley, ISBN 0-201-17568-1, 1989.

[Stroustrup, 1997] Bjarne Stroustrup. *The C++ Programming Language*. Third edition, Addison-Wesley, ISBN 0-201-88954-4, 1997.

[Sutherland, 1963] Ivan E. Sutherland. Sketchpad: A man-machine graphical communication system. *Proceedings of the Spring Joint Computer Conference*, Detroit, Michigan, May 21–23 1963, pages 329–345, AFIPS Press, 1963.

[Taivalsaari, 1996] Antero Taivalsaari. On the Notion of Inheritance. *ACM Computing Surveys*, Vol. 28, No. 3, September 1996, pages 438–479.

[Tsang, 1993] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, ISBN 0-12-701610-4, 1993.

[Veltkamp, 1995] Remco C. Veltkamp. A Quantum Approach to Geometric Constraint Satisfaction. [Laffra et al., 1995], pages 54–70.

[Veltkamp et al., 1992] R.C. Veltkamp and F. Arbab. Geometric constraint propagation with quantum labels. *Proceedings of the Eurographics Workshop on Computer Graphics and Mathematics*, Springer-Verlag, 1992.

[Veltkamp et al., 1995] Remco C. Veltkamp and Richard H.M.C. Kelleners. Information hiding and the complexity of constraint satisfaction. *Programming Paradigms in Graphics*, Remco C. Veltkamp and Edwin H. Blake (editors), Springer-Verlag, 1995, pages 49–66.

[Veltkamp et al., 1996] Remco C. Veltkamp and Edwin H. Blake. Event-based.constraints: coordinate.satisfaction -> object.state. *Focus on Computer Graphics*, Springer, 1996, pages 159–169.

[Warmer et al., 1999] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, ISBN 0-201-37940-6, 1999. (The Addison-Wesley object technology series)

[Wegner, 1989] Peter Wegner. Concepts and Paradigms of Object-Oriented Programming. *Expansion of Oct 4 OOPSLA-89 Keynote Talk.*

[Wesselink et al., 1995] Wieger Wesselink and Remco C. Veltkamp. Interactive Design of Constrained Variational Curves. *Computer Aided Geometric Design*, volume 12, number 5, 1995, pages 533–546.

[Wilk, 1991] Michael Wilk. Equate: an object-oriented constraint solver. *Proceedings OOPSLA'91*, 1991, pages 286–298.

[Winograd et al., 1996] Terry Winograd, John Bennett, Laura De Young, Bradley Hartfield. *Bringing Design to Software*. ACM Press Books, ISBN 0-201-85491-0, 1996.

Index

abstract data types, 5
AC, *see* arc-consistency
Active Set Method, 106
AI, *see* Artificial Intelligence
alternation model, 16, 17, 22
animation, 9
application engineer, 70, 78, 79, 87
arc-consistency, 13, 14
architectural design, 89
Artificial Intelligence, 9, 11, 12, 16, 17, 19

backtrack-free, 14
backtracking, 11, 14, 33, 84
Briar, 20

C++, 9, 12, 25, 28, 70, 71, 76, 78–80, 91–96, 109, 111
CAD/CAM, 9, 16
CCP, *see* Concurrent Constraint Programming
CH-API, 72, 73, 80
channel, 71
CIP, *see* Constraint Imperative Programming
class, 6
- aggregation, 7
- hierarchy, 6, 7
- specialization, 7
- sub-, *see* subclass
- super-, *see* superclass

CLP, *see* Constraint Logic Programming
CLP($\mathcal{X}$), 11
CODE, 2, 53, 57, 67–73, 75, 78, 87–89, 92–96, 111, 113, 117–120
`commit()`, 78, 94, 109
compiler, 6
computer graphics, 23
conceptual model, 41
Concurrent Constraint Programming, 10, 12
constrainable, 43
- normal mode, 46
- solving mode, 46

constraint, 46
- binary, 32, 99
- unary, 32, 98

constraint engineer, 70, 76–80
Constraint Imperative Programming, 21
Constraint Logic Programming, 9–11, 17, 119
constraint manager, 47
constraint rewriting, 102
Constraint Satisfaction Problem, 12–16
COPE, 21
CSP, *see* Constraint Satisfaction Problem
cycle manager, 82

data abstraction, 7
data flow, 45, 96
data structure, 6
databases, 9
declarative, 24, 26
DeltaBlue, 20
design pattern, 9
direct manipulation, 20
disjunctive normal form, 104
DNF, *see* disjunctive normal form
domain
- continuous, 32
- discrete, 32

Drawtool, 69, 87
dynamics, 91

Eiffel, 9
encapsulation, 7, 27
entity, 44
Equate, 25, 34–36
event, 43, 44, 71, 94
- targeted, 44

event source, 71
event table, 53

fair propagation, 18

feature based modeling, 90

GA-CHE, 69–72, 79, 80, 82, 87
GA-CI, 70–72, 76, 79, 80
GDP, 1–3, 23, 24, 41, 89–92, 107, 109, 111–113, 116, 118
Go, 24
Graph Coloring, 32
guard, 75

imperative, 24, 26
implementation independence, 37
information hiding, 7, 27
  violation, 27
inheritance, 6
instance variable, *see* internal variable
internal consistency, 37
internal state, *see* object, state
internal variable, 6
IWIM, 70, 71, 75

