Interactive Exploration of Numerical Constraint Satisfaction Problems

by

SESHU PARVATANENI
BE, Osmania University, 1996

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming to the required standard

Dr. Maarten H. van Emden, Supervisor (Department of Computer Science)

Dr. Daniel M. Hoffman, Departmental Member (Department of Computer Science)

Dr. Jens H. Jahnke, Departmental Member (Department of Computer Science)

Dr. Zuomin Dong, External Examiner (Department of Mechanical Engineering)

© SESHU PARVATANENI, 2003

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part by photocopy or other means, without the permission of the author.

QA 297
P37

Supervisor: Dr. Maarten H. van Emden

ABSTRACT

One of the problems of conventional numerical computation is rounding error. Interval methods like interval arithmetic and interval constraints automatically compute an estimate of the error for each real number. These methods have proved to be valuable in numerical computation because of their certainty in the results. But problems with conventional numerical analysis performance is the key issue. The only way to get the guarantees of the interval methods accepted, is to offer the guarantees without penalty in performance. An application of a commercially available package NUMERICA, which finds all solutions to nonlinear systems of equations, proves this. However, the package is proprietary. In this thesis, we develop an interval constraint system by composing reusable units of software using the Component Object Model(COM) architecture. Our design of the interval constraint system ensures reusability and interoperability among different programs on the same machine. The programs may have been developed using different programming languages. In addition, we develop a graphical user interface application for the interval constraint system. This is the first system for the interactive exploration of constraint satisfaction problems.

Examiners:

Dr. Maarten H. van Emden, Supervisor (Department of Computer Science)

Dr. Daniel M. Hoffman, Departmental Member (Department of Computer Science)

Dr. Jens H. Jahnke, Departmental Member (Department of Computer Science)

Dr. Zuomin Dong, External Examiner (Department of Mechanical Engineering)

Table of Contents

Abstract ii

Table of Contents iv

List of Figures vii

List of Tables ix

Acknowledgement x

Dedication xi

1 Introduction 1

1.1 The rise of interval methods 1

1.2 Our solution 2

1.3 Thesis overview 3

2 The Component Object Model 4

2.1 Terms and concepts 7

2.2 Technical detail 8

2.2.1 Marshaling and unmarshaling 11

2.2.2 Component object activation 11

2.2.3 In-process vs local and remote object activation 12

2.2.4 Automation, Binding and Type libraries 13

2.2.5 Interface definition language 15

2.2.6 IUnknown 17

3 Interval Arithmetic and Interval Constraints 21

3.1 Intervals 22

3.2 Relations, Constraints, and Boxes 23
3.3 Interval Arithmetic 24
3.3.1 Classification of non-empty real intervals 25
3.3.2 Interval addition and subtraction 26
3.3.3 Interval multiplication 27
3.3.4 Interval division 27
3.3.5 Interval exponentiation 28
3.4 Interval Constraints 30
3.4.1 Primitive constraints 31
3.4.2 The constraint contraction operator 32
3.4.3 The interval constraint system 35
3.4.4 Constraint propagation 36

4 Interval arithmetic and constraint system components **38**
4.1 Interval arithmetic component 39
4.1.1 Implementation 40
4.2 Interval constraint system component 44
4.2.1 Implementation 45

5 Interval Constraint System Expression Parser **50**
5.1 Syntax graphs 50
5.2 Obtaining code from syntax graphs 52
5.3 Expressions: design and implementation 55

6 Graphical User Interface and System Usage **60**
6.1 The user tasks 60
6.1.1 Elimination 60
6.1.2 Probing 62
6.2 Differences between Elimination and Probing 63
6.3 Graphical controls and their behavior 63
6.4 Component activation and usage 66

7 Conclusion and Future work **68**
7.1 Main features of this thesis research 68

7.2 Contributions 69
7.3 Limitations of ICS Explorer 70
7.4 Future work 70

Bibliography **75**

Appendix A **77**

List of Figures

Figure 2.1 Binary representation of a component exposing an interface . . 8
Figure 2.2 Binary layout of a client getting a pointer to the interface node 9
Figure 2.3 A component with multiple interfaces 9
Figure 2.4 The SCM, the proxy and the stub interacting between component servers 13
Figure 2.5 Bit pattern of the information in a HRESULT 16
Figure 2.6 An example of a type library defined in an idl file 19
Figure 2.7 TypeLib typelibrary viewed with OLE/COM Object viewer application 20

Figure 3.1 Interval constraint contraction 31
Figure 3.2 Primitive Constraints. 31
Figure 3.3 Waltz constraint propagation algorithm 37

Figure 4.1 COM notation for `CImpIReal` class 44
Figure 4.2 Class hierarchy diagram for the primitive constraints 46
Figure 4.3 COM notation for `CImpIICSystem` class 48

Figure 5.1 Variable 51
Figure 5.2 Unsigned Integer 51
Figure 5.3 Unsigned Number 51
Figure 5.4 Constant 51
Figure 5.5 Factor 52
Figure 5.6 Term 52
Figure 5.7 Expression 52
Figure 5.8 Atomic Formula 53
Figure 5.9 Conjunction 53
Figure 5.10 Formula 53

Figure 5.11 Sequence of elements 53
Figure 5.12 Choice of elements 54
Figure 5.13 Loop 54
Figure 5.14 Element of a graph denoting another graph 55
Figure 5.15 Element of a graph denoting a terminal symbol 55
Figure 5.16 Parsing program for a Variable 56
Figure 5.17 The `COParser` component class and the `IParser` interface . . 57
Figure 5.18 Interface definition for `PConstraint` 58
Figure 5.19 Interface definitions for `Exp` and `Formula` 58
Figure 5.20 The Expression class hierarchy 59

Figure 6.1 Interval Constraint System Explorer 64
Figure 6.2 Automatic input 64

List of Tables

Table 3.1 Classification of intervals by sign. Only non-empty intervals are classified; hence $a \leq b$ 26
Table 3.2 Case analysis for multiplication of real intervals, $[a, b] * [c, d]$ . . 27
Table 3.3 Case analysis for division of real intervals, $[a, b]/[c, d]$ 28
Table 3.4 Power of a real interval, $[a, b]^n$ 29
Table 3.5 Root of a real interval, $[a, b]^{\frac{1}{n}}$ 29

Table 4.1 Naming convention for the source code in this thesis 41

Acknowledgement

I gratefully acknowledge the support and presence in this work of many people without whom I never would have completed the program. I would like to thank Professor Maarten van Emden, my supervisor, whom without his technical discussions, enthusiastic supervision and financial aid this thesis could not have been complete. I greatly appreciate his patience and support.

I would like to thank Professors Zuomin Dong, Daniel Hoffman and Jens Jahnke for taking the time in reading this thesis and for their comments.

I am greatful to all my friends during these years. I thank Jayakrishnan Nair for his inspiration. Rambabu Karumudi for his valuable discussions and references that helped me during the course of my research. Salil Shukla for giving me moral support when I most needed.

Finally, I am forever indebted to my parents and brother for their undying love, support and encouragement when it was most required.

Dedication

To Open Source Software Development

"let people be freed from hunger"

— anonymous

Chapter 1

Introduction

One of the problems of conventional numerical computation is rounding error. Interval methods such as interval arithmetic and interval constraints automatically compute an estimate of the error for each real number. These methods have proved to be valuable in numerical computation because of the certainty in the results they provide. Interval arithmetic has been under development since the 1960s in numerical analysis starting from the book [1]. The interval constraint method was developed later in artificial intelligence [2] and in logic programming [3].

1.1 The rise of interval methods

In numerical analysis, performance is the key issue. The only way to get the guarantees of the interval methods accepted is to offer superior quality at acceptable performance in terms of time and memory. Numerical computations involving interval arithmetic were initially rejected but have gained recognition recently for the following reasons:

- *Presence of memory hierarchies*: The extra computations that interval arithmetic involves are local and mostly refer to the cache memory or lower levels of memory. This makes interval arithmetic less expensive compared to the period when memory consisted of a single layer.
- *Acceptance of IEEE-standard for binary floating point arithmetic*: Due to this fact an interval arithmetic system can be implemented that is
  - *Sound*: The resulting interval from the computations always contain the theoretical real number value.

  - *Closed*: The result from the interval computations is always defined. The computations never need to be aborted due to an invalid operation, division by zero, overflow, or from production of a NaN.
  - *Efficient*: Using the features of the IEEE 754 floating point arithmetic the number of tests can be minimized to achieve soundness and closedness.
- *Development of interval constraints*: Interval constraints is more than a mere application of interval arithmetic. Interval constraints solve in general an interval constraint system. Whereas interval arithmetic evaluates only expressions.

An application of the package NUMERICA [4], finding all solutions to nonlinear systems of equations, has shown that the superior security is achievable without penalty in speed. Test results are given in [4]. Another most commonly used example for offering soundness and improvement in performance is proving the uniqueness of the previously known solution to the Ebers-Moll transistor model [5, 6]. Moreover, interval constraints reduced the required computation time considerably compared to interval analysis.

The obstacles that NUMERICA faces and the motivations for another interval constraint system to be developed are:

- The programs in NUMERICA are written in a Matlab-like language. This feature prevents the programs to be integrated into existing applications.
- The source code of the package is proprietary. A user cannot verify the implementation of interval arithmetic with respect to the claim that the package makes about the results. A correct and efficient interval arithmetic is a non-trivial task which can be done only by studying the system's source code.

1.2 Our solution

The above two obstacles can be overcome by implementing a suitable suite of components and making the source code open. The Component Object Model can be used to implement these components in the C++ programming language. Implementing the components in C++ has the following advantages.

- C++ possesses powerful features such as manipulating the hardware facilities in a direct and an efficient way. Such features are lacking in Java.

- When skillfully programmed in C++, the programs compile to the fastest code achievable in a high-level language.
- C++ is the native language for implementing components in COM. It should be noted that components can be implemented in other programming languages too. Java and Visual Basic scripts are the other programming languages.
- Visual interfaces can be designed easily by using the software development kit for Visual Basic.
- Finally COM is the Lingua Franca among Microsoft's applications. Especially Microsoft Excel is a good candidate for interval-based enhancements.

1.3 Thesis overview

In Chapter 2, we introduce the problems that the software industry was facing before the advent of component software technology. Later we introduce the terms and concepts in the Component Object Model followed by the technical details in the model. Chapter 3 gives a summary of interval arithmetic and interval constraint methods from previous research with tables showing the interval operations. In this chapter we discuss the different parts that make up an interval constraint system. We describe the constraint contraction operator and the constraint propagation algorithm. Chapter 4 discusses the design and implementation of the COM components for interval arithmetic and the interval constraint system. In Chapter 5, we introduce the need for a parser, discuss the parser syntax graphs, the translation of these syntax graphs to C++ source code and the component provided for the parser. Chapter 6 walks through the graphical components in the visual interface that we develop, the tasks the user is expected to perform and the interval constraint system usage in a Visual Basic script. We conclude the thesis in Chapter 7 with the contributions and possible future work for the interval constraint system and the visual interface.

Chapter 2

The Component Object Model

This chapter presents our motivation for software component technology and for selecting one of the existing component technologies for our software design and development. We present the terms, the concepts and the technical details in the component software standard, Component Object Model, with examples where ever appropriate.

The high demands from software users has created, innovative languages and techniques in programming. As a result of this innovation the object-oriented programming paradigm was introduced. The software industry embraced the paradigm due to its ability to solve complex problems by breaking them down into smaller sub-problems. However, the software industry faces some challenges which cannot be overcome solely by using object-oriented programming. Some of the challenges are:

- *Interoperability*: It often happens that one application cannot be integrated into another application. One way in which this difficulty arises is that the data and functionality of one application is not available to another application even if they are running on the same machine and if they are programmed in the same language. It is also necessary that they be generated by the same compiler. The source of the problem lies in the layout of the binary executable code that is generated from the program source code. Generation of executable code from source code involves two stages: *compiling* and *linking*.
  Compiler: A compiler is a program that converts the source code into an intermediate binary code known as object code. The object code cannot be executed by itself and may span across several files.
  Linker: A Linker is a program that links the object code with any other library functions that are called from the program to form a single *executable* object

file or simply a *program*.

A program is executed using a *loader*. It is an operating system utility that copies a program into the computer's main memory in such a form that it can be run. The loader also allocates the required storage space for the program.

There exists a wide range of compilers and linkers for a particular programming language ; GNU CC, Borland C++, Apple's MrCpp to name a few.

As there is no specific standard to generate object code, each compiler has its proprietary way of rendering the source files. As a result the object code that is generated by one compiler may not be able to link successfully with the library that was created using another compiler.

Another problem that exists due to the lack of a standard arises from the Object-Oriented feature, *function overloading*. In Object-Oriented programming there exists a feature to write methods with same name and return type but with different parameter lists known as function overloading or *overloaded functions*. When an overloaded method is called from a program, the appropriate method is chosen by the compiler at compile time. Linkers cannot handle overloading. The compiler has to translate the method names into distinct symbols in order to identify the methods correctly. This translation is known as *name mangling*. Since there is no standard to implement the name mangling scheme each compiler has its own proprietary technique for name mangling too. These incompatibilities between compilers and linkers make it difficult or impossible for programs to interoperate.

- *Resistance to reuse*: Object-Oriented programming facilitates reuse of source code. Unfortunately, most of the time more effort goes in understanding other developer's source code than reimplementing the code from scratch. Sometimes there is a mental overhead that is required to understand the developer's design and programming style. This is especially true while designing wrapper-style libraries. The developer has to understand not only the underlying technology being wrapped but also the additional abstractions added by the library itself [7].

The solution to the above challenges can be overcome by:

- Imposing a binary standard on the layout of the binary code that a compiler

and a linker generates and

- Composing an application from various, independent, reusable units of software in binary form. *Procedural* and *class libraries* are the oldest examples of these kind of reusable software units.

The software composed by following the above two conditions is known as *component software* and the individual software units as *components*. In addition to overcoming the above challenges the use of software components has several other advantages, some of which are listed below:

- Application developers can compose applications very easily and also attain higher productivity as they can learn one component system for many platforms.
- Application users may choose from among various available components, the best component, depending on the level of performance, efficiency, robustness, etc. as a part of their application. This also creates competition among component developers to create the best component and the application users to offer their best service.
- Component software also eliminates the problem of massive upgrade cycles [8]. Periodic upgrades are inevitable in traditional software which is integrated into one big application. These upgrades normally take a long time involving testing and verification of the upgraded application, ensuring upward compatibility and sometimes also necessitate retraining the software users. In component based software development, individual components can be upgraded whenever necessary without causing major disruptions to the user's operations.

Today, several technologies that impose a binary standard for components exist. The most prominent standards in the component software technology are: **C**omponent **O**bject **M**odel (COM), Common **O**bject **R**equest **B**roker **A**rchitecture (CORBA) and **J**ava **2** **E**nterprise **E**dition (J2EE). These standards are formulated and maintained by organizations each consisting of several software industries. COM is maintained by The Open Group [9], CORBA by The Object Management Group [10] and J2EE by Java [11].

We have selected the COM technology to design and develop our software components for the following reasons:

- Our decision to provide interval-based enhancements to the Microsoft spreadsheet application, Microsoft Excel
- COM being the Lingua Franca of all Microsoft applications

An introduction to terms and concepts in software programming and COM are required to understand the component software design process using COM. The following section introduces the necessary terms and concepts.

2.1 Terms and concepts

A *client* is a computer program that requests or uses the data and functionality of another program, the *server*. Specific to COM, a client is an application that controls the program sequence between itself and other software *components* (described later in this section).

A *server* is a computer program that responds to the requests of a client. Specific to COM, a server is an application that structures the component in a specific fashion so as to enable clients to communicate with it in a standard manner.

An *object* is an encapsulated unit of software consisting of a specific state and behavior. Objects are instances of classes.

A *component* is an equivalent name for an object in COM with the following differences. An object can have direct access to all the data and functionality of another object, whereas a component can never access another component's data and functionality in its entirety. One component can access other component's data and functionality only through a pointer to an *interface* (described later in this section). Sometimes a component is also referred to as a *component object.* Figure 2.1 shows a component, `COMyObject`, in binary form exposing its functionality through the interface, `IMyObject`.

An *interface* is a contract between the client and a component. The contract states what the client has to do to use the interface and what the component has to implement to meet the services promised by the interface. A more elaborate description of an interface will be given later in this chapter.

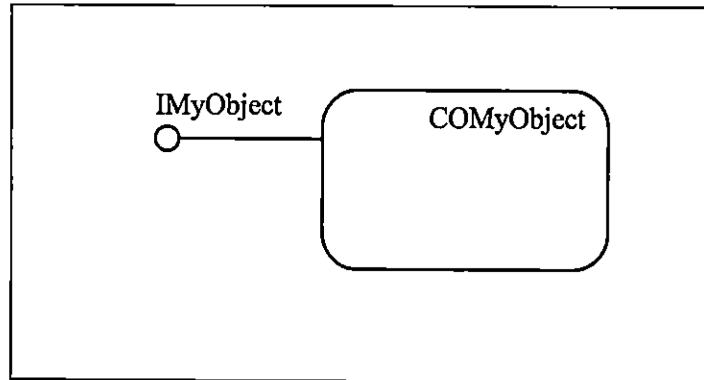

Figure 2.1. *Binary representation of a component exposing an interface*

A *process* is an instance of a program that is running on a computer. A process has its own context (explained under Context)

A *thread* is a single sequential flow of program control within a process. A process may have more than one thread. A thread has no context of its own.

A *multi-threaded program* is a program which has more than one thread.

Context: When an application or an executable is launched the operating system starts a new process, creates a thread for the process, loads the application data into memory and begins processing at the first instruction.
In a multi-tasking environment programs that are being executed in memory may be suspended and removed from the processor's thread of execution to start another program which needs to be processed. To resume the suspended program from the identical operating state, the operating information has to be stored. This operational state data is known as the *process's context.*

2.2 Technical detail

The Component Object Model defines a binary standard for the interface between a client and a server. The standard does not specify how these interfaces should be implemented. It only specifies how these interfaces should be created in binary format. COM promises interoperability between components based only on this principle.

Interoperability in COM is achieved through the use of virtual function pointers

known as *vptrs* and a table of vptrs known as *vtbl.* When a request for a component is made, the client only gets a pointer to the interface node of the component. This interface node is another pointer which points to a vtable. As long as the compiler that generated the binary code for the component obeys the COM standard, the implementation language for the client and the component does not matter. Figure 2.2 shows the binary layout of a client getting a pointer to the interface node as well as the vptr and vtbl layout of the component.