Java, 9

Kaleidoscope, 21
kinematics, 91

Lagrange Method, 106
local propagation, 17, 18, 20, 80
logic programming, 9–11
Looks, 1, 3, 24, 90–92, 107, 109, 111–113, 118

manager process, 71
manifold, 71
MANIFOLD, 2, 44, 69–75, 77–81, 88, 93, 94, 116, 117
message, 5–6
  direct access, 37
method, 6
middleware, 88
mode diagram, 58
model checking, 67

NC, *see* node-consistency
node-consistency, 13

object, 5–9
  state, 6
object diagram, 92
Object Modeling Technique, 7
  constraints, 8
Object Pascal, 9
object-oriented
  design, 5, 6, 21
  language, 5, 9
  programming, 5, 6, 23
Object-Oriented Design, 7
objective function, 16, 20, 105
Objective-C, 9
OMT, *see* Object Modeling Technique
OO, *see* object-oriented
OO-API, 72, 73, 76, 77, 80, 87
OOCS, 25, 34–36
OOD, *see* Object-Oriented Design
optimization, 105
overconstrained, 16, 20, 46, 47

path-consistency, 13
perturbation, 16
polymorphism, 6, 9
port, 71
process, 71
Prolog, 9–11
propagation of degrees of freedom, 17, 19, 20
propagation of known states, 17, 34
PROTOM, 2, 69–81, 86–88, 91–94, 113, 116
PROTOM engineer, 70, 77
prototyping, 115

QP, *see* quadratic programming
quadratic programming, 105

refinement model, 16–18, 22
relaxation, 17, 18, 20
rigid object, 91

SCAFFOLD, 3, 89–99, 102, 105, 107, 109, 111–113, 116–118
side-effect, 38
Siri, 21
Sketchpad, 19, 20
SkyBlue, 20, 119
SLD-resolution, 10–11
Smalltalk, 5, 9, 20
software
  component, 5
  engineering, 7, 23

solver, 46
spiral model, 115
state diagram, 8
stream, 71
subclass, 6
superclass, 6

ThingLab, 20
TOOLBUS, 88

UML, *see* Unified Modeling Language
underconstrained, 20, 46, 47
unification, 10–11
Unified Modeling Language, 7
user interfaces, 9, 16

VD-CHE, 89–93, 97, 98, 101, 103, 104, 106, 109, 111, 113
virtual reality, 89
VR, *see* virtual reality
VR-DIS, 1–3, 23, 24, 41, 89, 90, 92, 97, 107, 111–113, 116, 118

waterfall approach, 115
worker process, 71

Summary

In the area of interactive computer graphics, two important approaches to deal with the complexity of designing and implementing graphics systems are object-oriented programming and constraint-based programming. From literature, it appears that combination of these two has clear advantages but has also proven to be difficult. One of the main problems is that constraint programming infringes the information hiding principle of object-oriented programming. The goal of the research project is to combine these two approaches to benefit from the strengths of both.

Two research groups at the Eindhoven University of Technology investigate the use of constraints on graphics objects. At the Architecture department, constraints are applied in a virtual reality design environment. At the Computer Science department, constraints aid in modeling 3D animations. For these two groups, a constraint system for 3D graphical objects was developed.

A conceptual model, called **CODE** (Constraints on Objects via Data flows and Events), is presented that enables integration of constraints and objects by separating the object world from the constraint world. In the design of this model, the main aspect being considered is that the information hiding principle *among* objects may not be violated. Constraint solvers, however, should have direct access to an object's internal data structure.

Communication between the two worlds is done via a protocol orthogonal to the message passing mechanism of objects, namely, via *events* and *data flows*. This protocol ensures that the information hiding principle at the object-oriented programming level is not violated while constraints can directly access "hidden" data. Furthermore, **CODE** is built up of distinct elements, or *entity types*, like *constraint, solver, event, data flow*. This structure enables that several special purpose constraint solvers can be defined and made to cooperate to solve complex constraint problems.

A prototype implementation was built to study the feasibility of **CODE**. Therefore, the implementation should correspond directly to the conceptual model. To this end, every entity (object, constraint, solver) of the conceptual model is represented by a separate process in the language **MANIFOLD**. The (concurrent) processes communicate by events and data flows. The implementation serves to validate the conceptual model and to demonstrate that it is a viable way of combining constraints and objects.

After the feasibility study, the prototype was discarded. The gained experience was used to build an implementation of the conceptual model for the two research groups. This implementation encompassed a constraint system with multiple solvers and constraint types. The constraint system was built as an object-oriented library that can be linked to the applications in the respective research groups. Special constructs were designed to ensure information hiding among application objects while constraints and solvers have direct access to the object data.

CODE manages the complexity of object-oriented constraint solving by defining a communication protocol to allow the two paradigms to cooperate. The prototype implementation demonstrates that **CODE** can be implemented into a working system. Finally, the implementation of an actual application shows that the model is suitable for the development of object-oriented software.

Samenvatting

Object-georiënteerd programmeren en constraint programmeren zijn twee belangrijke manieren om met de complexiteit van het ontwerp en de implementatie van grafische systemen om te gaan. Uit de literatuur blijkt dat de combinatie enerzijds duidelijke voordelen biedt, maar anderzijds moeilijk te verwezenlijken is. Eén van de grootste problemen wordt veroorzaakt doordat constraint programmeren het principe van *information hiding* schendt. Het doel van het onderzoeksproject is de combinatie van deze twee benaderingen, zodat de sterke eigenschappen van beiden gebruikt kunnen worden.

De toepassing van constraints op grafische objecten is onderzocht in twee gebruikersgroepen aan de Technische Universiteit Eindhoven. De eerste groep is verbonden aan de faculteit Bouwkunde. Hier worden constraints toegepast in een virtuele ontwerpomgeving. De tweede groep is verbonden aan de capaciteitsgroep Informatica. Hier worden constraints toegepast ter ondersteuning van de modellering van 3D animatie. Voor deze twee groepen is een constraint systeem ontwikkeld, waarin constraints kunnen worden aangebracht tussen 3D grafische objecten.

Een conceptueel model, genaamd **CODE** (Constraints on Objects via Data flows and Events), maakt de integratie van constraints en objecten mogelijk door een scheiding te maken tussen de objectwereld en de constraintwereld. Bij het ontwerp van dit model mag het principe van information hiding niet geschonden worden. Desalniettemin moeten constraint *solvers* directe toegang hebben tot de interne structuur van een object.

Communicatie tussen de twee werelden vindt plaats via een protocol dat volledig onafhankelijk is van het mechanisme van berichtenuitwisseling tussen objecten, namelijk via *events* en *data flows*. Dit protocol verzekert dat het principe van information hiding niet geschonden wordt op het niveau van object-georiënteerde programma's. Desondanks hebben constraints direct toegang tot "verborgen" gegevens. **CODE** is opgebouwd uit verschillende elementen, of *entiteitstypen* (zoals *constraint*, *solver*, *event* en *data flow*). Deze opbouw maakt de definitie van toepassingsspecifieke constraint solvers mogelijk, die kunnen samenwerken om gecompliceerde constraintproblemen op te lossen.

De toepasbaarheid van het model is onderzocht door middel van de implementatie van een prototypesysteem bij het CWI in Amsterdam. Het doel was het realiseren van een een-eenduidige afbeelding tussen het conceptuele model en de implementatie. Op grond van de taal **MANIFOLD**, wordt iedere entiteit (object, constraint, solver) in het conceptuele model, gerepresenteerd door een apart process. De parallelle processen communiceren door middel van events en data flows. Het systeem bevat een *local propagation solver* en een numerieke solver voor lineaire gelijkheden. De implementatie vormt een validatie van het conceptuele model en toont aan dat het een geschikte manier is om constraints en objecten te combineren.

Na het toepasbaarheidsonderzoek werd afstand gedaan van het prototype. De hiermee opgedane ervaring werd aangewend bij de implementatie van het conceptuele model voor beide onderzoeksgroepen. Deze implementatie behelsde een constraint systeem voor de toepassing van constraints op 3D objecten. Het constraint systeem werd gebouwd in de vorm van een object-georiënteerde biblio-

theek, die aangeroepen kon worden door de applicaties van beide onderzoeksgroepen. Information hiding tussen objecten in een applicatie werd gegarandeerd door speciale constructies, terwijl constraints en solvers toch directe toegang hadden tot de gegevens van een object.

CODE toont aan dat een conceptueel model gedefiniëerd kan worden, waarin de twee programmeerparadigma's kunnen samenwerken door ze van elkaar te scheiden. De implementatie van het prototype toont aan dat het model in een werkend systeem toegepast kan worden. De uiteindelijke implementatie toont aan dat het model gebruikt kan worden bij de ontwikkeling van praktische, goed gestructureerde modulaire software.

Curriculum Vitae

Richard Kelleners was born on Friday, July 26, 1968 in Einighausen. He attended the Bisschoppelijk College in Sittard from 1980 to 1986 and started the Higher Informatics Education (HIO) in Heerlen. After receiving the Master's Degree in 1990 at the Technical Information Systems section, he moved on to the Eindhoven University of Technology (TUE) to study Computer Science. In 1993, he received his Master's Degree at the Computer Graphics section. In 1994, he took a position as a Ph.D. student which was partly carried out at the CWI (Centrum voor Wiskunde en Informatica) in Amsterdam and partly at the TUE in Eindhoven. He worked at CWI from 1994 until 1996 where he was a member of the MANIFOLD group. From 1996 until 1999, he worked at the Computing Science department of the TUE. Since April 1999, he is employed at VDO Car Communication and works on the development of the CARiN navigation system.