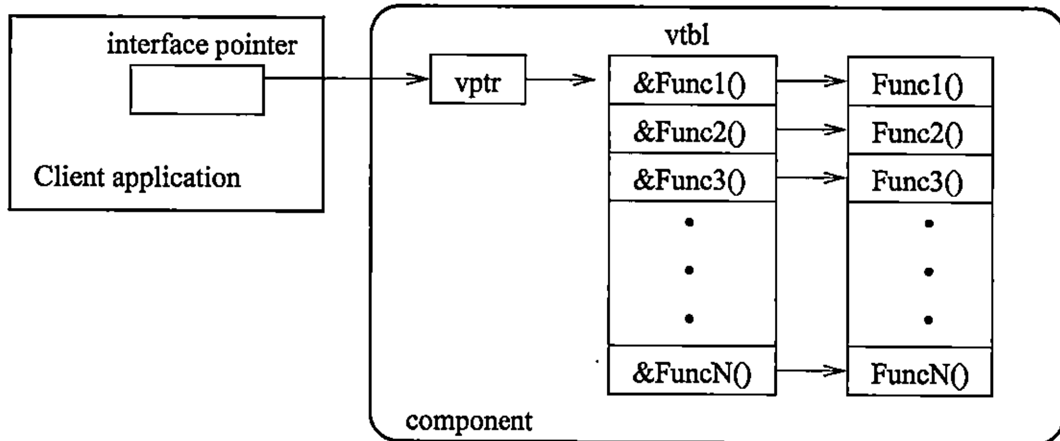

Figure 2.2. *Binary layout of a client getting a pointer to the interface node*

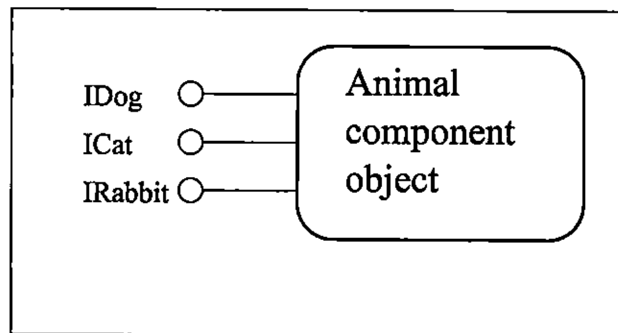

Figure 2.3. *A component with multiple interfaces*

A component can support many interfaces. Different interfaces can expose different functionality of the component. Figure 2.3 shows an example of a component, `Animal`, that exposes three interfaces: `IDog`, `ICat` and `IRabbit`. The `IDog` interface may contain functions to `bark` and `wag` its tail. The `ICat` interface may contain

functions to `eat` and `ignore` the master. Similarly the `IRabbit` may have functions to `eat` and `hop`. However, all these functions may be implemented by the `Animal` component only. Other components cannot access the functionality of the `Animal` component in its entirety. The only way for them to access the functionality of the component is by getting a reference to one of the interfaces the component exposes.

Interfaces are considered to be immutable. An interface definition should not be changed or modified once it has been published. This is very important in COM in order to keep the contract valid at all times. If the interface is modified after it has been published the vtbl layout of the component will be altered causing the function pointers to offset from their original position. This may result in undefined behavior by the client applications and may also cause them to fail severely. The only way to add new functionality to a component without causing catastrophic failure is by adding a new interface to the component. The addition of this interface does not impact the virtual table layout of the existing interfaces that the component supports thus, avoiding the catastrophic failure.

Every component executes in a server. A server may contain more than one component. The component in a server can be accessed in one of three ways: as an *In-process server*, a *Local object server* and a *Remote object server*.

1. An in-process server is a component server that loads into the client's process space. The communication between the client and the server is achieved through function calls. On all Microsoft windows operating systems, in-process servers are implemented as *Dynamic Link Libraries* or DLLs.
2. A local object server is a component server that executes in a separate process and exists on the same machine as the client. The communication between the client and server is accomplished through Remote Procedure Calls or RPCs. On the Microsoft windows operating systems local object servers are implemented as executables or EXEs.
3. A remote object server is a component server that executes on a machine different from that of the client. Since the server executes on a different machine it executes on a different process too. Communication between the client and the server is achieved through Remote Procedure Calls that are provided by the Distributed Computing Environment (DCE). DCE is a standard and is

maintained by The Open Group [9]. The procedure calls are known as DCE RPCs. The mechanism through which this communication is achieved is called Distributed Component Object Model or DCOM.

2.2.1 Marshaling and unmarshaling

Data between the client and the server can be shared directly if they execute on the same process space. This sharing is not possible if the data is in a different process. To enable sharing of data between processes, COM formats and bundles the client's or server's data in one process before communicating with the client or server in the other process. When the data is received by the other process COM unpacks, unformats and makes the data available to be read. This process of formatting and bundling the data is known as *marshaling.* The process of unpacking and making the data available to be read is known as *unmarshaling.* The code for marshaling and unmarshaling can be generated using a COM compliant compiler, example Microsoft's IDL compiler. The code that is generated by the compiler is put in two separate components, a *proxy* and a *stub.* The stub component will be required by the server's process and the proxy component by the client's process.

When a client makes a function call to a function in the server component, the proxy marshals the function parameters and passes them to the server stub. The server side stub then unmarshals the parameters and makes the actual function call inside the server process. The stub after completing the call marshals the return values and passes them back to the proxy. The proxy then unmarshals the return values and returns them to the client.

2.2.2 Component object activation

A client's request for a component involves in loading a DLL or starting up a process for the server. The process of bringing a component to life in this manner is called *component object activation.* If a component that is present in an in-process server is activated it is called an *in-process object activation.* If the component that is present in a local object server is activated it is called a *local object activation* and if the component that is present in a remote object server is activated it is called a *remote*

object activation. The services required for an activation process are offered by the COM *Service Control Manager* (SCM). Every machine that supports COM has its own Service Control Manager.

If a client makes an object activation request on a local machine the SCM locates, creates the component and binds the initial interface pointer with the client. When the request is made for an object on a remote machine the SCM locates the remote machine and forwards the request to the SCM on the remote machine. The SCM on the remote machine then treats the request as a local activation request, creates the component and binds the initial interface pointer with the client. Once the interface pointer is bound to the client, the SCM is out of the picture. On the Microsoft operating systems the SCM is implemented as an application and is named RPCSS.EXE. The services provided by the SCM are exposed to programs through high-level COM objects which locate components and as low-level API functions. These COM objects and functions including other services offered by COM are packaged in a COM library. The majority of the COM library on the Microsoft windows platform is implemented in a DLL named OLE32.DLL. Figure 2.4 shows the SCM interacting with different types of component servers.

2.2.3 In-process vs local and remote object activation

Programs or applications that execute in the same process and thread can share the data directly if the location of the data is known.

In an in-process object activation the component and the client are in the same process space. Therefore the client need not switch from the current thread and process each time it tries to access a component's function. COM treats the function call as a local call.

In the local or remote object activation the component and the client are in different processes and therefore different threads. Unlike the in-process object activation a direct communication between the client and the component cannot be established. The client side proxy component which executes on the client's process and thread has to marshal the data and make remote procedure calls to the server's stub. This involves context switching. There is a considerable operational overhead resulting from the context switches which may lead to reduced performance of the client.

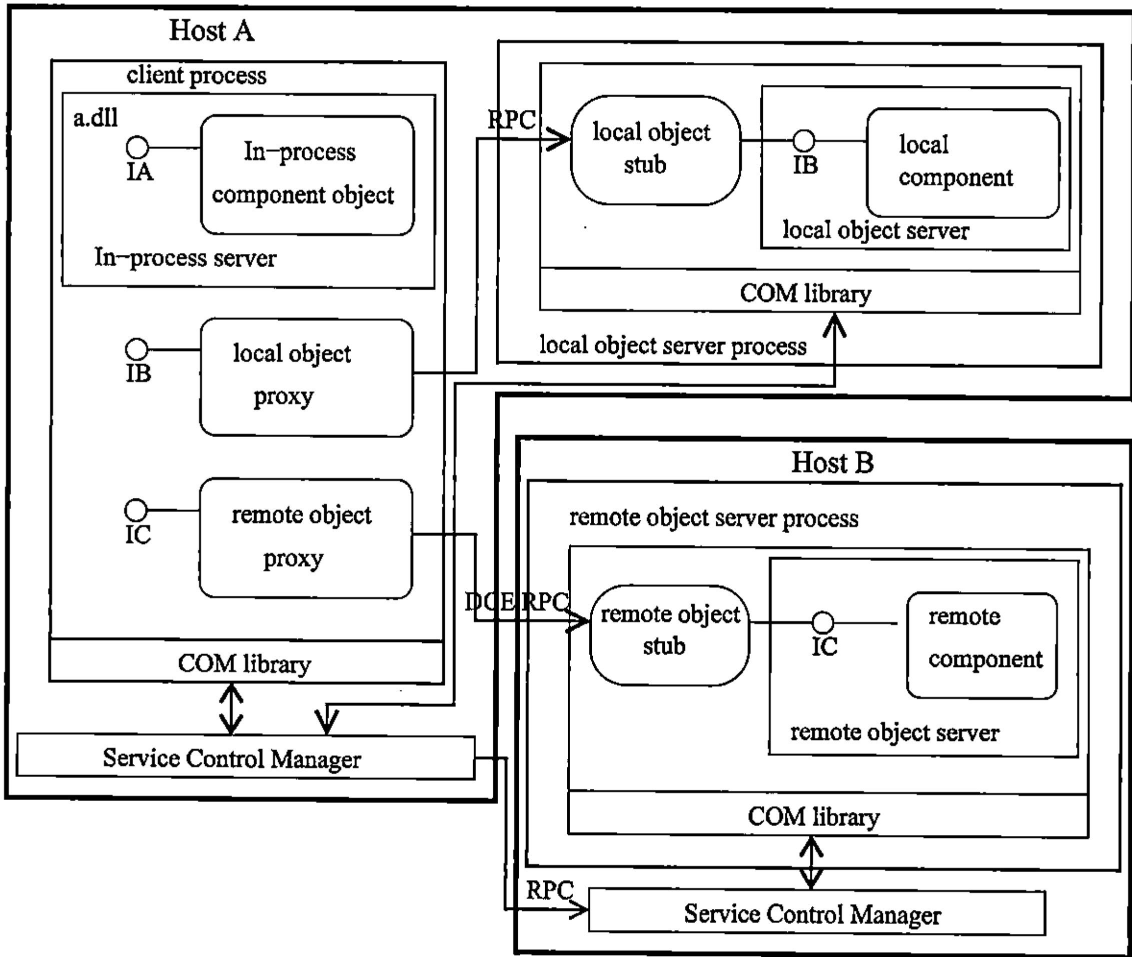

Figure 2.4. *The SCM, the proxy and the stub interacting between component servers*

Implementing the components in a local or a remote object server would degrade the performance of our system. Therefore, considering the performance factor we have decided to implement the components involved in our system as an in-process server — a DLL.

2.2.4 Automation, Binding and Type libraries

An incorrect function call or parameter to a function in a component may lead to undefined behavior of the client program or may even cause the program to fail. COM provides a mechanism to prevent this from happening. Each time the client makes a function call COM verifies the validity of each function call before actually making the

call. The verification process is known as *binding* and the mechanism used for binding is called *automation.* Automation enables applications to communicate with software components, possibly developed in different languages. In particular for programs in Java and scripting languages such as Microsoft Visual Basic and JavaScript. A client for the component object equipped with automation is called an *automation controller* and a component server equipped with automation, an *automation server.* In an automation controller binding can be performed during runtime or at compile time. Binding that occurs at runtime is called *late binding* and that occurs during compile time is called *early binding.*

Late binding: Often, the exact interface of a component cannot be discovered until the runtime of the automation controller. Late binding is the automation process where the interface of the component in the automation server is discovered and bound to the automation controller during runtime. An automation controller activates a component only after it checks the availability of the component's interface in the automation server. For each function call in the automation server's component the automation controller first checks the availability and validity of the component's interface and then check the function in the component. These look-ups are time-consuming and are performed every time a component's function is called. The look-ups can sometimes considerably degrade the performance of an application.

Early binding: If the interface that a component supports is known at compile time a reference to the interface can be set instantly rather than delaying the component type discovery until runtime. Early binding is an automation process where the binding of all the components and functions in the automation controller are done during compile time. Early binding eliminates the need for the time consuming look-ups that verify the validity of the component's interface and the function calls.
To allow early binding in automation clients the automation server should define the component classes in a *type library.* A type library groups a collection of data types and the component classes that the automation server exports into a namespace. For the automation controller to take advantage of early binding it has to set a reference to the automation server's type library. By doing so, the components and the function calls in the server are bound to the automation controller during the controller's compile time and hence, the automation controller is said to be early-bound.

A type library is generated using the MIDL compiler and describes the following:

- instantiable class definitions of the components known as component object classes or coclasses for short,
- interfaces the components support, and
- type definitions, enumerations or constants the automation server makes available.

We allow our automation controllers to be early-bound for the following two reasons:

- late binding would degrade the performance of our system and
- the user is always aware of the required components and their interfaces before compile-time.

2.2.5 Interface definition language

Type libraries, component object classes, interfaces and functions in COM are specified using a computer language called the *Interface Definition Language* or IDL. The IDL can be easily understood by any computer programmer regardless of their development environment. The IDL should not be mistaken for a programming language. It does not have any programming constructs such as variable assignments, looping, selection and definitions for procedures. The MIDL compiler is required to convert the IDL source code to binary code. The MIDL compiler requires that the source code be contained in a file with a `.idl` extension.

A typical interface defined in IDL consists of four parts: an interface name, a base interface name, an interface body and the attributes of the interface.
COM specifies that every interface should inherit from the base interface `IUnknown` (described later in this chapter), directly or from another interface that inherits it.

COM requires that an interface contain at at least two IDL attributes: the `object` attribute and the `uuid()` attribute. The `object` attribute is required to indicate that the interface being declared is a COM interface.
Quite often it happens that two different programmers choose the same name for a function, method or a class that might express similar functionality. Once the source code is converted into binary it is impossible to determine which version of the code

should be linked to the client's application. One might face a similar situation using components. Because interoperability in COM is at the binary level it is crucial to get a reference to the expected interface. To eliminate name collisions between COM interfaces, every COM interface is assigned a unique name at design time. A *unique universal identity* (uuid) is the physical name given to an interface. uuids are 128-bit numbers called *Globally Unique IDentifiers* or GUIDS. GUIDs are guaranteed to be unique at any given point of time and machine. GUIDs can be generated using an application called Guidgen. uuids when assigned to COM interfaces are called Interface IDs or IIDs. The following is a sample GUID that was generated using Guidgen in its canonical form:

{61BABF78-B380-45BC-AAE7-F7621638A7F1}

Guidgen guarantees that this number will never be generated again in the future. The uuid attribute takes the canonical form of the GUID as a parameter.

The body of an interface consists of function declarations. In IDL, every function declaration should return a value of type HRESULT. The function parameters may have attributes that indicate the direction in which they have to be copied. HRESULTs are 32-bit integers that provide information to the caller about the kind of error that may occur in the function. HRESULTs are partitioned into three bit-fields: the severity bit, the facility code and the information code as shown in Figure 2.5 [12].

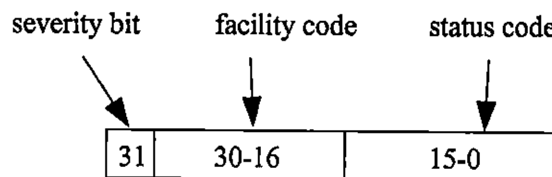

Figure 2.5. *Bit pattern of the information in a HRESULT*

The severity code in the most significant bit reports whether the function call succeeded or failed. A value of zero corresponds to a success and a value of one, a failure.

The facility code consists of 15 bits and provides information about the type and the origin of the return code. The following are the facility codes in hexadecimal format with the textual representation of the code that a HRESULT supports.

0 - FACILITY_NULL

1 - FACILITY_RPC

2 - FACILITY_DISPATCH

3 - FACILITY_STORAGE

4 - FACILITY_ITF

7 - FACILITY_WIN32

8 - FACILITY_WINDOWS

10 - FACILITY_CONTROL

The last 16 bits of the HRESULT consist of the status code. The status code contains the actual information that the function is returning.

The hex code 0x7040030 is an example of a HRESULT: 0 for the severity bit, 4 for the facility code and 3 for the status code. New HRESULTs can be composed using the macro MAKE_HRESULT. It takes three parameters: the severity error, the facility code and the status code. The facility code for these kind of HRESULTs should be FACILITY_ITF which indicates that the error is declared solely by a specific interface. A code higher than 0x200 should be assigned to avoid reusing the status codes that have already been defined in COM.

The optional attributes to the function parameters describe which way the parameters need to be copied. For the following function declaration, fx, in IDL

```
HRESULT fx([in]int x, [out]int *y, [in, out]int *z);
```

The attribute [in] specified for the first parameter, x, will copy the value from the client to the server. The attribute [out] specified for the second parameter, y, will copy the value from the server to the client. The attribute [in, out] specified for the third parameter, z, will copy the value in both the directions. As the value in the last two parameters have to be modified inside the server they have to be passed as pointers.

2.2.6 IUnknown

The IUnknown interface is the base COM interface. Every COM interface should either directly or indirectly inherit from the IUnknown interface. It contains three functions:

```
HRESULT QueryInterface(REFIID riid, void **ppv);
ULONG AddRef(void);
ULONG Release(void);
```

The `QueryInterface` function allows the client to query whether a component object supports a specific interface identified by passing the IID as the first parameter. A pointer to the interface is returned via the ppv parameter if the query succeeded. The success or failure of the query can be checked by the HRESULT code that the function returns.

To keep track of the number of activations of a component object there exists a mechanism called *reference counting* in COM. Reference counting is achieved by the use of a counter. Every time a component object is activated the counter is incremented. The counter is decremented when the reference to the component object is removed or no longer in use. When the counter's value becomes zero COM deletes the component object from the memory. This counter is incremented by the `AddRef` function. The `Release` function decrements the counter and deletes the component object when the value of the counter becomes zero.
IDL specifies that the type library and the coclasses have physical names too. This is the only attribute to the type library or a coclass and is not optional.

Figure 2.6 shows IDL source code for a type library `TypeLib`. The type library describes an instantiable component object class, `COExample`. The interface that this component supports is `IExample`. It is also worth noting that the component class, `COExample`, can support more than one interface. `IExample` inherits from the `IDispatch` interface. `IDispatch` inherits from the base interface `IUnknown` and contains the code required for automation. `IExample` interface exposes a function, add. The function takes two integers as the first two arguments as input, adds the two integers and returns the result through its third argument. The `import "oaidl.idl"` and `import "ocidl.idl"` statements import the standard interfaces. The `importlib "stdole32.tlb"` and `importlib "stdole2.tlb"` import the standard type libraries.

We can see that the given IDL code is self-explanatory and can be easily understood by any programmer regardless of their development environment.

When the code in Figure 2.6 is saved in a file with a `.idl` extension and compiled using a MIDL compiler it generates the proxy/stub code, the code needed for automa-

```
import "oaidl.idl";
import "ocidl.idl";
[object, uuid(8A6B4777-2A99-4D43-A20A-7D75975B8E21)]
interface IExample : IDispatch {
   HRESULT add([in]int x, [in]int y, [out,retval]int *z);
};

[uuid(6C4F86C2-0E23-4CEE-8E57-DB06D3E029E0)]
library TypeLib {
   importlib("stdole32.tlb");
   importlib("stdole2.tlb");
   [uuid(3EF2B590-B5C9-4982-AD0B-5B4F3E7B36B8)]
   coclass COExample {
     interface IExample;
   };
};
```

Figure 2.6. *An example of a type library defined in an idl file*

tion and the type library in binary form. The generated type library can be viewed with an application called OLE/COM Object viewer. Figure 2.7 shows a snapshot of the above type library viewed with the OLE/COM Object viewer.

Below is the Visual Basic code implemented as a automation controller showing the activation of the component present in the above automation server example.

```
Dim objExample As COExample 'declare the component object
Set objExample = New COExample 'create an instance of the object
If objExample Is Nothing Then 'could not create the object
    MsgBox "Could not get reference to a Example."
    Exit Sub
End If
Dim x As Integer
x = objExample.Add(7, 3) 'call the component object's add function
```

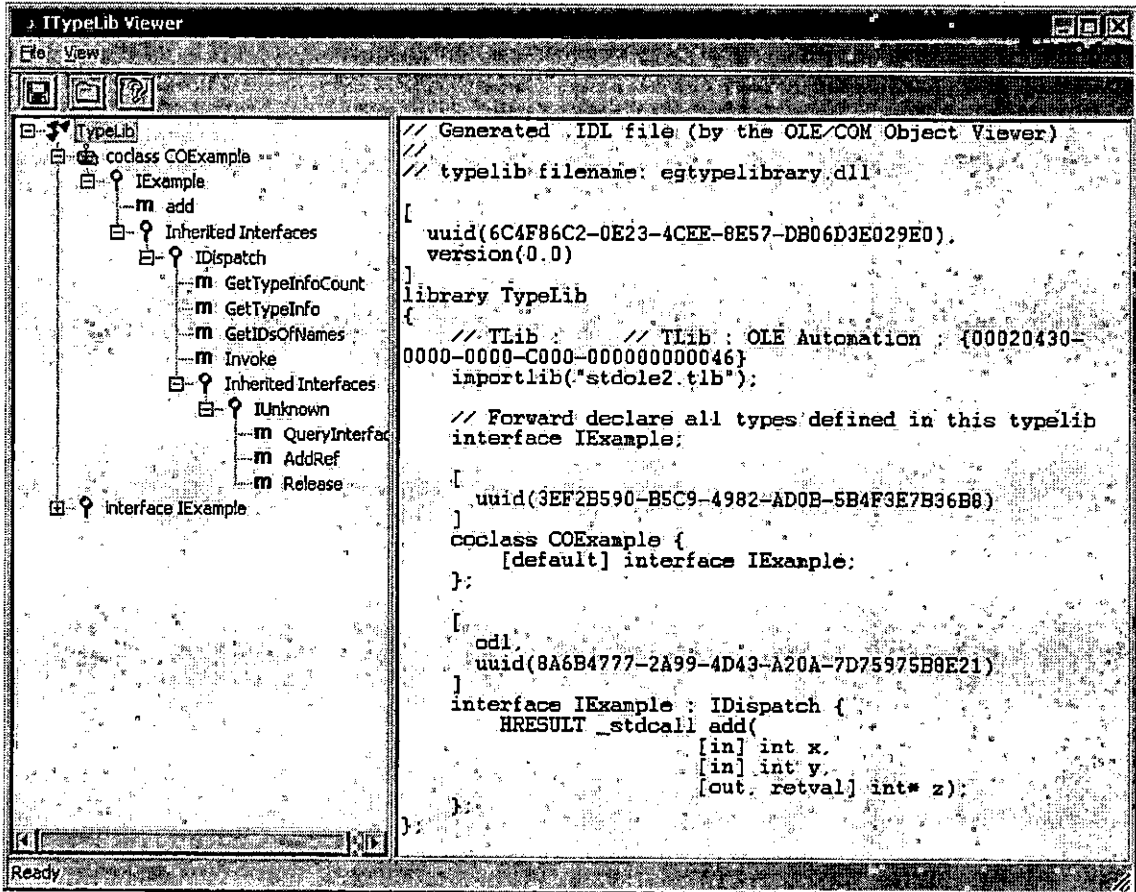

Figure 2.7. *TypeLib typelibrary viewed with OLE/COM Object viewer application*

The terms, concepts, technical details and examples provided in this chapter are sufficient to understand the design process of the software components involved in the interval constraint system. Some references for component object design using COM are [7, 13, 14] and for CORBA [15, 16].

Chapter 3

Interval Arithmetic and Interval Constraints

Intensive research has been done in establishing theorems and proofs involved in interval arithmetic and interval constraints. In this chapter we give background on these interval methods and summarize the theoretical results from [17] and [18] that pertain to our implementation. These publications have extensively used the IEEE standard [19] in their discussions.

Interval arithmetic forms the foundation for interval constraints. Interval arithmetic was introduced by Ramon Moore [1] in the 1960's. Interval arithmetic is an alternative for representing the value of computed results as $x + \epsilon$, where x is the result and ϵ is the *error tolerance.*

On a computer, as we have to use a finite set of floating-point numbers that the hardware makes available, the error tolerance, ϵ becomes important. In interval arithmetic, instead of expressing the value as a real number and the error tolerance, the value is expressed using two floating point numbers. Interval arithmetic guarantees the presence of the result within these two floating point numbers. The operations in interval arithmetic are performed on the interval between the two floating point numbers. Interval arithmetic guarantees the presence of the result in the resulting interval by making sure that the rounding mode of the chip during every arithmetic operation is set in the outward direction (see definition 3.1.3). Setting the rounding mode of the chip is an important step in interval arithmetic to justify the guarantee that it makes.

As an example of interval arithmetic, let us consider the expression $u+v = w$, having

the intervals $[0,2]$ and $[3,4]$ for u and v respectively. Evaluating the expression using interval arithmetic we will have $[0,2]+[3,4]=[3,6]$ as the result for w.

3.1 Intervals

In interval constraints we are interested in the intervals that define sets of real values. Such intervals can be empty or non-empty. An empty interval contains no values. A non-empty interval is represented by a pair of values $[a,b]$ where $a \leq b$ and a and b can be any IEEE floating-point value except NaN but including $\pm\infty$ and ± 0.
A non-empty interval is a closed connected set (in the sense of topology) of reals and can be represented as follows:

$$
\begin{array}{ll}
[a,b] & \stackrel{\text{def}}{=} \{x \in \Re \mid a \leq x \leq b\} \\
[-\infty,b] & \stackrel{\text{def}}{=} \{x \in \Re \mid x \leq b\} \\
[a,+\infty] & \stackrel{\text{def}}{=} \{x \in \Re \mid a \leq x\} \\
[-\infty,+\infty] & \stackrel{\text{def}}{=} \Re
\end{array}
$$

Note that $\emptyset$, the empty set, is also considered as a closed connected set.
The soundness of the results in interval arithmetic is guaranteed by always performing the arithmetic in outward rounding. This means that lower bounds are computed by rounding downward(towards $-\infty$) and upper bounds by rounding upward(towards $+\infty$).
Note: In an interval, $-\infty$ never occurs as an upper bound. $+\infty$ never occurs as a lower bound. This is important later, to prevent undefined operations.

Definition 3.1.1 *[20]* **(Real Intervals)** *A set of reals is defined to be a real interval iff the set is closed and connected.*

Definition 3.1.2 *[20]* **(Machine numbers)** *The set $\mathcal{M}$ contains the real number 0 as well as finitely many other reals. In addition it contains two elements that are not reals that are denoted by $-\infty$ and $+\infty$. $\mathcal{M}$ is totally ordered. Any two real elements in $\mathcal{M}$ are ordered as in $\Re$. Also, for all real $x \in \mathcal{M}, -\infty < x < +\infty$.*

Definition 3.1.3 *[20] For any $x \in \Re, x^-$ is the greatest machine number not greater than x; x^+ is the smallest machine number not smaller than x. Obtaining x^- from x is called rounding down, and obtaining x^+ from x is called rounding up.*

Definition 3.1.4 *[20]* **Machine intervals** *A machine interval is any one of the following sets of reals:*

$$\emptyset$$

$$\Re$$

$$\{x \in \Re \mid a \leq x \leq b\} \qquad \textit{(written as } [a, b])$$

$$\{x \in \Re \mid a \leq x\} \qquad \textit{(written as } [a, +\infty])$$

$$\{x \in \Re \mid x \leq b\} \qquad \textit{(written as } [-\infty, b])$$

where a and b are machine numbers. $[-\infty, +\infty]$ *is an acceptable notation for* $\Re$*.*

3.2 Relations, Constraints, and Boxes

Interval constraints usually involve more than one interval representing real numbers in a relation to form a constraint.

Definition 3.2.1 *[20] An n-ary relation (n is a positive integer) is a set of n-tuples of reals.*

For example: $\{< x, y, z > \in \Re^3 \mid x \circ y = z\}$ is a ternary relation, where $\circ$ is a binary operation.

Definition 3.2.2 *[20] An n-ary real box (machine box) is the Cartesian product of n real intervals (machine intervals).*

For example, the ternary box of intervals $[a, b], [c, d]$ and $[e, f]$ is $[a, b] \times [c, d] \times [e, f]$. If $n = 1$ the n-ary real box (machine box) is the real interval (machine interval).

Lemma 3.2.1 *[20] For every n-ary relation, there is a unique least n-ary machine box containing it.*

This existence and uniqueness suggest the definition of a function:

Definition 3.2.3 *[20] Let r be an n-ary relation. Then bx(r) is defined to be the smallest n-ary machine box containing r.*

Lemma 3.2.2 *[20] Let S and T be n-ary relations,* $\forall S, T \subset \Re^n$*, we have* $S \subset T \Rightarrow bx(S) \subset bx(T)$ *(monotonicity of bx).*
$\forall S \subset \Re^n$*, we have* $bx(S) = bx(bx(S))$ *(idempotence of bx).*

3.3 Interval Arithmetic

In [17] the interval operations $X + Y$, $X - Y$ and $X * Y$ are defined by

Definition 3.3.1 *Let X and Y be real intervals and let $\odot$ be one of the operators $+, -,$ or $*$.*

$$X \odot Y = \{x \odot y \mid x \in X \wedge y \in Y\}$$

However, there is no agreement of this definition on interval division as there has to be a special case when $y = 0$. Relational division is a new definition of interval division and is introduced in [21]. The definitions are as follows:

Definition 3.3.2 *Let X and Y be the real intervals, then*

a. the functional quotient of X and Y is defined by:

$$X/Y = \{z \mid \exists x \in X, y \in Y.\ y \neq 0,\ z = x/y\}$$

b. the relational quotient of X and Y is defined by:

$$X \oslash Y = \{z \mid \exists x \in X, y \in Y.\ x = y * z\}$$

[20] argues that relational division makes constraint contraction correct. The following theorem proves that relational division is more conservative than functional division.

Theorem 3.3.1 *Let X and Y be the real intervals, then $X/Y \subseteq X \oslash Y$ and*

$$X \oslash Y = \begin{cases} X/Y & \text{if } 0 \notin X \cap Y \\ \Re & \text{otherwise} \end{cases}$$

The following is the definition for unary interval operations.

Definition 3.3.3 *Let X be a real interval, then*

$$\circ X = \{\circ x \mid x \in X\}$$

where $\circ$ is a unary operator.

The same problem, division by zero, arises with interval exponentiation, which is regarded as a unary operation. The operation is unary as the exponents are not unknowns but known integer values. The problem arises for $b^{1/n}$ when b is negative and n is even. The following are the primitive unary interval arithmetic operations.

Definition 3.3.4 *For all intervals* $[a,b]$ *and integers* n,

$$\begin{aligned}
\leq [a,b] &= \{y \mid \exists x.\ y \leq x \land x \in [a,b]\},\\
\geq [a,b] &= \{y \mid \exists x.\ x \leq y \land x \in [a,b]\},\\
= [a,b] &= \{y \mid \exists x.\ y = x \land x \in [a,b]\},\\
[a,b]^n &= \{y \mid \exists x.\ x^n = y \land x \in [a,b]\}, \text{ and}\\
[a,b]^{1/n} &= \{y \mid \exists x.\ y^n = x \land x \in [a,b]\}.
\end{aligned}$$

From the above definition for $[a,b]^{1/n}$ it can be seen that the case is handled in the same way as in relational division in [17]. For the sake of uniformity Wu [18] defines the binary operation for subtraction in the same way as it is defined for relational division.

Definition 3.3.5 *For all real intervals* $[a,b]$ *and* $[c,d]$,

$$\begin{aligned}
[a,b] + [c,d] &= \{z \mid \exists x,y.\ z = x + y \land x \in [a,b] \land y \in [c,d]\},\\
[a,b] - [c,d] &= \{z \mid \exists x,y.\ x = y + z \land x \in [a,b] \land y \in [c,d]\},\\
[a,b] * [c,d] &= \{z \mid \exists x,y.\ z = x * y \land x \in [a,b] \land y \in [c,d]\}, \text{ and}\\
[a,b] \oslash [c,d] &= \{z \mid \exists x,y.\ x = y * z \land x \in [a,b] \land y \in [c,d]\}.
\end{aligned}$$

The formulas for computing optional IEEE approximations of the four basic arithmetic operations, $+$, $-$, $*$ and $/$ are summarized in Sections 3.3.2, 3.3.3, 3.3.4 and 3.3.5. In [17] the results are in the form of definitions, theorems, corollaries and tables. Only the relevant details have been extracted.

3.3.1 Classification of non-empty real intervals

In [17] real intervals have been classified into different categories depending on their sign. These classifications are useful in defining and avoiding undefined operations, such as $\pm\infty/\pm\infty$ and $0 * \pm\infty$. The four classified real intervals are:

- Class M ("Mixed"): Defined as the set of real intervals containing at least one positive and one negative real. Thus, for all intervals $[a,b]$ in class M, we have $a < 0 < b$.

Class of $[a, b]$	at least one negative	at least one positive	Signs of Bounds
M	yes	yes	$a < 0 \wedge b > 0$
Z	no	no	$a = 0 \wedge b = 0$
P	no	yes	$a \geq 0 \wedge b > 0$
P_0	no	yes	$a = 0 \wedge b > 0$
P_1	no	yes	$a > 0 \wedge b > 0$
N	yes	no	$a < 0 \wedge b \leq 0$
N_0	yes	no	$a < 0 \wedge b = 0$
N_1	yes	no	$a < 0 \wedge b < 0$

Table 3.1. *Classification of intervals by sign. Only non-empty intervals are classified; hence $a \leq b$*

- Class Z ("Zero"): Defined as the set of non-empty real intervals containing neither a positive nor a negative number. In class Z we have $[0, 0]$.
- Class P ("Positive"): Defined as the set of real intervals containing at least one positive, but no negative number. For all intervals $[a, b]$ in class P, $a \geq 0$ and $b > 0$. P is further partitioned into class P_0 for intervals where $a = 0$ and P_1 for intervals where $a > 0$.
- Class N ("Negative"): Defined as the set of real intervals containing at least one negative, but no positive number. For all intervals $[a, b]$ in class N, $a \leq b \leq 0$. Class N is further partitioned into class N_0 for intervals where $b = 0$ and N_1 for intervals where $b < 0$.

  The classification of intervals by sign is shown in Table 3.1.

3.3.2 Interval addition and subtraction

Theorem 3.3.2 *[17] If $[a, b]$ and $[c, d]$ are real intervals, then*

$$[a, b] + [c, d] = [a + c, b + d], \text{ and}$$
$$[a, b] - [c, d] = [a - d, b - c].$$

The theorem assumes that the lower bound is never allowed to be $+\infty$ and the upper bound never allowed to be $-\infty$. This check avoids such undefined operations.

It is guaranteed that this restriction is applied throughout the rest of the interval arithmetic operations that we use. Therefore, $a+c$, $b+d$, $a-d$ and $b-c$ are defined as extended reals.

3.3.3 Interval multiplication

The restriction in interval addition and subtraction was sufficient to prevent undefined operations. In interval multiplication the undefined case is $0 * \pm\infty$. Table 3.2 defines the operations for interval multiplication depending on the classification scheme of the real intervals. From Table 3.2 we can see that the undefined operations such as $0 * \pm\infty$ are avoided. [17] provides the details for the case analysis in Table 3.2.

Class of $[a,b]$	Class of $[c,d]$	Lower Bound of $[a,b]*[c,d]$	Upper Bound of $[a,b]*[c,d]$	Symmetry
N	N	$b*d$	$a*c$	$x*y=-(x*-y)$
N	M	$a*d$	$a*c$	$x*y=-(-x*y)$
N	P	$a*d$	$b*c$	$x*y=-(-x*y)$
M	N	$b*c$	$a*c$	$x*y=-(x*-y)$
M	M	$min(a*d, b*c)$	$max(a*c, b*d)$	*proved directly*
M	P	$a*d$	$b*d$	$x*y=y*x$
P	N	$b*c$	$a*d$	$x*y=-(x*-y)$
P	M	$b*c$	$b*d$	*proved directly*
P	P	$a*c$	$b*d$	*proved directly*
Z	*any*	0	0	*proved directly*
any	Z	0	0	*proved directly*

Table 3.2. *Case analysis for multiplication of real intervals,* $[a,b]*[c,d]$

3.3.4 Interval division

In interval division the undefined operations $\pm\infty/\pm\infty$ and $\pm 0/\pm 0$ should be avoided. They are avoided by using the relational division, restricting the signed zeros defined in the IEEE standard [19]. Table 3.3 summarizes our basis for implementing interval division. A detailed analysis of the expressions in the table can be found in [17].

Class of $[a,b]$	Class of $[c,d]$	The set of $[a,b]/[c,d]$
N_1	N	$[b/c, a/d] \setminus \{0\}$
N_0	N	$[0, a/d]$
M	N	$[b/d, a/d]$
P_0	N	$[b/d, 0]$
P_1	N	$[b/d, a/c] \setminus \{0\}$
N_1	M	$([-\infty, b/d] \cup [b/c, +\infty]) \setminus \{0\}$
N_0	M	$[-\infty, +\infty]$
M	M	$[-\infty, +\infty]$
P_0	M	$[-\infty, +\infty]$
P_1	M	$([-\infty, a/c] \cup [a/d, +\infty]) \setminus \{0\}$
N_1	P	$[a/c, b/d] \setminus \{0\}$
N_0	P	$[a/c, 0]$
M	P	$[a/c, b/c]$
P_0	P	$[0, b/c]$
P_1	P	$[a/d, b/c] \setminus \{0\}$

Table 3.3. *Case analysis for division of real intervals,* $[a,b]/[c,d]$

Disjoint intervals and fused operations Some of the formulas in interval division produce disjoint sets like in cases N_1M and P_1M in Table 3.3. Simplifying this result by combining the two intervals could lead to information loss in computations like $(X/Y) \cap Z$. For example, the operation $[1,1]/[-1,1] \cap [-1/2, 1/2]$ would result in the interval $[-1/2, 1/2]$ instead of $\emptyset$. This operation happens to be the most common operation in interval constraints. A solution for handling such a situation is to use the disjoint intervals internally in the operations. Using this solution for the above example produces an empty set, $\emptyset$. This result is optimal because the intersection operation used the pair of disjoint intervals resulting from the division operation.

3.3.5 Interval exponentiation

Interval exponentiation is a unary operation and the involvement of the exponent, n, brings unexpected complexities to the implementation. The operation yields a number of undefined operations which should be treated properly. Undefined operations such as $0^0, 0^{\frac{1}{0}}, x^{\frac{1}{0}}$, and $\pm 1^{\frac{1}{0}}$ need to be dealt with properly. Wu [18] provides a complete and detailed theoretical foundation for interval exponentiation. Tables 3.4 and 3.5

summarize Wu's work for real interval power and root operations respectively.

Class of $[a, b]$	Value of n	The set of $[a, b]^n$
N_1	$n = 0$	$[1, 1]$
N_0	$n = 0$	$[-\infty, +\infty]$
N	$n \neq 0$	$[min(a^n, b^n), max(a^n, b^n)]$
M	$n > 0 \wedge n \text{ is even}$	$[0, max(a^n, b^n)]$
M	$n > 0 \wedge n \text{ is odd}$	$[a^n, b^n]$
M	$n < 0 \wedge n \text{ is even}$	$[min(a^n, b^n), +\infty] \setminus \{0\}$
M	$n < 0 \wedge n \text{ is odd}$	$[-\infty, a^n] \cup [b^n, +\infty] \setminus \{0\}$
P_1	$n = 0$	$[1, 1]$
P_0	$n = 0$	$[-\infty, +\infty]$
P	$n \neq 0$	$[min(a^n, b^n), max(a^n, b^n)]$
Z	$n > 0$	$[0, 0]$
Z	$n < 0$	$\emptyset$
Z	$n = 0$	$[-\infty, +\infty]$

Table 3.4. *Power of a real interval,* $[a, b]^n$

Class of $[a, b]$	Value of n	The set of $[a, b]^{\frac{1}{n}}$
N	$n > 0 \wedge n \text{ is even}$	$\emptyset$
N	$n > 0 \wedge n \text{ is odd}$	$[a^{\frac{1}{n}}, b^{\frac{1}{n}}]$
N	$n < 0 \wedge n \text{ is even}$	$\emptyset$
N	$n < 0 \wedge n \text{ is odd}$	$[b^{\frac{1}{n}}, a^{\frac{1}{n}}] \setminus \{0\}$
M	$n > 0 \wedge n \text{ is even}$	$[-\lvert b^{\frac{1}{n}}\rvert, \lvert b^{\frac{1}{n}}\rvert]$
M	$n > 0 \wedge n \text{ is odd}$	$[a^{\frac{1}{n}}, b^{\frac{1}{n}}]$
M	$n < 0 \wedge n \text{ is even}$	$[-\infty, -\lvert b^{\frac{1}{n}}\rvert] \cup [\lvert b^{\frac{1}{n}}\rvert, +\infty] \setminus \{0\}$
M	$n < 0 \wedge n \text{ is odd}$	$[-\infty, -\lvert a^{\frac{1}{n}}\rvert] \cup [\lvert b^{\frac{1}{n}}\rvert, +\infty] \setminus \{0\}$
P	$n > 0 \wedge n \text{ is even}$	$[-\lvert b^{\frac{1}{n}}\rvert, -\lvert a^{\frac{1}{n}}\rvert] \cup [\lvert a^{\frac{1}{n}}\rvert, \lvert b^{\frac{1}{n}}\rvert]$
P	$n > 0 \wedge n \text{ is odd}$	$[a^{\frac{1}{n}}, b^{\frac{1}{n}}]$
P	$n < 0 \wedge n \text{ is even}$	$[-\lvert a^{\frac{1}{n}}\rvert, -\lvert b^{\frac{1}{n}}\rvert] \cup [\lvert b^{\frac{1}{n}}\rvert, \lvert a^{\frac{1}{n}}\rvert] \setminus \{0\}$
P	$n < 0 \wedge n \text{ is odd}$	$[\lvert b^{\frac{1}{n}}\rvert, \lvert a^{\frac{1}{n}}\rvert] \setminus \{0\}$
Z	$n > 0$	$[0, 0]$
Z	$n < 0$	$\emptyset$

Table 3.5. *Root of a real interval,* $[a, b]^{\frac{1}{n}}$

3.4 Interval Constraints

Interval constraints is a more general method compared to interval arithmetic. In an interval constraint system problems are solved by applying contraction operators to the intervals associated with the real-valued unknowns. The contraction operators remove as much as possible the inconsistent values. Constraints share variables. Therefore, the contraction operator has to be applied again to all the constraints sharing that variable. Because changes are contractions of the intervals and the intervals are represented by floating-point numbers, a finite number of contractions suffices to reach a state where all constraints reach a null contraction.

As the contraction operator removes only the inconsistent values and may not remove all such values, it may happen that the resulting interval contains no solution. Thus the results obtained have the meaning: if a solution exists, then it is in the resulting intervals.

As an example for constraint contraction, let us consider the *primitive constraint* (see Sub-section 3.4.1) $u + v = w$, with intervals $[0, 2], [0, 2]$ and $[3, 5]$ for u, v and w respectively. All three intervals contain inconsistent values. Further, $v \leq 2$ and $w \geq 3$ imply that $u = w - v \geq 1$. Therefore u cannot be less than 1. Similarly, values less than 1 for v and values greater than 4 for w are inconsistent and are ruled out from the current interval. Removing the inconsistent values from the given intervals, we have the intervals $[1, 2]$ for u and $[3, 4]$ for w. To obtain the new intervals from the old ones the contraction operator associated with the constraint is applied, which in this case is $u + v = w$. Removing the inconsistent values from the intervals involved with the constraint is called *constraint contraction.* The new bounds 1 and 4 are computed from the rules of interval arithmetic which require the rounding direction of the processor to be set appropriately. Thus interval constraints depend on interval arithmetic. Figure 3.1 shows the contraction of the initial intervals for this example. In the figure, box B is the cartesian product of the initial intervals associated with u, v and w, and *sum* is the relation associated with the constraint $u + v = w$. The result of contracting the box B is the triangle labeled $sum \cap B$. The box $\gamma_{sum}(B)$ is the smallest machine box containing the triangle. The symbol γ is defined as the *constraint contraction operator* and is discussed in Section 3.4.2.

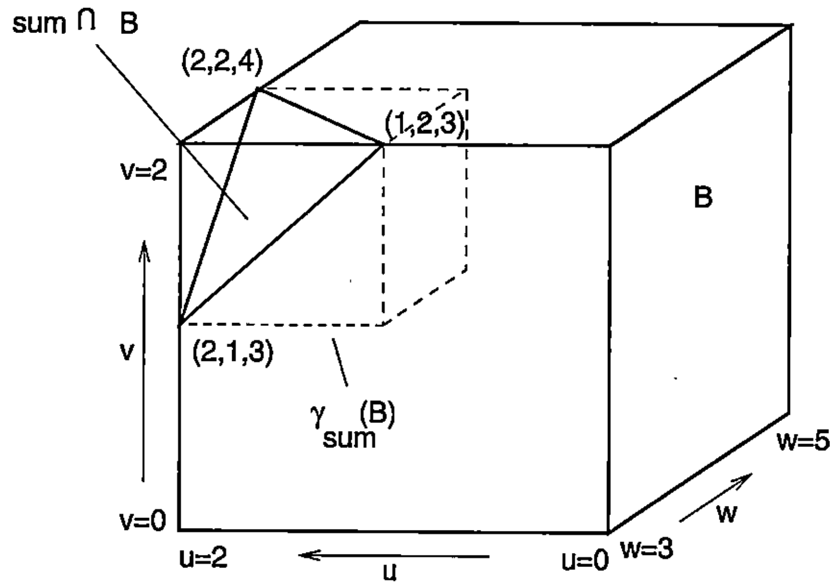

Figure 3.1. *Interval constraint contraction*

$$x + y = z$$
$$x * y = z$$
$$x^n = y \text{ for integer n}$$
$$x = y$$
$$x \leq y$$

Figure 3.2. *Primitive Constraints.*

3.4.1 Primitive constraints

Interval constraints are introduced in this section beginning with an example. Let us consider the problem of computing the x and y coordinates of the intersection of a circle and a parabola. Eliminating the negative values of x from the solution we have the following interval constraint system:

$$x^2 + y^2 = 1 \wedge y = x^2 \wedge 0 \leq x \tag{3.1}$$

This is a conjunction of three logical formulas related by sharing x and y. The

above model can be translated to primitive constraints by introducing auxiliary variables x_2 and y_2:

$$x^2 = x_2 \wedge y^2 = y_2 \wedge x_2 + y_2 = 1 \wedge x_2 = y \wedge 0 \leq x \tag{3.2}$$

The most commonly used primitive constraints are shown in a table in Figure 3.2. Each of the variables in Formula 3.2, including the auxiliary variables, is regarded as an unknown real. Each real unknown is associated with an interval containing all the values of this real that might contain in a solution. The formal definitions of the set of constraints given in [22, 17] are as follows:

$$\begin{aligned}
sum &\stackrel{\text{def}}{=} \{(x, y, z) \in \mathcal{R}^3 \mid x + y = z\} \\
prod &\stackrel{\text{def}}{=} \{(x, y, z) \in \mathcal{R}^3 \mid x * y = z\} \\
eq &\stackrel{\text{def}}{=} \{(x, y) \in \mathcal{R}^2 \mid x = y\} \\
leq &\stackrel{\text{def}}{=} \{(x, y) \in \mathcal{R}^2 \mid x \leq y\} \\
power_n &\stackrel{\text{def}}{=} \{(x, y) \in \mathcal{R}^2 \mid x^n = y\} \text{ for integer n}
\end{aligned}$$

where *sum* and *prod* are ternary relations, while *eq*, *leq* $power_n$ are binary relations.

3.4.2 The constraint contraction operator

The constraint contraction operator contracts intervals by removing inconsistent values. The constraint contraction operator, γ, defined for each constraint in [20] is summarized in this section.

Definition 3.4.1 *The constraint contraction operator γ_r associated with an n-ary relation r acting on an n-ary machine box B is defined as $\gamma_r(B) = bx(r \cap B)$.*

where r is *sum*, *prod*, *eq*, *leq*, or $power_n$, and B is the machine box $abcdef \stackrel{\text{def}}{=} [a, b] \times [c, d] \times [e, f]$ or the machine box $abcd \stackrel{\text{def}}{=} [a, b] \times [c, d]$, for a, b, c, d, e, $f \in \mathcal{M}$ and $a \leq b$, $c \leq d$, $e \leq f$.

The following are the properties of a constraint contraction operator [22]:

1. *Contractance*: The contracted intervals are contained in the original intervals. Thus, the contraction operator always contracts the intervals or keeps them unchanged.

2. *Correctness*: Every solution lies in the contracted intervals.
3. *Monotonicity*: The contraction preserves inclusion.
4. *Idempotence*: Applying the contraction operator more than once in immediate succession yields the same result.

Each of the primitive relations has its own contraction operator. Definition 3.4.1 can be applied to specify the contraction operator for each of the primitive relations.

The projection of the box $r \cap B$ is used to represent $bx(r \cap B)$. Each projection of the box corresponds to a smaller box in Figure 3.1. The projections represent the box that contains $r \cap B$. The formulas for the projections are based on the fused interval arithmetic operations described in Section 3.3.4, and are defined as follows [20]:

Lemma 3.4.1 *For all $a, b, c, d, e, f \in \mathcal{M}$:*

$$\pi_1(sum \cap abcdef) = [a,b] \cap ([e,f] - [c,d])$$
$$\pi_2(sum \cap abcdef) = [c,d] \cap ([e,f] - [a,b])$$
$$\pi_3(sum \cap abcdef) = [e,f] \cap ([a,b] + [c,d])$$

$$\pi_1(prod \cap abcdef) = [a,b] \cap ([e,f]/[c,d])$$
$$\pi_2(prod \cap abcdef) = [c,d] \cap ([e,f]/[a,b])$$
$$\pi_3(prod \cap abcdef) = [e,f] \cap ([a,b] * [c,d])$$

where $\pi_i(X)$ is the i-th projection of the relation X.

Lemma 3.4.2 *For all $a, b, c, d \in \mathcal{M}$ and all integers n:*

$$\pi_1(eq \cap abcd) = [a,b] \cap = ([c,d])$$
$$\pi_2(eq \cap abcd) = [c,d] \cap = ([a,b])$$

$$\pi_1(leq \cap abcd) = [a,b] \cap \leq ([c,d])$$
$$\pi_2(leq \cap abcd) = [c,d] \cap \geq ([a,b])$$

$$\pi_1(power_n \cap abcd) = [a,b] \cap [c,d]^{1/n}$$
$$\pi_2(power_n \cap abcd) = [c,d] \cap [a,b]^n$$

where $\pi_i(X)$ is the i-th projection of the relation X.

Lemma 3.4.3 *$bx(\pi_i(r)) = \pi_i(bx(r))$, where r is an n-ary relation and $i = 1, \ldots, n$.*

From the above lemma and definitions of the projections over the primitive relations the following implementations of the contraction operator γ can be considered based on interval arithmetic.

Theorem 3.4.1 *For all $a, b, c, d, e, f \in \mathcal{M}$:*

$$\pi_1(\gamma_{sum}(abcdef)) = [a, b] \cap [(e-d)^-, (f-c)^+]$$
$$\pi_2(\gamma_{sum}(abcdef)) = [c, d] \cap [(e-b)^-, (f-a)^+]$$
$$\pi_3(\gamma_{sum}(abcdef)) = [e, f] \cap [(a+c)^-, (b+d)^+]$$

Since interval division $[a, b]/[c, d]$ sometimes produces disjoint intervals, it needs to be handled differently to achieve optimal contraction in interval constraints.

Theorem 3.4.2 *For all $a, b, c, d, e, f \in \mathcal{M}$:*

$$\begin{aligned}
\pi_1(\gamma_{prod}(abcdef)) &= bx([a,b] \cap (RI_1 \cup RI_2)) && if [e,f]/[c,d] = RI_1 \cup RI_2, \\
&\subseteq [a,b] \cap [v^-, w^+] && if [e,f]/[c,d] = [v,w], \\
\pi_2(\gamma_{prod}(abcdef)) &= bx([c,d] \cap (RI_1 \cup RI_2)) && if [e,f]/[a,b] = RI_1 \cup RI_2, \\
&\subseteq [c,d] \cap [v^-, w^+] && if [e,f]/[a,b] = [v,w], \\
\pi_3(\gamma_{prod}(abcdef)) &\subseteq [e,f] \cap [l^-, u^+] && if [a,b] * [c,d] = [l,u]
\end{aligned}$$

where $\pi_i(X)$ is the i-th projection of the box X, and l, u are reals, where $RI_1 \cup RI_2$ is a union of two disjoint connected sets of reals, and $[v, w]$ is a connected sets of reals. RI_1, RI_2, v and w are computed from [e,f]/[c,d] (or [e,f]/[a,b]) based on the expressions provided in Table 3.3. l and u are reals computed from [a,b][c,d] based on the expressions provided in Table 3.2.*

Theorem 3.4.3 *For all $a, b, c, d \in \mathcal{M}$:*

$$\pi_1(\gamma_{eq}(abcd)) = \pi_2(\gamma_{eq}(abcd)) = [a, b] \cap [c, d]$$
$$\pi_1(\gamma_{leq}(abcd)) = [a, b] \cap [-\infty, d]$$
$$\pi_2(\gamma_{leq}(abcd)) = [c, d] \cap [a, +\infty]$$

where $\pi_i(X)$ is the i-th projection of the box X.

Theorem 3.4.4 *For all $a, b, c, d \in \mathcal{M}$ and integer n:*

$$\begin{aligned}
\pi_1(\gamma_{power_n}(abcd)) &= bx([a,b] \cap (RI_1 \cup RI_2)) && if [c,d]^{1/n} = RI_1 \cup RI_2 \\
&\subset [a,b] \cap [l^-, u^+] && if [c,d]^{1/n} = [l,u] \\
\pi_2(\gamma_{power_n}(abcd)) &= bx([c,d] \cap (RI_3 \cup RI_4)) && if [a,b]^n = RI_3 \cup RI_4 \\
&\subset [c,d] \cap [l^-, u^+] && if [a,b]^n = [l,u]
\end{aligned}$$

where $\pi_i(X)$ is the i-th projection of the box X, l, u are reals, $RI_1 \cup RI_2$ and $RI_3 \cup RI_4$ is a union of two disjoint connected sets of reals, not necessarily closed. RI_1, RI_2, v and w are computed from $[c,d]^{1/n}$ based on the expressions in Table 3.5. RI_3, RI_4, l and u are computed from $[a,b]^n$ based on the expressions in Table 3.4.

3.4.3 The interval constraint system

An interval constraint system is useful for analyzing sets of equalities or inequalities between real-valued expressions. Interval constraint systems are defined in [22, 20] and have the following components.

- **A set of primitive constraints** Each constraint is an expression of the form $p(x_1, \ldots, x_n)$, where p is an n-ary relation($n = 2$ or 3). x_i is either a literal or a symbol denoting an unknown real. If x_i is a symbol it may occur in another constraint denoting the same real. The relationship p is one of the primitive constraints defined in Section 3.4.1
- **A sequence of unknowns** The sequence of unknown reals occurring in the elements of the set of constraints. In an interval constraint system the unknowns are usually shared by two or more elements of the set of primitive constraints.
- **A state** is a sequence of machine intervals. For $i = 1, \ldots, n$, the i-th unknown is known to be the i-th interval. A state should be thought of as a state of information about the sequence of unknowns. The state of a system S is said to be *stable* if for every constraint $p(x_1, \ldots, x_n) \in S$, $\gamma_r(B) = B$, where B is a machine box containing the cartesian product of the intervals associated with $< x_1, \ldots, x_n >$, and r is the relation associated with p.

Initially each of the unknown reals in the constraint system is associated with an interval containing all the values that might occur in a solution. The system starts with

intervals large enough to contain all the solutions of interest. The contraction operator then removes only the inconsistent values from the intervals and may not remove all such values. The contraction operation is repeated until the intervals no longer contract. The repetition is performed by means of an algorithm discussed in the next section. As the process does not always remove all the inconsistent values the resulting interval may contain no solution. Therefore, the results of interval constraints have the meaning: *if* a solution exists, then it is in the resulting intervals [20].

3.4.4 Constraint propagation

The constraint propagation algorithm is the vital part for solving an interval constraint system. The purpose of the constraint propagation algorithm is to remove as much as possible the inconsistent values from the intervals present in the interval constraint system. A stable system is achieved by repeatedly applying the contraction operator on each of the constraints in order to contract the intervals associated with the unknowns of the constraint. When an interval for an unknown is contracted, the effect of the contraction is propagated to all other constraints that share the unknown. Contraction is applied until the constraint system reaches a stable state and no further contraction can be applied to the constraints. It is worth noting that a stable state does not necessarily mean that a solution has been found. It only means that if the system has any solutions, then they are in the resulting intervals.
The Waltz algorithm (see Figure 3.3) is used for the propagation. In the algorithm r is the relation associated with a constraint C, π_i is the i-th projection of the intervals involved in C. B is normally infinite in all directions unless otherwise it is initialized.

The algorithm starts with a set Q of active constraints containing the complete set of constraints in the system. In every iteration the constraint C is extracted from Q and the contraction operator, γ_r associated with the constraint C is applied. The constraint that has just been extracted is said to be inactive. As a result of this contraction the intervals of the unknowns might have contracted causing a change in state from B to B'. $\pi(B)$ indicates the interval for the i-th unknown before contraction. $\pi(B')$ indicates the contracted interval for the i-th unknown after the contraction operator has been applied. If for any of the unknowns the contracted interval becomes empty, then the system has no solution and the propagation stops.

begin $Q \leftarrow$ the set of all constraints
  while $Q \neq \emptyset$ **do**
    remove constraint C from Q
    $B' \leftarrow \gamma_r(B)$
    if $\exists i : \pi_i(B') = \emptyset$ **then** exit with failure
    for each variable x_i that occurs in C **do**
      if $\pi_i(B') \neq \pi_i(B)$ **then**
        for each constraint $C' \neq C$ which has x_i as an argument **do**
          add C' to Q unless $C' \in Q$
    endfor
    $B \leftarrow B'$
  endwhile
end

Figure 3.3. *Waltz constraint propagation algorithm*

If the interval for the unknown, x_i, resulted in a contraction, then all the constraints that share this unknown and are inactive should be reactivated. A constraint is said to be activated after it is inserted in Q. The algorithm proceeds in this way until Q becomes empty. This represents the termination condition for the system and indicates that a stable state has been reached and no further contraction is possible. The algorithm has the following properties:

- The algorithm always terminates, since the intervals are contracted by the contraction operator and there exists a finite number of machine intervals.
- The algorithm gives a set of intervals containing all the solutions. Even if the propagation had been interrupted abruptly due to time constraints.
- The algorithm reaches a fixed point which is unique and does not depend on the order in which the constraints were contracted.
- A constraint may be added to an already propagating system. The unknowns in this constraint are updated when the contraction operator for this constraint is applied. This algorithm is well suited for incremental systems.

Chapter 4

Interval arithmetic and constraint system components

COM components are provided to perform interval arithmetic operations and create and propagate a constraint system. The components are defined in the Interface Definition Language. The component classes contain a list of interfaces that are exported by instances of components. The component classes always appear in the context of a type library definition. The interfaces for interval arithmetic and the primitive constraints are implemented in C++ classes. Design details of the component classes, interface definitions and the implementation of the interfaces are discussed in this chapter.

All the component classes pertaining to an interval constraint system are defined in a type library named `ICSystemLib`. The IDL source code of the type library is as follows:

```
[uuid(713E8EE1-860B-11D5-BC73-00104B68FD3C), version(1.0),
 helpstring("Interval constraint system type library.")]
library ICSystemLib {
  importlib("stdole32.tlb");
  importlib("stdole2.tlb");
};
```

4.1 Interval arithmetic component

Corresponding to the mathematical concept of real with an unknown value we define a component COReal and expose the functionality of the component through an interface IReal. The mathematical concept has been implemented in [18] through a C++ class called real. We take advantage of software reuse and reuse the class real by making minimal changes.

Interfaces in COM can be implemented as C-style functions or C++ classes. The C++ class CImpIReal implements the interface IReal. The simplest way to reuse the real with minimal changes is to let the class CImpIReal inherit real. Hence, CImpIReal implements IReal and inherits real.

Additional methods that allow the user to interact with the lower and upper bounds of real are defined and implemented in CImpIReal. The additional methods are exposed to the user through the IReal interface. The operator overloaded functions declared in real are used internally by the constraint contraction operators only. Therefore, we do not make any effort to expose the functionality of these overloaded operators through IReal. It is also worth pointing out that operator or function overloading, a feature in C++, is not supported in the IDL. Following is the IDL source code for the component class COReal and the interface IReal:

```
[uuid(713E8EF1-860B-11D5-BC73-00104B68FD3C),
 helpstring("Component object class")]
 coclass COReal {
    [default] interface IReal;
 };
[object, uuid(713E8EF0-860B-11D5-BC73-00104B68FD3C),
 dual, helpstring("Real Interface"), pointer_default(unique)]
interface IReal : IDispatch {
  HRESULT init();
  HRESULT initr([in]IReal *real);
  HRESULT initd([in]double lu);
  HRESULT initdd([in]double lb, [in]double ub);
  HRESULT isLbGtEqUb([out,retval]BOOL *result);
  HRESULT compare([in]double x, [in]double y,
```

```
                   [out,retval]int *result);
  HRESULT incrBound([in]BOOL lowerBound, [in]int incr,
                    [in]double min, [in]double max,
                    [in]int range, [in]double initialValue,
                    [out,retval]BOOL *result);
  HRESULT rtoi([in]BOOL lowerBound, [in]double oldr, [in]double newr,
               [in]int min, [in]int max, [in]double lb,
               [in]double ub, [in]int oldi, [in, out]int *newi,
               [out, retval]BOOL *result);
  HRESULT setLB([in] double lb);
  HRESULT setUB([in] double ub);
  HRESULT setBounds([in]double lb, [in] double ub);
  HRESULT resetBounds();
  HRESULT getPosINF([out, retval]double *posINF);
  HRESULT getNegINF([out, retval]double *negINF);
  HRESULT getLB([out, retval] double *pLB);
  HRESULT getUB([out, retval] double *pUB);
  HRESULT getInitialBounds([out]double *pLB, [out] double *pUB);
  HRESULT getBounds([out]double *pLB, [out] double *pUB);
  HRESULT getDecLB([out, retval] BSTR *pLB);
  HRESULT getDecUB([out, retval] BSTR *pUB);
  HRESULT getHexLB([in]BOOL bias, [out, retval] BSTR *pLB);
  HRESULT getHexUB([in]BOOL bias, [out, retval] BSTR *pUB);
  HRESULT toDecString([out, retval] BSTR *pDec);
  HRESULT toHexString([in]BOOL bias, [out, retval] BSTR *pHex);
};
```

4.1.1 Implementation

The following were the modifications made to the class `real`:

- Following the naming convention given in Table 4.1, renamed the class `real` to `Real`, and
- Added the following member variables and methods:

Type	Convention	Example
Class	First letter of each word is capitalized	`Real`
Method	First word is lowercase. Additional words have first letter capitalized	`getBounds`
Variable	First letter lowercase. Additional words have first letter capitalized	`lb`
Constant	All letters uppercase with words separated by underscores	`POS_INF`
Globals	Starting with a lower case g followed by an underscore	`g_ics`

Table 4.1. *Naming convention for the source code in this thesis*

```
protected:
  double initLB, initUB;
  vector<CImplConstraint*> vConstraints;

public:
  void resetBounds();
  bool addConstraint(CImplConstraint* c)  ;
  vector<CImplConstraint*>& getConstraints();
```

- Replaced the operator overloaded method

  `Interval* operator/(const Real&);`

  with

  `Real divIntersect(const Real &, const Real &);`

  This method performs the fused operation for division that was discussed in Section 3.3.4.

The `Real` class can be instantiated with lower and upper bounds. The variables `initLB` and `initUB` store the values of the initial lower and upper bounds respectively. All the constraints that this `Real` is shared by is stored in `vConstraints`. The original lower and upper bounds of this `Real` can be restored by calling the method `resetBounds`. Given an instance of a `Constraint`, it can be added to `Real` by calling

the method addConstraint. The method getConstraints returns a list of all the Constraints in which this instance of Real occurs.

The following is the C++ interface specification of the class CImplReal:

```
class ATL_NO_VTABLE CImpIReal :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CImpIReal, &CLSID_COReal>,
  public IDispatchImpl<IReal, &IID_IReal, &LIBID_ICSystemLib>,
  public ISupportErrorInfo, public Real {
public:
  DECLARE_REGISTRY_RESOURCEID(IDR_REAL)
  DECLARE_PROTECT_FINAL_CONSTRUCT()
BEGIN_COM_MAP(CImpIReal)
  COM_INTERFACE_ENTRY(IReal)
  COM_INTERFACE_ENTRY(IDispatch)
  COM_INTERFACE_ENTRY(ISupportErrorInfo)
END_COM_MAP()
// ISupportErrorInfo
  STDMETHOD(InterfaceSupportsErrorInfo)(REFIID riid);
// IReal
public:
  CImpIReal() : Real() { }
  CImpIReal(const Real &r) :Real(r) { }
  CImpIReal(double lu) : Real(lu) { }
  CImpIReal(double lb, double ub) : Real(lb, ub) { }
  STDMETHOD(init)();
  STDMETHOD(initr)(IReal *real);
  STDMETHOD(initd)(double lu);
  STDMETHOD(initdd)(double lb, double ub);
  STDMETHOD(isLbGtEqUb)(BOOL *result);
  bool isLeftGtEqRight(double left, double right);
  bool isLeftGtRight(double left, double right);
  inline bool isEqual(double x, double y) {
```

```
    return (ABS(x - y) < Epsilon);
  }
  inline bool isLessOrEqual(double x, double y) {
    return (x - y < Epsilon);
  }
  STDMETHOD(compare)(double x, double y, int *result);
  STDMETHOD(incrBound)(BOOL lowerBound, int incr,
                       double min, double max,
                       int range, double initialValue, BOOL *result);
  STDMETHOD(rtoi)(BOOL lowerBound, double oldr, double newr, int min,
                  int max, double lb, double ub, int oldi,
                  int *newi, BOOL *result);
  STDMETHOD(getPosINF)(double *posINF);
  STDMETHOD(getNegINF)(double *negINF);
  STDMETHOD(setLB)(double lb);
  STDMETHOD(setUB)(double ub);
  STDMETHOD(setBounds)(double lb, double ub);
  STDMETHOD(resetBounds)();
  STDMETHOD(getLB)(double *pLB);
  STDMETHOD(getUB)(double *pUB);
  STDMETHOD(getInitialBounds)(double *pLB, double *pUB);
  STDMETHOD(getBounds)(double *pLB, double *pUB);
  STDMETHOD(getDecLB)(BSTR *pLB);
  STDMETHOD(getDecUB)(BSTR *pUB);
  STDMETHOD(getHexLB)(BOOL bias, BSTR *pLB);
  STDMETHOD(getHexUB)(BOOL bias, BSTR *pUB);
  STDMETHOD(toDecString)(BSTR *pDec);
  STDMETHOD(toHexString)(BOOL bias, BSTR *pHex);
};
```

Figure 4.1 shows the COM notation for the `CImpIReal` class. Note that `CImpIReal` objects expose two interfaces; `IReal` and `IDispatch`.

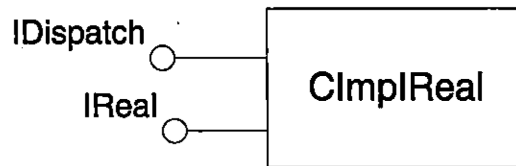

Figure 4.1. *COM notation for `CImpIReal` class*

4.2 Interval constraint system component

Ideally, an interval constraint system component should allow the user to add all the unknowns and constraints involved in a constraint system, activate all the constraints and start the propagation. It is an arduous and error-prone task for a user to create an interval constraint system for every mathematical model under consideration. To ease the task we provide the following solution:

- take the mathematical model from the user as input,
- create a constraint system, and
- return the constraint system to the user.

In the implementation the first two steps of the solution have been merged into one which is implemented in a parser component. The details about the component are given in Chapter 5. The parser takes the mathematical model as an input string, parses the input string, creates a constraint system and returns an instance of the constraint system to the user. The constraint system can then be solved by the user simply by initiating the propagation algorithm. The initiation is the only part of the constraint system that needs to be exposed to the user. The interval constraint system component class, `COIICSystem` implements and exposes this functionality through the interface `IICSystem`. Following are the definitions of the component class and the interface in IDL:

```
[uuid(713E8EF8-860B-11D5-BC73-00104B68FD3C),
 helpstring("Component object class")]
 coclass COICSystem {
 [default] interface IICSystem;
 };
```

```
[object, uuid(713E8EF7-860B-11D5-BC73-00104B68FD3C),
 dual, helpstring("Interval Constraint system interface"),
 pointer_default(unique)]
 interface IICSystem : IDispatch {
  [id(1), helpstring("Activate the constraint system.")]
  HRESULT activateAll();
  [id(2), helpstring("Apply the propagation algorithm to
  constraint system. Returns true if the propagation
  succeeds.")]
  HRESULT propagate([out,retval]BOOL *x);
  [id(3), helpstring("Checks the state of the constraint system.")]
  HRESULT done([out, retval]BOOL *x);
  [id(4), helpstring("Resets all the unknowns in this system
  to the intervals that each of the real in this system was
  initialized with.")]
  HRESULT resetUnknowns();
  [id(5), helpstring("Prints all the reals in this constraint
  system.")]
  HRESULT printReals();
};
```

4.2.1 Implementation

Each primitive constraint is associated with a contraction operator. In [18] Wu has defined an abstract class called `constraint`. It is natural to define the abstract class to encapsulate the general concept of a constraint. Each individual constraint should be of a certain type. For this reason subclasses `sum`, `prod`, `eq`, `leq` and `powerN` are derived classes to represent the actual primitive constraints. The class names represent their associated primitive constraints. Figure 4.2 shows the class hierarchy diagram for the constraints.

The source code for the abstract class `constraint` and the primitive constraints have been reused with minimal changes in our constraint system. The following is the C++ interface definition of the `constraint` class in [18]:

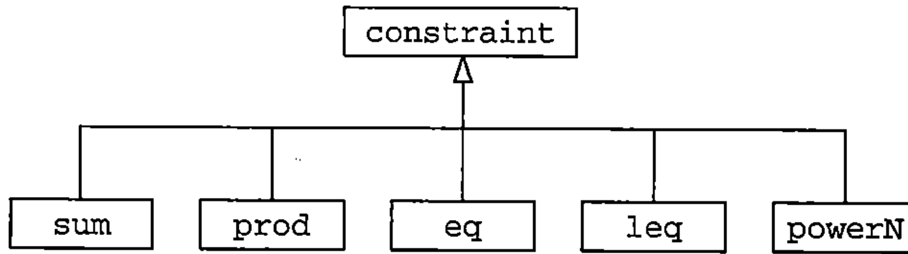

Figure 4.2. *Class hierarchy diagram for the primitive constraints*

```
class constraint {
public:
  virtual bool shrink() = 0; //contraction function, to be overridden
};
```

has been modified to:

```
class Constraint {
public:
  //contraction function, to be overridden
  virtual bool shrink(vector<Real*> &vReal) = 0;
  virtual bool has(Real *r) = 0;
};
```

The contraction function, `shrink` now takes a `vector` of `Real`s as an argument. The vector contains all the unknowns that may have contracted after the contraction had been applied to the constraint. The vector remains unchanged if none of the unknowns for the constraint have contracted. `has()` takes an instance of a `Real` as an argument and returns `true` if the `Real` is in the constraint, `false` otherwise.

The following is the C++ class interface definition for one such derived class, namely `Sum`. Similar class interfaces have been defined for `Prod`, `Eq`, `Leq` and `PowerN`.

```
class Sum: public Constraint {
private:
  Real *x, *y, *z;
public:
  Sum() { }
  Sum(Real *x, Real *y, Real *z);
```

```
  ~Sum() { }
  bool shrink(vector<Real*> &vReal);
  bool has(Real *r);
};
```

Instances of these primitive constraints are created by the constraint system component.

Interfaces in COM are immutable. To add new functionality to an existing component, a new interface has to be defined and implemented. Anticipating the addition of new primitive constraints to the constraint system, to avoid the introduction of new interfaces to reflect the change, and for modularity purposes we design two C++ classes. One class encapsulates the functionality for creating the constraints. The other class encapsulates the functionality for activating and propagating the constraint system, and exposing its functionality through an interface. The following is the interface specification of the C++ class that creates a constraint system:

```
class ConstraintSystem {
protected:
  vector<Real*> vReals;
  vector<Constraint*> vConstraints;
  Queue<Constraint*> qac; //queue of active constraints
public:
  Constraint *addConstraintSum(Real *x,Real *y,Real *z);
  Constraint *addConstraintProd(Real *x,Real *y,Real *z);
  Constraint *addConstraintPowerN(Real *x,Real *y,int n);
  Constraint *addConstraintLeq(Real *x, Real *y);
  Constraint *addConstraintEq(Real *x, Real *y);
  bool contains(Constraint *pC, Real *r);
  bool addConstraintForReal(Constraint *pC , Real *r);
  bool addUnknown(Real *r);
  bool findReal(Real *r);
  bool findConstraint(Constraint *pC);
};
```

The other class, `CImplICSystem`, encapsulates the constraint system's activation

and propagation, implements the interface IICSystem, and inherits ConstraintSystem. The following is the interface specification of the C++ class:

```
class ATL_NO_VTABLE CImpIICSystem :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CImpIICSystem, &CLSID_COICSystem>,
  public IDispatchImpl<IICSystem, &IID_IICSystem, &LIBID_ICSystemLib>,
  public ConstraintSystem
{
public:
DECLARE_REGISTRY_RESOURCEID(IDR_ICSYSTEM)
DECLARE_PROTECT_FINAL_CONSTRUCT()
BEGIN_COM_MAP(CImpIICSystem)
  COM_INTERFACE_ENTRY(IICSystem)
  COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()
// IICSystem
public:
  STDMETHOD(printReals)();
  STDMETHOD(done)(BOOL *b);
  STDMETHOD(propagate)(BOOL *b);
  STDMETHOD(activateAll)();
  STDMETHOD(resetUnknowns)();
};
```

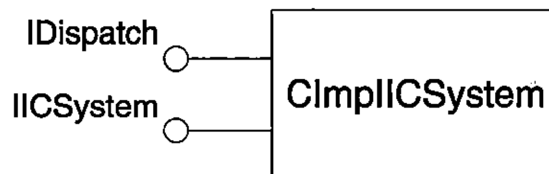

Figure 4.3. *COM notation for CImpIICSystem class*

Figure 4.3 shows the COM notation for the CImpIICSystem class. Note that CImpIICSystem objects expose the interfaces IICSystem and IDispatch.

The interface that CImpIICSystem implements exposes the functionality needed for activating and propagating a constraint system. The functionality that is exposed by IICSystem can only be accessed from the automation controllers. However, from within the automation server, complete access including creating, activating and initiating the propagation of the constraint system can be obtained. This is achieved by using the reinterpret_cast cast operator on the pointer to the interface, IICSystem. The following code snippet shows the conversion from an interface pointer of type IICSystem to type CImpIICSystem:

```
IICSystem *pIIcs;
CImpIICSystem *ics;
ics = reinterpret_cast<CImpIICSystem *>(pIIcs);
```

Support for new primitive constraints (e.g. trigonometric functions) can be added to the existing list of constraint classes later. The new constraint will be created and added to the constraint system by the ConstraintSystem class. This addition neither affects the interface definition, IICSystem, that is already published nor creates a need for a new interface. Thus, one can keep the vtbl layout of the component intact.

Chapter 5

Interval Constraint System Expression Parser

Translating a mathematical problem to its corresponding set of primitive constraints is a nontrivial task. It would be a tedious task for users to manually translate every mathematical problem into the set of primitive constraints discussed in Chapter 4. Automating the translation process makes the system easier to use and appropriate for any application that uses the system. We design a parser module that automates the process by taking in a mathematical model, in the form of mathematical expressions, and returning to the user a constraint system for the model. In this chapter we discuss the design and implementation of this expression parser module.

The expression parser has to depend on a software structure to allow other programmers to read, maintain, modify, test and port the module easily. The software structure can be brought about in several ways. One way is to adopt a software architecture or to use a design pattern. Another way is to follow a programming paradigm. There are several programming paradigms: data-flow programming, functional programming, relational programming and Data-Directed Design [23]. We use Data-Directed Design(DDD) for the module design.

5.1 Syntax graphs

To start with, we take the readily available syntax graphs [24] for the Pascal compiler. The syntax graph follows one-symbol-lookahead without backtracking. This means that the choice of every analysis step depends on the current state and on the next

symbol being read. During this process no step will ever be revoked later on.

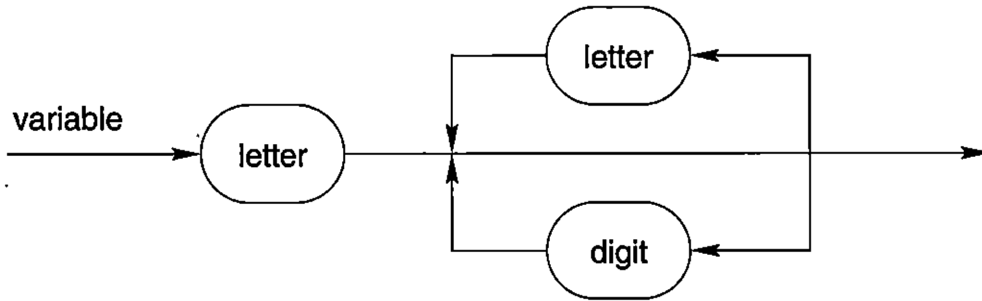

Figure 5.1. *Variable*

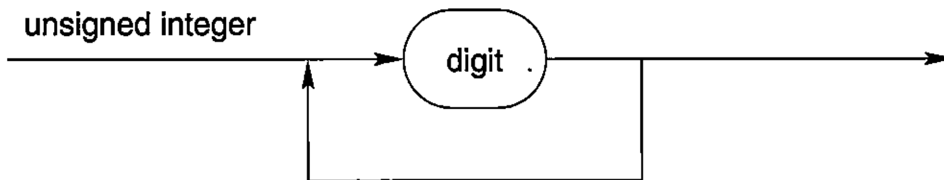

Figure 5.2. *Unsigned Integer*

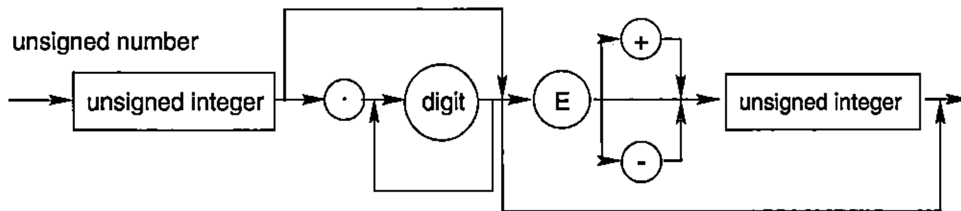

Figure 5.3. *Unsigned Number*

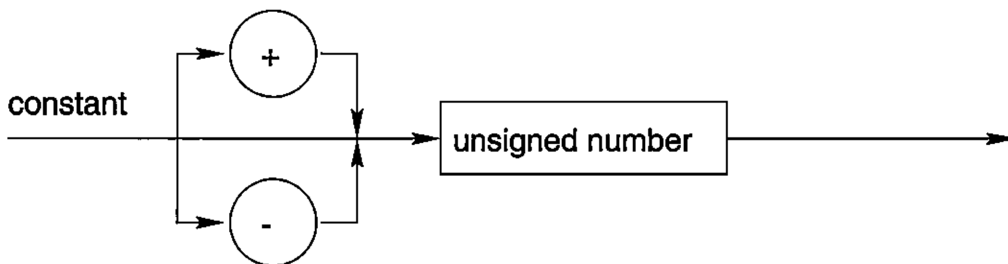

Figure 5.4. *Constant*

The syntax graphs shown in Figures 5.1 - 5.10 were obtained by removing all the unnecessary components from the original syntax diagrams for the Pascal compiler.

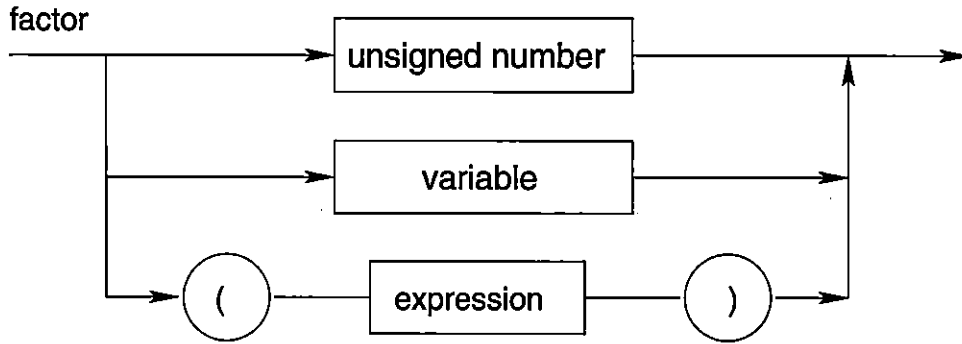

Figure 5.5. *Factor*

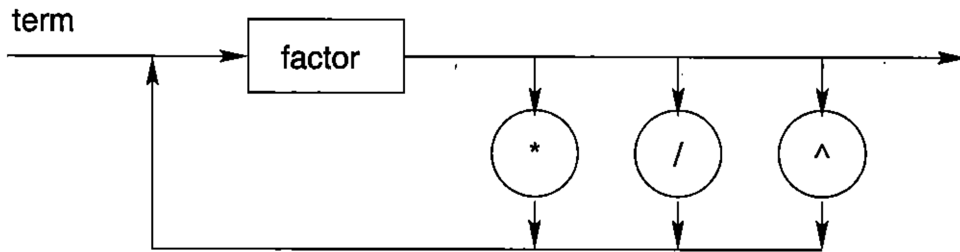

Figure 5.6. *Term*

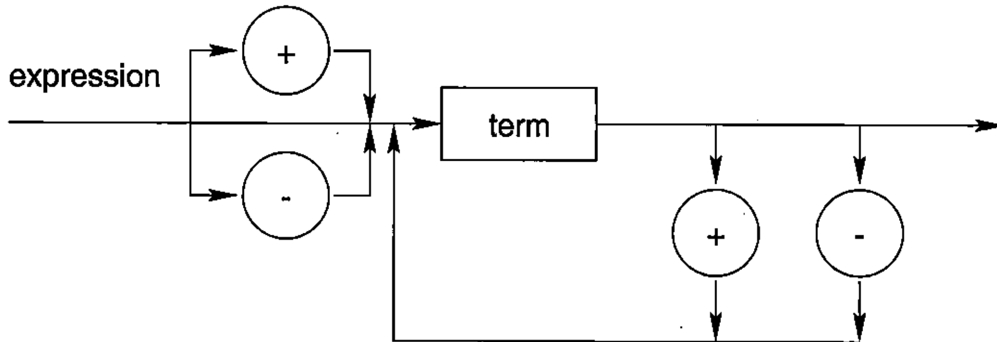

Figure 5.7. *Expression*

5.2 Obtaining code from syntax graphs

In the previous section the syntax for user input has been explained. We elected not to use any parser grammar such as Lex or Successor, but instead manually obtain code as described by Wirth [24]

Obtaining the code, `T(S)`, from a syntax graph `S` is straightforward and is achieved by the following rules:

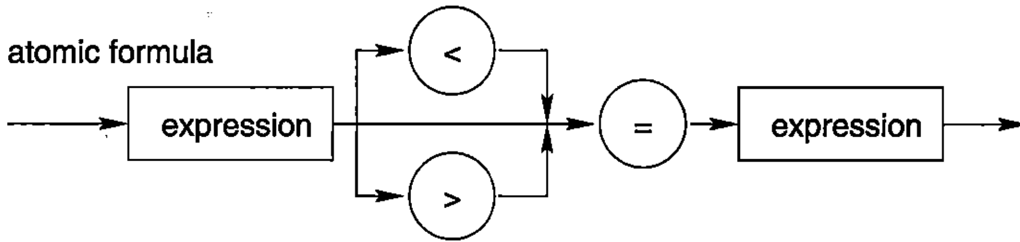

Figure 5.8. *Atomic Formula*

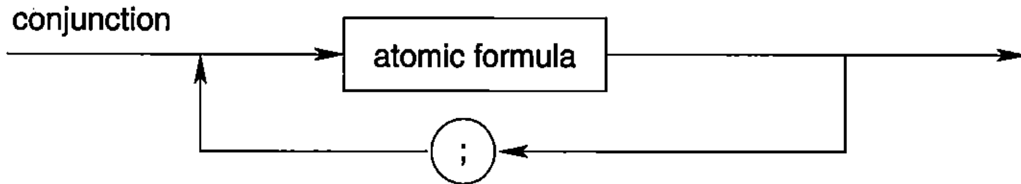

Figure 5.9. *Conjunction*

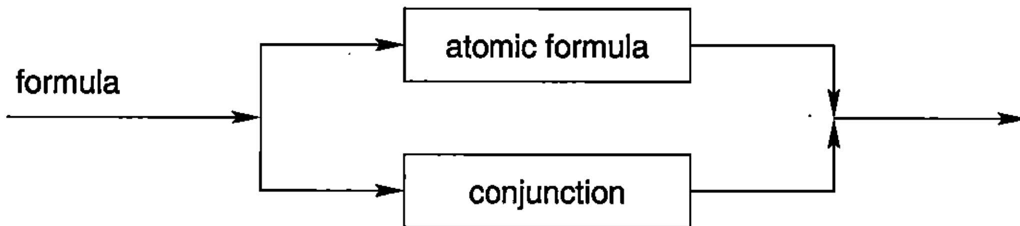

Figure 5.10. *Formula*

1. Translate each graph into a function declaration according to the subsequent rules 2 through 6.
2. A *sequence* of elements shown in Figure 5.11 is translated into the compound statement shown below.

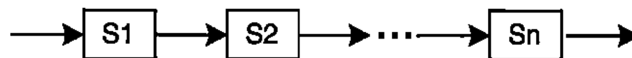

Figure 5.11. *Sequence of elements*

```
T(S1); T(S2); ...; T(Sn);
```

3. A *choice* of elements shown in Figure 5.12 is translated into the `switch` statement

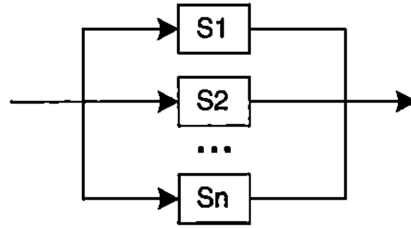

Figure 5.12. *Choice of elements*

```
switch (ch) {
 case A11: case A12:
   T(S1);
   break;
 case A21: case A22:
   T(S2);
   break;
   ......
 default:
   break;
}
```

where `A11, A12` denote the initial symbols in `S1`.

4. A loop of the form shown in Figure 5.13 is translated into the statement

```
while ((ch == A1) || (ch == A2)) {
       T(S);
}
```

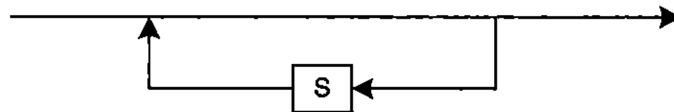

Figure 5.13. *Loop*

where `A1, A2` denote the initial symbols in `S`.

5. An element of the graph denoting another graph `S` shown in Figure 5.14 is translated into the function call statement `S;`.

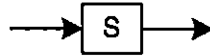

Figure 5.14. *Element of a graph denoting another graph*

6. An element of the graph denoting a terminal symbol x shown in Figure 5.15 is translated into the statement

```
if (ch == x) {
    getc(ch);
} else {
    error
}
```

Figure 5.15. *Element of a graph denoting a terminal symbol*

An ill-formed input expression is indicated by an `error` and the parsing is aborted. Due to the fact that no step in the process is revoked, the value of the input is not restored to its original value.
The C program shown in Figure 5.16 demonstrates the application of these rules by translating the graph shown in Figure 5.1. From the C program it is evident that the translation process is straightforward and the code that is obtained is easily verifiable with its corresponding syntax graph. In the long run, with the help of the syntax graphs and the translated code, the software properties such as maintainability, inspectability, modifiability and testability can be achieved very easily.

5.3 Expressions: design and implementation

If the input is well-formed, the parser module parses the input and returns an instance of a constraint system. The constraint system contains a set of primitive constraints that correspond to the input expressions. There exists a parser component, `COParser`, that returns this instance to the user as a pointer to an interface, `IICSystem`. The

```
void variable(char &ch, string &s) {
  if (!isLetter(ch)) return;
  ch = getNext(s);
  while (isLetter(ch) || isDigit(ch)) {
        ch = getNext(s);
  }
}
```

Figure 5.16. *Parsing program for a Variable*

functionality of the component is exposed through the interface `IParser`. The interface is implemented in the C++ class, `CImpIParser`. The component class definition of `COParser` and the interface definition of `IParser` is shown in Figure 5.17.
The method `run` in the parser component takes in the mathematical model as a string input, `pInExpr`, and returns a string, `pOutExpr`, with the pointer to the interface, `pIcsystem`, initialized. The returned string is empty if the input expression is well-formed (follows the syntax graph rules in Section 5.1). For ill-formed input, the returned string contains the input up to and including the illegal character and the interface pointer, `pIcsystem`, pointing to `NULL`.
`getVarsAsString()` returns a list of unknowns in the constraint system delimited by a semi-colon.
`getUnknown()`, given a string representation of an unknown, returns the instance of the unknown as a pointer to an `IReal` if it is found in the constraint system.

Simple mathematical expressions that correspond to the primitive constraints can be divided into two main categories: Expressions and Formulas. Expressions are divided into eight categories: signed expression, number, variable, add, subtract, multiply, divide and power. Formulas are divided into four categories: equal, less than or equal, greater than or equal and conjunction of formulas. The categories can directly be translated to their corresponding C++ classes. The main categories as abstract classes and the sub-categories as derived classes.

The expressions and formulas in a mathematical expression are identified and created by the parser module. The primitive constraints are created and added to the constraint system when the constructors for the expressions and formulas are called.

```
[uuid(32718C83-8324-4210-A8EE-92B62D939295),
helpstring("Component object class")] coclass COParser {
[default] interface IParser; };

[ object, uuid(F1BB73FF-9D1F-4B3F-9426-27EEC81389CD),
  dual,helpstring("Parser Interface"),
  pointer_default(unique) ]
interface IParser: IDispatch {
  HRESULT run([in]BSTR *pInExpr, [out]IICSystem **pIcsystem,
              [out, retval]BSTR *pOutExpr);
  HRESULT getVarsAsString([out, retval]BSTR *variables);
  HRESULT getUnknown([in]BSTR *str, [in,out] IReal **unknown);
};
```

Figure 5.17. *The `COParser` component class and the `IParser` interface*

Since the primitive constraint properties are exhibited in both expressions and formulas a class that encapsulates the behavior of adding/getting the constraint to/from the constraint system is required. An abstract class, `PConstraint`, is introduced for this purpose with the interface definition shown in Figure 5.18.
The abstract classes that encapsulate the behavior of an expression, `Exp`, and formula, `Formula`, derive from `PConstraint`. Figure 5.19 show the class interface definitions for `Exp` and `Formula`:

The class hierarchy diagram of the abstract classes and the classes implementing the sub-categories is shown in Figure 5.20.

```
class PConstraint {
protected:
  Constraint *c;
public:
  virtual Constraint *getConstraint() { return c; }
  virtual void eval(vector<Real*> &vReal) {
            if (c != NULL)
                c->shrink(vReal);
  }
  virtual bool addConstraint() = 0;
};
```

Figure 5.18. *Interface definition for PConstraint*

```
class Exp: public PConstraint {
public:
  virtual Real *getAuxVar() { return NULL; }
  virtual char *toString() = 0;
  virtual void print() { cout << toString(); }
};

class Formula: public PConstraint {
public:
  virtual IICSystem *getSystem() { return g_pIIcs; }
  virtual char *toString() = 0;
  virtual void print() { cout << toString(); }
};
```

Figure 5.19. *Interface definitions for Exp and Formula*

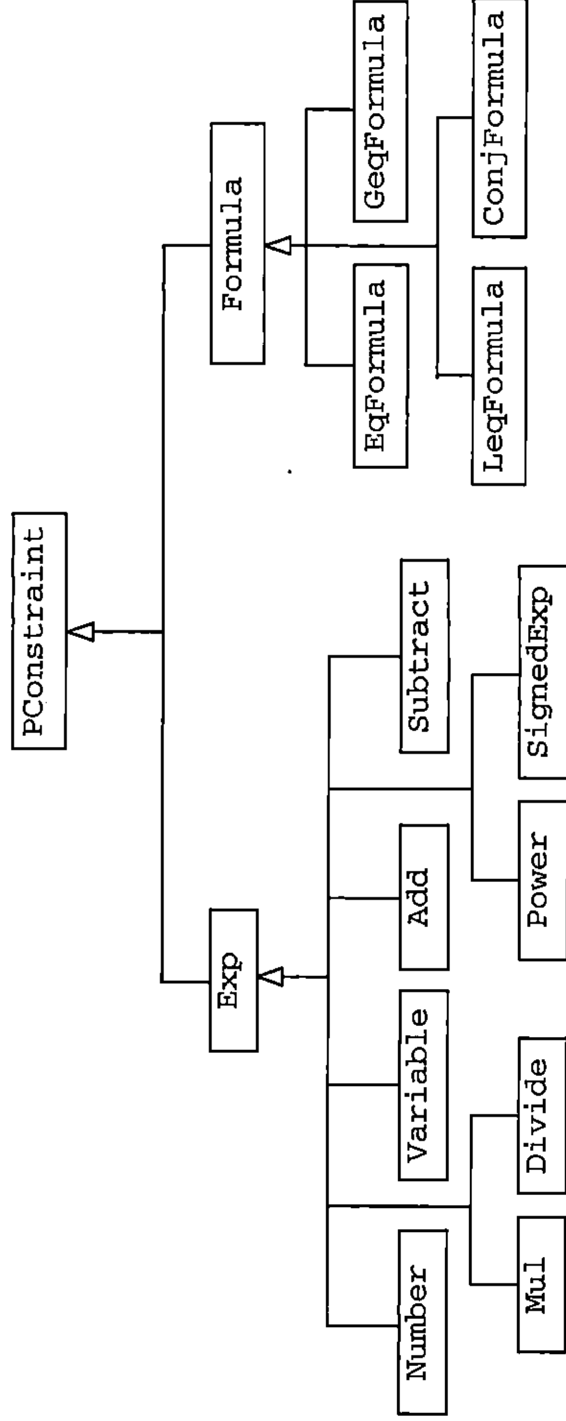

Figure 5.20. *The Expression class hierarchy*

Chapter 6

Graphical User Interface and System Usage

An interval constraint system removes as much as possible the inconsistent values from the domains. The system can be explored further by means of a graphical user interface that can manipulate the intervals. In this chapter we give details about the interface that allows the user to interactively explore a given constraint system.

6.1 The user tasks

An interval constraint system can be explored interactively in two ways: Elimination and Probing. In *Elimination* (section 6.1.1) the user tries to eliminate the uninteresting parts from the initial consistent system that resulted from propagation. The interval that has resulted from Elimination will be under the control of the GUI for further selection. In *Probing* the goal is to find and remove an inconsistent boundary interval by investigating (section 6.1.2) the outer parts of the intervals obtained from propagation.

6.1.1 Elimination

Generally in Elimination the subspace that is occupied by the few dozen user unknowns will be of interest. For interactive control and display only a few of these user unknowns are selected. Elimination is described as follows. The bound for a selected interval is moved inward with the help of a graphical control, a slider. The slider consists of a movable pointer. The pointer is used to change the values that the slider

represents. Each move on the slider can take place in either small or large steps. The change in value for one of the bounds causes an additional constraint to be added to the existing system and causes the propagation to be initiated. If the propagation succeeded the bounds for the displayed intervals will contract. These contractions are displayed to the user instantly by moving the contracted bounds' sliders inward. By performing many such operations the user can get a feel for the system. From the results that are displayed during these operations the user might notice that the system is more sensitive to some unknowns than to others.

When bounds are moved inward beyond a critical point, the system will become inconsistent. It is not useful to keep the system and the display in an inconsistent state. Therefore, in such a situation the last consistent state remains the state of the system and display. The graphical controls for the interval that caused the system to be inconsistent are frozen. It might happen that the bounds for other intervals can be moved further inward without making the system inconsistent. But it might also happen that none of the bounds can be moved inward. At this stage it can be said that the box that is defined by the projection of the intervals is the smallest box that can be achieved. On the other hand a smallest box does not mean that all the solutions are contained in the box. By moving the bound(s) inward the user is eliminating the uninteresting parts of the solution contained outside the bound(s) for the interval(s). By performing these operations repeatedly the user can analyze the subsolution space that is of specific interest. Another advantage of having such a feature is for underconstrained systems. The user can narrow down the solution space by adding constraints to the system guided by intuition or in an ad-hoc manner.

A constraint system changes only when new constraints are added. This addition is irreversible and suggests that the system be organized into sessions. A control that allows the user to indicate the start of a new session is provided. Starting a new session resets the constraint system to the same initial constraint system and state. A reason for starting a new session might be due to any of the following reasons:-

- the user is interested in investigating a different region in the solution space
- the user might have regretted a move that was made to a bound
- the system might have ended suddenly with all the bounds in their innermost position

6.1.2 Probing

In Probing the bounds are moved inwards in such a way that the set of solutions, if any, is preserved. Note that propagation has the property of returning intervals containing the set of all solutions. Even from the set of intervals presented the system can become inconsistent when an interval that is left from propagation is bound to a certain boundary interval. The boundary interval can be removed from the interval presented by the initial propagation. This will not affect the set of solutions that are already present. The process of finding this boundary interval for a selected bound of an interval is called Probing.
Elimination must be done interactively. The initial propagation takes care of Probing automatically. But Probing through user interaction can be instructive and is worth considering.

Probing offers two options for user interaction: react to the first slider movement and detect the release of the mouse button after the slider has moved. Let's assume that the slider of the upper bound for the unknown, x, was moved from x_1 to x_2. The interval of the unknown is then restricted to the boundary interval, $[x_2, x_1]$. This is applied by adding a new constraint, $x_1 \geq x \geq x_2$ to the existing constraint system. The propagation is initiated for the new system. If the propagation results in failure the slider follows the mouse to x_2. Otherwise the slider stays at x_1. The second option is available for slower processors where the slider may not be able to follow the mouse. This option in Probing can be implemented differently for slower processors. Propagation is not started until after the boundary interval is defined. In this implementation no change is made to the constraint system as long as the bound's slider is being dragged with the mouse. The boundary interval is defined by the release of the mouse button. The propagation is then initiated.

While the mouse is moved and the system propagated the GUI can indicate the state of the system. Changing the color of the GUI can indicate the state for example. Propagation may fail for a certain boundary interval. In that case propagation is re-initiated by considering the outer half of the defined boundary interval. This process is repeated until the boundary interval leads the system to inconsistency or until a smallest boundary interval has been found that is consistent. The second outcome indicates that the selected bound is already at the innermost position. This outcome

is an important part of interactive exploration for boundary intervals. The graphical user interface can be colored suitably for the outcome or can be said that the boundary is "hard". Initially the boundary interval is said to be "soft". Probing helps the user to harden and determine the boundaries of interest.

6.2 Differences between Elimination and Probing

In Elimination, the selected bound for an interval is always moved inward. This move may make the other bounds move inward too. Unlike Elimination, the inward movement in Probing does not affect other bounds.
Under Elimination, since the user is interested in finding solutions, shrinking the bounds further is of interest. The state of the displayed constraint system after many moves is a result of constraint propagation. Shrinking the bounds further to find solutions can be effected by Probing.

6.3 Graphical controls and their behavior

The GUI, Interval Constraint System Explorer or ICS Explorer, is the interface between the user and the constraint system component which is responsible for creating an interval constraint system and propagating the constraint system. The graphical controls in ICS Explorer can be grouped into 4 main categories. Figure 6.1 shows the ICS Explorer with the groups framed and labeled.
Input: A constraint system can be given to the ICS Explorer from the text box provided (see frame 1.a in Figure 6.1). Selected test cases are available from within the application through a drop-down list. By default this list is hidden. When the check box corresponding to the text label `Auto` is checked, frame 1.a is replaced by frame 1.b (see Figure 6.2) revealing the drop-down list. When a test case is selected the propagation for the constraint system is initiated. Clicking the button labeled `Propagate` initiates the propagation for the constraint system under consideration. The button labeled `Clear` deletes the constraint system under consideration and resets all the graphical controls to their default status and positions.
Mode: Two radio buttons(frame 2 of Figure 6.1) are provided for the user to select

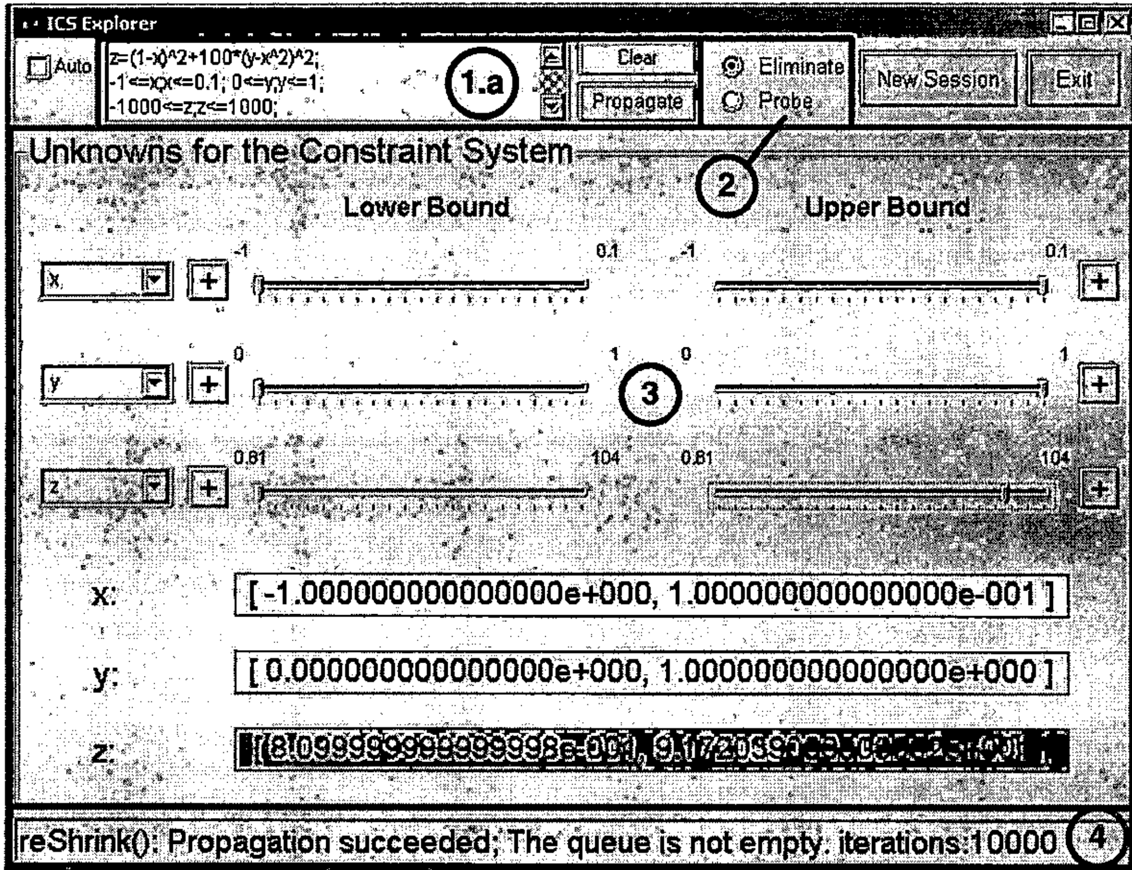

Figure 6.1. *Interval Constraint System Explorer*

Figure 6.2. *Automatic input*

between the two modes, Eliminate and Probe. The ICS Explorer defaults to Eliminate mode.

Reset and Exit A button labeled `New Session` is provided to start a new session. The `Exit` button closes the ICS Explorer application.

Explore: The graphical controls in frame 3 are enabled if the propagation was successful. The drop down lists in this frame contain a list of all the user variables in the

current constraint system. If propagation fails the frame and its components remain disabled with an appropriate message displayed in the colored text boxes contained in the frame. The interval for a selected user variable is displayed by the name of the variable followed by the interval in the colored text box.
To allow the user to modify the bounds the slider widget is used. Ideally, each interval should be represented by a slider with two movable pointers. Each pointer represents the bound of an interval. The space between the sliders represents the interval itself. The slider widget that comes with the standard Visual Basic kit has only one pointer. Therefore, to represent the movable bounds we have a pair of sliders (see frame 3 in Figure 6.1) for each interval. As a result we lose the natural representation that an interval would have by a slider with two pointers. But such behavior can be emulated as follows. At any time in a session the left and right sliders both represent an interval. The interval on the left slider can be represented by the space in the slider to the right of the pointer. In the right slider the space to the left of the pointer represents the interval. Since the lengths of the spaces for these sliders may not be the same it is possible to present the same interval in different scales. While a slider is being moved, the current value of the bound is displayed in a window that pops up just above the slider's pointer. When the slider is not moving the current value for a bound can be displayed in a pop-up window by pointing the mouse on the slider.

A very wide interval can be represented on the very small slider distance. In such situations it is difficult and sometimes impossible for a user to make changes to the bounds at different magnitudes. This difficulty arises especially when a very small change is required. Moreover the resolution of the screen and the smoothness of the hand-mouse control are very limited. These limitations can be overcome by providing zooming for each of the bounds. The zooming operation is described as follows.

With every interval there is associated a button labeled + (see frame 3) that, when activated, causes zooming to happen. This means that the bound's slider that is associated with the zooming button moves to its outermost position. This action does not change the interval represented by the slider but enlarges the scale of the interval. This creates a zooming effect on the slider. By zooming the bounds multiple times the width of the interval can be controlled at even higher resolutions. The above setup is the same during both: Elimination and Probe.

Status: Frame 4 in Figure 6.1 contains a text box. This text box is used to display any general, warning or error messages that might have been generated by the ICS Explorer application.

6.4 Component activation and usage

Visual Basic maintains a database of all the type libraries and component objects installed on a computer. To access an object a reference to the object or the type library has to be set in the Visual Basic project. This reference is set by selecting the name of the type library available from the menu option, `References...`, found under the `Project` main menu option.

After a reference to the interval constraint system's object library has been set a parser component object can be created as shown below:

```
'A reference to the interface, IParser
Dim iParser As IParser
'create the Parser component
Set iParser = New COParser
If (iParser Is Nothing) Then
   MsgBox "Could not get reference to the Parser."
End If
```

To access the COM components present in the interval constraint system type library from a C++ application the following header files should be included:-

- `comdef.h`: contains COM definitions
- `ics_i.c`: contains GUID definitions of IIDs and CLSIDs
- `ics.h`: contains definitions of interfaces

COM components from the C++ application can be accessed only after the COM library has been initialized. The COM library function call, `CoInitialize()`, with a `NULL` argument initializes the COM library. The function call `CoUninitialize()` closes the COM library and unloads any DLLs loaded by the application.

Shown below is the equivalent source code in C++ for creating a parser component object:-

```
//A reference to the interface, IParser
IParser *iParser = NULL;
//create the Parser component
HRESULT hr = CoCreateInstance(CLSID_COParser, NULL, CLSCTX_ALL,
                              IID_IParser,
                              reinterpret_cast<void **>(&iParser));
if (FAILED(hr)) {
    cout << "Could not get reference to the Parser.\n";
}
```

For a given input the parser component's **run** method returns an instance of a constraint system. The Visual Basic code below shows the method, **run**, returning an instance of the constraint system component.

```
'A reference to the interface, IICSystem
Dim iConstraintSystem As IICSystem
Dim inputString As String
res = iParser.run(inputString, iConstraintSystem)
If (iConstraintSystem Is Nothing) Then
    Status = "Syntax error: " & res
End If
```

The equivalent source in C++ is as follows:-

```
//A reference to the interface, IICSystem
IICSystem *iConstraintSystem = NULL;
BSTR inputString, out;
iParser->run(&inputString, &iConstraintSystem, &out);
if (iConstraintSystem == NULL) {
    CString res = (LPCWSTR)out;
    cout << "Syntax error: " << (LPCSTR)res << endl;
}
```

Chapter 7

Conclusion and Future work

Interval methods have proved to be valuable in numerical computations because of the certainty in the results that they provide. This has been proved by an application of the package, NUMERICA, finding all solutions to nonlinear systems of equations. NUMERICA faces some obstacles. To overcome these obstacles we developed an interval constraint system with a suite of components and made the source code open.

7.1 Main features of this thesis research

Non-linear systems of equations can be solved by using interval constraint systems. It is guaranteed that if solutions exist, they are present in the resulting intervals displayed by the system. We also prove that the solution that is provided is sound. This can be verified with the source code of the system which is kept open.

The key concepts in the COM technology were introduced and explained with respect to the thesis. Wherever possible the source code for interval arithmetic and primitive constraints from [18] was reused. COM components for performing interval arithmetic operations, translating a mathematical model to a set of primitive constraints and solving a constraint system were designed in this thesis. We specified the constraint contraction operator and the propagation algorithm as a part of the interval constraint system.

The COM components available from the interval constraint system type library were implemented from scratch. References to the interfaces these COM components exposed could be created from an application developed using C++. References to the COM interfaces could not be obtained from applications developed using Visual

Basic. This problem was overcome by re-designing the components using the ATL COM Wizard, a feature that comes with the Visual C++ software development kit.

Constraint systems are solved by translating each constraint into a set of primitive constraints and creating a network between the primitive constraints and the unknowns that they share. This task is tedious and highly error prone. This problem was solved by automating the translation process. A parser component was designed and implemented for this purpose.

The graphical user interface application developed in Visual Basic helps to solve and analyze a constraint system interactively. It also stands as a proof that our components are reusable and interoperable.

7.2 Contributions

In this thesis we have designed and developed the COM component objects `COReal`, `COICSystem` and `COParser` which expose the immutable interfaces `IReal`, `IICSystem` and `IParser` respectively. The component objects are packaged in the `ICSystemLib` type library and are available to the user through this type library.

A consistent system that may result from propagation can be explored further by means of a graphical interface that can manipulate the intervals. We have developed a graphical user interface with this capability. An interval for a selected unknown can be modified by changing the slider control that corresponds to the unknown. The granularity of the values represented by a slider can be modified by using the zooming control.

The Visual Basic interface communicates with the component objects through interface pointers. New functionality to component objects can be added only by introducing new interfaces and exposing them. Introducing new interfaces does not affect the graphical user interface. In other words the user interface application need not be recompiled. This holds for other applications that use the components too. This is due to the fact that the *vtbl* layout of the functions that already exists in the components remain unchanged. This is a feature in COM.

7.3 Limitations of ICS Explorer

The ICS Explorer tool can be used to interactively explore the solution space for a constraint satisfaction problem consisting of about a hundred unknowns. During an exploration the ICS Explorer has a limitation on the number of unknowns that can be displayed simultaneously. But manually it is not possible to observe and study the behavior of more than a few unknowns simultaneously. Hence, the graphical components that display the unknowns in the ICS Explorer are sufficient to explore any constraint satisfaction problem interactively.

7.4 Future work

Fractional powers The primitive constraint $x^n = y$, has been implemented only for the integer values of n. The primitive constraint can be re-implemented to support fractional values for n. The new implementation will effect the following functions and methods:-

- `PowerN::PowerN(Real *x, Real *y, int n)`
- `rootIntsect(const Real &y, const int &n, const Real &x)`
- `powerIntsect(const Real &x, const int &n, const Real &y)`
- `ConstraintSystem::addConstraintPowerN(Real *x,Real *y,int n)`
- `Power::addConstraint()`

Note that the expression parser already supports fractional powers in expressions.

Transcendental functions Transcendental functions are used in many non-linear problems. Support to the transcendental functions could be added to the existing constraint system. Each function can be considered as a primitive relation. For example, the expression $y = cos(sin(x))$ would be translated into $y1 = sin(x)$, and $y = cos(y1)$. The variables used by the functions are real intervals. The operations on the reals have to be defined depending on the class of the interval (see Table 3.1) used in the transcendental functions. The transcendental operations on intervals can be implemented in the class `Real` (see Appendix A). An appropriate expression class that creates the primitive constraint and adds the constraint to the constraint system

can then be introduced (see class hierarchy diagram in Figure 5.20). A new function that parses a transcendental function would be added to the parser module. Other functions in the parser module would be modified accordingly. In the lowest level, the transcendental functions are applied to floating-point numbers that are used to calculate the lower and upper bounds of the resulting interval.

Translation of complex mathematical functions Extend the parser module to expand and translate complex mathematical expressions involving $\sum$ and $\prod$.

Use of force feedback The elimination and Probing modes that were described in sections 6.1.1 and 6.1.2 use the mouse for user interaction. Similar interactions can be performed with the help of a joystick. With a joystick, the resistance to the force applied by the user can be varied. This is an added advantage. The resistance to force can be made proportional to the computational effort caused by the slider movement controlled by the user. Giving the feedback in this way the user can find that some boundaries of the interval are softer compared to the others. The intervals displayed as a result of propagation can contain soft boundaries. The interpretation of soft boundaries is applied differently for Elimination and for Probing.

Probing Though not important, probing through user interaction is instructive and is worth implementing. The implementation only affects the Visual Basic source code and application. The definition of probing and the behavioral details of the GUI under the probing mode were given in section 6.1.2.

Glossary

automation A mechanism used for binding. It enables applications to communicate with components, may be developed in diferent programming languages. *page 14*

automation controller A client for a component object featured with automation. *page 14*

automation server A component server featured with automation. *page 14*

binding A verification process in COM for the validity of each function call before actually making the call. *page 14*

class In Object-Oriented programming is a unit of software that encapsulates the state and behavior of an abstraction. *page 7*

class library A library created from an Object-Oriented programming language like C++. *page 6*

compiler A program that converts high-level source code to an intermediate object code. The object code cannot be executed by itself and may span across several files. *page 4*

component In component software context is an independent, reusable unit of software in binary form. *page 6*

γ The constraint contraction operator. *page 30*

ϕ An empty interval that has no real values. *page 22*

EXE Referred to an executable on Microsoft windows environment. Usually an EXE has a .exe extension. *page 10*

executable A program that can be executed directly on a micro-processor. *page 5*

function overloading A feature in Object-Oriented programming where methods have the same name and return type but with different parameter lists. *page 5*

GUID *Globally Unique IDentifier* It is a 128-bit integer that is unique at any given

point of time and on any computer. An application called Guidgen generates these numbers. *page 16*

library A library contains object code for operations that are most commonly required by other programs. A compiler is required to generate a library. Libraries may come with a compiler, can be installed on a machine separately or can also be generated from the source code using a compiler. *page 4*

linker A program that binds the object code and links with any other library functions that are called to form a *single executable object file* or simply a *program. page 4*

loader It is an operating system utility that is responsible for copying a program from a storage device to the main memory and also allocating the memory required for the program. *page 5*

M The set of real intervals containing at least one positive and one negative real. *page 25*

machine box Is a cartesian product of machine intervals. *page 23*

machine interval Any set of reals bounded by two machine numbers, including the set ϕ and $\Re$. *page 23*

machine numbers A finite set of reals, floating-point numbers representable on a computer. *page 22*

marshaling The process of formatting and bundling the data to be able to transferred across processes. *page 11*

N The set of real intervals containing at least one negative, but no positive number. *page 26*

name mangling The process of translating the overloaded function names into distinct symbols in order to identify the methods correctly by the linker. The translation process is performed by a compiler. *page 5*

NaN Not a Number, the IEEE arithmetic representation for an undefined operation. *page 22*

object code The machine code that a processor can process or execute one instruction at a time. *page 4*

outward rounding Rounding the lower bound towards $-\infty$ and the upper bound

towards $+\infty$. *page 22*

P The set of real intervals containing at least one positive, but no negative number. *page 26*

procedural library A library created from a procedural language like C. *page 6*

process space The memory occupied by a process. *page 10*

proxy The code that exists in a client and is responsible for marshaling, unmarshaling and sending the data to and from the server. *page 11*

x^- The greatest floating-point number not greater than the real number x. *page 22*

x^+ The smallest floating-point number not smaller than the real number x. *page 22*

SCM *Service Control Manager* A service offered by COM to locate component servers and components that reside on the local machine as well as on remote machines. *page 12*

spreadsheet A computer program used mainly for accounting purposes. *page 7*

stub The code that exists in a server and is responsible for marshaling, unmarshaling and sending the data to and from the client *page 11*

type library Groups a collection of data types and component classes that an automation server exports into a namespace. *page 14*

unmarshaling The process of unbundling and unformatting the data to be able to read in a different process. *page 11*

uuid *unique universal identity* The physical name given to an interface in the Component Object Model. *page 16*

vptrs *virtual function pointers* Contain the memory addresses of virtual functions. *page 9*

vtbl *virtual table* A table of vptrs. *page 9*

wrapper-style library A software library that encloses another library. *page 5*

Z The set of non-empty real intervals containing neither a positive nor a negative number. *page 26*

Bibliography

[1] Ramon E. Moore, *Interval Analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1966.

[2] E. Davis, "Constraint Propagation with Interval Labels," *Artificial Intelligence*, vol. 32, pp. 281–331, 1987.

[3] J.G. Cleary, "LOGICAL ARITHMETIC," *Future Computing Systems*, vol. 2(2), pp. 125–149, 1987.

[4] Pascal van Hentenryck, Laurent Michel and Yves Deville, *Numerica: A Modeling Language for Global Optimization*, MIT Press, 1997.

[5] H. Ratschek and J.Rokne, "Experiments using interval analysis for solving a circuit design problem," *Journal of Global Optimization*, vol. 13(1), pp. 410–423, 1998.

[6] Jean-Francois Puget and Pascal Van Hentenryck, "A Constraint Satisfaction Approach to a Circuit Design Problem," 1998.

[7] Don Box, *Essential COM*, Addison-Wesley, 3rd edition, 1998.

[8] Szyperski C., *Component Software: Beyond Object-Oriented Programming*, ACM press and Addison-Wesley, New York, 1st edition, 1999.

[9] *The website address of The Open Group: http://www.opengroup.org/.*

[10] *The website address of The Object Management Group: http://www.omg.org.*

[11] *The website address of Java 2 Platform, Enterprise Edition: http://java.sun.com/j2ee/.*

[12] *http://support.microsoft.com/support/kb/articles/Q189/1/34.asp.*

[13] *The specifications for COM are available for download from the site http://www.microsoft.com/com/resources/comdocs.asp.*

[14] *http://www.microsoft.com/com/default.asp.*

[15] *Object Management Group. The Common Object Request Broker Architecture and Specification. OMG, Inc., Feb. 1998. Version 2.2 (Revision 98-02-01). A most recent version can be downloaded from http://www.omg.org/technology/documents/specifications.htm.*

[16] Dan Harkey Robert Orfali, *Client/Server Programming with Java and CORBA*, John Wiley & Sons, 2nd edition, 1998.

[17] T. Hickey, Q. Ju, and M.H. van Emden, "Interval arithmetic: from principles to implementation," *Journal of the ACM*, 2001.

[18] Huan Wu, "Defining and Implementing a Unified Framework for Interval Constraints and Interval Arithmetic," M.S. thesis, University of Victoria, May 1999.

[19] A. IEEE, "IEEE standard for Binary Floating-Point Arithmetic," 1985.

[20] Maarten H. van Emden Timothy J. Hickey and Huan Wu, "A unified framework for interval constraints and interval arithmetic," *In Principles and Practice of Constraint Programming – CP98 Michael Maher and Jean-Francois Puget(eds.), Springer-Verlag, volume 1520 of Lecture Notes in Computer Science*, pp. 250–264, 1998.

[21] D. Ratz, "On extended interval arithmetic and inclusion isotonicity," Institut für Angewandte Mathematik, Universität Karlsruhe, 1996.

[22] Frédéric Benhamou and William J. Older, "Applying interval arithmetic to real, integer, and boolean constraints," *The Journal of Logic Programming*, vol. 32, no. 1, pp. 1–24, 1997.

[23] Maarten van Emden, "DATA-DIRECTED DESIGN (DDD)," Software Engineering 365 course, Summer 2002, University of Victoria.

[24] Niklaus Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

Appendix A

```
1  /*
2    Implementation of the Real class
3  */
4  #include "stdafx.h"
5  #include "Ics.h"
6  #include "real.h"

7  #ifndef __INTERVAL_H_
8  #include "interval.h"
9  #endif

10 #ifndef __FLPT_H_
11 #include "flpt.h"
12 #endif


13 //Real class methods
14 Real::Real() {
15   initLB = lb = NEG_INF;
16   initUB = ub = POS_INF;
17 }

18 Real::Real(const Real &x) {
19   initLB = this->lb = x.lb;
20   initUB = this->ub = x.ub;
21 }

22 Real::Real(double lu) {
23   if (lu == POS_INF || lu == NEG_INF) {
24     exceptionMode em = InvalidBound;
25     throw((except)(em));
26   }
27   if (lu == 0) {
28     this->lb = POS_ZERO;
29     this->ub = NEG_ZERO;
30   }
31   else {
32     Flpt::roundDown();
33     this->lb = lu;
```

```
34      Flpt::roundUp();
35      this->ub = lu;
36    }
37    initLB = lb;
38    initUB = ub;
39  }

40  Real::Real(double lb, double ub) {
41    if (lb == POS_INF || ub == NEG_INF) {
42      exceptionMode em = InvalidBound;
43      throw((except)(em));
44    }
45    if (lb == 0 || lb == NEG_ZERO)
46      lb = POS_ZERO;
47    if (ub == 0 || ub == POS_ZERO)
48      ub = NEG_ZERO;

49    initLB = this->lb = lb;
50    initUB = this->ub = ub;
51  }

52  Real::~Real() { }

53  void Real::resetBounds() {
54    lb = initLB;
55    ub = initUB;
56  }

57  Real Real::operator+(const Real &y) {
58    double a, b, c, d, ac, bd;
59    a = lb;      b = ub;
60    c = y.getLB();      d = y.getUB();
61    Flpt::roundDown();
62    ac = a+c;
63    Flpt::roundUp();
64    bd = b+d;
65    return Real(ac, bd);
66  }

67  Real Real::operator-(const Real &y) {
68    double a, b, c, d, ad, bc;
69    a = lb;      b = ub;
70    c = y.getLB();      d = y.getUB();
71    Flpt::roundDown();
72    ad = a-d;
73    Flpt::roundUp();
74    bc = b-c;
75    return Real(ad, bc);
76  }
```

```
 77 Real Real::operator*(const Real &y) {
 78   double a, b, c, d, ac, bd, bc, ad;
 79   a = lb;      b = ub;       // x = [a, b]
 80   c = y.getLB();      d = y.getUB(); // y = [c, d]
 81   /* handle the cases: a=b=0, or c=d=0 separately to avoid 0*INFINITY */
 82   if ((a==0.0 && b==0.0) || (c==0.0 && d==0.0))
 83     return Real(0.0, 0.0);
 84   if (a>=0.0) /* 0<=a, 0<b */ {
 85     if (c>=0.0) /* 0<=c, 0<d */ {
 86       Flpt::roundDown();
 87       ac = a*c;
 88       Flpt::roundUp();
 89       bd = b*d;
 90       return Real(ac,bd);
 91     }
 92     if (d>0.0) /* c<0<d */ {
 93       Flpt::roundDown();
 94       bc = b*c;
 95       Flpt::roundUp();
 96       bd = b*d;
 97       return Real(bc,bd);
 98     }
 99     /* c<0, d<=0 */
100     Flpt::roundDown();
101     bc = b*c;
102     Flpt::roundUp();
103     ad = a*d;
104     return Real(bc,ad);
105   };
106   if (b>0.0) /* a<0<b */ {
107     if (c>=0.0) /* 0<=c, 0<d */ {
108       Flpt::roundDown();
109       ad = a*d;
110       Flpt::roundUp();
111       bd = b*d;
112       return Real(ad,bd);
113     }
114     if (d>0.0) /* c<0<d */ {
115       Flpt::roundDown();
116       ad = minFlpt(a*d,b*c);
117       Flpt::roundUp();
118       ac = maxFlpt(a*c,b*d);
119       return Real(ad,ac);
120     }
121     /* c<0, d<=0 */
122     Flpt::roundDown();
123     bc = b*c;
124     Flpt::roundUp();
```

```
125       ac = a*c;
126       return Real(bc,ac);
127     };
128     /* a<0, b<=0 */
129     if (c>=0.0) /* 0<=c, 0<d */ {
130       Flpt::roundDown();
131       ad = a*d;
132       Flpt::roundUp();
133       bc = b*c;
134       return Real(ad,bc);
135     }
136     if (d>0.0) /* c<0<d */ {
137       Flpt::roundDown();
138       ad = a*d;
139       Flpt::roundUp();
140       ac = a*c;
141       return Real(ad,ac);
142     }
143     /* c<0, d<=0 */
144     Flpt::roundDown();
145     bd = b*d;
146     Flpt::roundUp();
147     ac = a*c;
148     return Real(bd,ac);
149   } //operator*


150   /* Returns a pointer to an instance of one of the subclasses of class interval.
151      Return by value will invoke the copy constructor of class interval by taking
152      different subclass instances. Then it will lose the whole purpose of polymorphism.
153   */
154   // res = (z/x) & y | (z/y) & x;
155   Real Real::divIntersect(const Real &y, const Real &x) {
156     double a, b, c, d;
157     a = lb;     b = ub;
158     c = y.getLB();     d = y.getUB();

159     // The inside_out cases, handles [-,-]/[-,+] and [+,+]/[-,+].
160     // generates an interval complement:
161     double s, t;
162     if (((c < 0.0) && (d > 0.0)) &&
163         ((b < 0.0) || (a > 0.0))) {    // [+,+]/[-,+] or [-,-]/[-,+]
164       if (c == NEG_INF && d == POS_INF) {  // both sides open at "0)(0"
165         return bothOutIntsect(0.0, 0.0, x);
166       }
167       if (b < 0.0) {
168         Flpt::roundUp();
169         s = b/d;
170         Flpt::roundDown();
```

```
171        t = b/c;
172        if (c == NEG_INF) {  // right side open at "s] (0"
173          return rightOutIntsect(s, t, x);
174        }
175        if (d == POS_INF) { // left side open at "0) [t"
176          return leftOutIntsect(s, t, x);
177        }
178        else {  // no side open
179          return noOutIntsect(s, t, x);
180        }
181      }
182      if (a > 0.0) {
183        Flpt::roundUp();
184        s = a/c;
185        Flpt::roundDown();
186        t = a/d;
187        if (c == NEG_INF) { // left side open at "0) [t"
188          return leftOutIntsect(s, t, x);
189        }
190        if (d == POS_INF) {  // right side open at "s] (0"
191          return rightOutIntsect(s, t, x);
192        }
193        else { // no side open
194          return noOutIntsect(s, t, x);
195        }
196      }
197    }

198    // The inside_in cases:
199    // Handle all the cases other than [-,-]/[-,+] and [+,+]/[-,+].
200    double ad, bc, ac, bd;
201    if ((a<=0.0 && b>=0.0) && (c<=0.0 && d>=0.0)) { // zero is in [a,b], [c,d]
202      return noInIntsect(NEG_INF, POS_INF, x);
203    }
204    if ((c==0.0 && d==0.0) && (a>0.0 || b<0.0)) { // zero is not in [a,b]
205      return noInIntsect(1.0, -1.0, x);
206    }
207    if ( (a==0.0 && b==0.0) && (c>0.0 || d<0.0) ) {  // zero is not in [c,d]
208      return noInIntsect(POS_ZERO, NEG_ZERO, x);
209    }
210    // Normal cases:
211    if (a>=0.0) {  // 0<=a, 0<b:
212      if (c>=0.0) {  // 0<=c, 0<d:
213        Flpt::roundDown();
214        ad = a/d;
215        Flpt::roundUp();
216        bc = b/c;
217        if (a>0.0 && d==POS_INF) {// excluded 0 only at lower bound
218          return leftInIntsect(ad, bc, x);
```

```
219         }
220         else {
221           return noInIntsect(ad, bc, x);
222         }
223       }
224       if (d>0.0) {    // c<0<d:
225         return noInIntsect(NEG_INF, POS_INF, x);
226       }
227       else { // c<0, d<=0:
228         Flpt::roundDown();
229         bd = b/d;
230         Flpt::roundUp();
231         ac = a/c;
232         if (a>0.0 && c==NEG_INF) { // excluded 0 only at upper bound
233           return rightInIntsect(bd, ac, x);
234         }
235         else {
236           return noInIntsect(bd, ac, x);
237         }
238       }
239     }
240     if (b>0.0) {  // a<0<b:
241       if (c>=0.0) {  // 0<=c, 0<d:
242         Flpt::roundDown();
243         ac = a/c;
244         Flpt::roundUp();
245         bc = b/c;
246         return noInIntsect(ac, bc, x);
247       }
248       if (d>0.0) {  // c<0<d:
249         return noInIntsect(NEG_INF, POS_INF, x);
250       }
251       else { // c<0, d<=0:
252         Flpt::roundDown();
253         bd = b/d;
254         Flpt::roundUp();
255         ad = a/d;
256         return noInIntsect(bd, ad, x);
257       }
258     }
259     else { // a<0, b<=0:
260       if (c>=0.0) {  // 0<=c, 0<d:
261         Flpt::roundDown();
262         ac = a/c;
263         Flpt::roundUp();
264         bd = b/d;
265         if (b<0.0 && d==POS_INF) { // excluded 0 only at upper bound
266           return rightInIntsect(ac, bd, x);
267         }
```

```
268       else {
269         return noInIntsect(ac, bd, x);
270       }
271     }
272     if (d>0.0) {  // c<0<d:
273       return noInIntsect(NEG_INF, POS_INF, x);
274     }
275     else { // c<0, d<=0:
276       Flpt::roundDown();
277       bc = b/c;
278       Flpt::roundUp();
279       ad = a/d;
280       if (b<0.0 && c==NEG_INF) { // excluded 0 only at lower bound
281         return leftInIntsect(bc, ad, x);
282       }
283       else {
284         return noInIntsect(bc, ad, x);
285       }
286     }
287   }
288 }// divIntersect


289 Real Real::operator&(const Real &y) {
290   double a, b, c, d, ac, bd;
291   a = lb; b = ub;
292   c = y.getLB(); d = y.getUB();

293   Flpt::roundDown();
294   ac = maxFlpt(a,c);
295   Flpt::roundUp();
296   bd = minFlpt(b,d);
297   return Real(ac, bd);
298 }

299 void Real::operator=(const Real &y) {
300   lb = y.lb;
301   ub = y.ub;
302 }

303 bool Real::operator==(const Real &y) {
304   return (lb == y.lb && ub == y.ub);
305 }

306 bool Real::operator!=(const Real &y) {
307   return (lb != y.lb || ub != y.ub);
308 }

309 /**
```

```
310     returns true if the upper bound is less than the lower bound
311   */
312   bool Real::empty() {
313     //if ((ub - lb) < NEG_INF)
314     if (ub < lb)
315       return true;
316     return false;
317   }

318   /*
319     Implementation of the IReal interface
320   */
321   STDMETHODIMP CImpIReal::InterfaceSupportsErrorInfo(REFIID riid) {
322     static const IID* arr[] =
323       {
324         &IID_IReal
325       };
326     for (int i=0; i < sizeof(arr) / sizeof(arr[0]); i++)
327       {
328         if (InlineIsEqualGUID(*arr[i],riid))
329           return S_OK;
330       }
331     return S_FALSE;.
332   }

333   STDMETHODIMP CImpIReal::init() {
334     AFX_MANAGE_STATE(AfxGetStaticModuleState())
335
336     initLB = lb = NEG_INF;
337     initUB = ub = POS_INF;

338     return S_OK;
339   }

340   STDMETHODIMP CImpIReal::initr(IReal *real) {
341     AFX_MANAGE_STATE(AfxGetStaticModuleState())

342       CImpIReal *x;
343     x = reinterpret_cast<CImpIReal*>(real);
344     initLB = lb = x->lb;
345     initUB = ub = x->ub;

346     return S_OK;
347   }

348   STDMETHODIMP CImpIReal::initd(double lu) {
349     AFX_MANAGE_STATE(AfxGetStaticModuleState())

350       if (lu == POS_INF || lu == NEG_INF) {
```

```
351        return AtlReportError(CLSID_COReal, _T("InvalidBound"), IID_IReal, REAL_E_INVALIDBOUND);
352      }

353    if (lu == 0) {
354      this->lb = POS_ZERO;
355      this->ub = NEG_ZERO;
356    }
357    else {
358      Flpt::roundDown();
359      this->lb = lu;
360      Flpt::roundUp();
361      this->ub = lu;
362    }
363    initLB = initUB = lu;
364    return S_OK;
365  }

366  STDMETHODIMP CImpIReal::initdd(double lb, double ub) {
367    AFX_MANAGE_STATE(AfxGetStaticModuleState())
368
369      if (lb == POS_INF || ub == NEG_INF) {
370        return AtlReportError(CLSID_COReal, _T("InvalidBound"), IID_IReal, REAL_E_INVALIDBOUND);
371      }

372    if (lb == 0 || lb == NEG_ZERO)
373      lb = POS_ZERO;
374    if (ub == 0 || ub == POS_ZERO)
375      ub = NEG_ZERO;

376    initLB = this->lb = lb;
377    initUB = this->ub = ub;

378    return S_OK;
379  }

380  /**
381   *  Given two doubles, x and y,
382   * returns:
383   *                 -1 if x is greater than y
384   *                  0 if x is equal to y
385   *                 +1 if x is less than y
386   */
387  STDMETHODIMP CImpIReal::compare(double x, double y, int *result) {
388    AFX_MANAGE_STATE(AfxGetStaticModuleState())
389
390      // similar to if (x > y)
391      if ((x - y) > EPSILON) {
392        *result = -1;
393        return S_OK;
```

```
394      }

395    //the stdlib function, abs, will not work here e.g: abs(-0.95)
396    //user defined function of ABS provided in util.cpp
397    // similar to if (x == y)
398    if (ABS(x - y) <= EPSILON) {
399      *result = 0;
400      return S_OK;
401    }
402
403    // similar to if (x < y)
404    if ((y - x) > EPSILON) {
405      *result = 1;
406      return S_OK;
407    }
408    return S_OK;

409  }

410  /**
411   * returns true if the lower bound is greater than or
412   * equal to the upper bound
413   */
414  STDMETHODIMP CImpIReal::isLbGtEqUb(BOOL *result) {
415    AFX_MANAGE_STATE(AfxGetStaticModuleState())

416      int res = -1;
417    compare(lb, ub, &res);

418    if (res <= 0)
419      *result = true;
420    else
421      *result = false;

422    return S_OK;
423  }

424  /**
425   * given two doubles, left and right, returns true if the value
426   * of left is greater than or equal to the value of right
427   */
428  bool CImpIReal::isLeftGtEqRight(double left, double right) {
429    int res = -1;
430    compare(left, right, &res);
431    if (res <= 0)
432      return true;
433    return false;
434  }
```

```
435 bool CImplReal::isLeftGtRight(double left, double right) {

436   if ((left - right) > EPSILON)
437     return true;
438   return false;
439 }

440 /**
441  * given the parameters that are required to convert from one scale to the
442  * other, calculates the new value on the other scale corresponding to the
443  * new value on the former scale.
444  * lowerBound: indicates whether the rounding mode has to be set up or down
445  * oldr:       The old value on scale one
446  * newr:       The new value on scale one
447  * min:        The minimum value on scale two(for which the new value has to be found)
448  * max:        The maximum value on scale two
449  * lb:         The minimum value on scale one
450  * ub:         The maximum value on scale one
451  * oldi:       The old value on scale two
452  * returns
453  *   newi:     The new value on scale two corresponding to the newr on scale one
454  *   result:   true if lb is greater than or equal to ub (this is just a sanity check)
455  * Note:
456  * Also deals with the situation where the values of oldr and newr
457  * are -ve and +ve infinities respectively.
458  */
459 STDMETHODIMP CImplReal::rtoi(BOOL lowerBound, double oldr, double newr,
460                              int min, int max, double lb, double ub,
461                              int oldi, int *newi, BOOL *result) {
462   AFX_MANAGE_STATE(AfxGetStaticModuleState())

463     if (ub == POS_INF)
464       ub = MAX_DBL;
465   if (lb == NEG_INF)
466     lb = MIN_DBL;

467   int val;
468   //the lower bound cannot be greater than or equal to the
469   //upper bound
470   compare(ub, lb, &val);
471   if (val >= 0)  {
472     *result = false;
473     return S_OK;
474   }

475   if (oldr == POS_INF)
476     oldr = MAX_DBL;
477   if (oldr == NEG_INF)
478     oldr = MIN_DBL;
```

```
479    if (newr == POS_INF)
480      newr = MAX_DBL;
481    if (newr == NEG_INF)
482      newr = MIN_DBL;

483    //return the same value if the new value is less than
484    //or equal to the old value
485    compare(oldr, newr, &val);
486    if (((val <= 0) && (lowerBound)) ||
487        ((val >= 0) && (!lowerBound))) {
488      *newi = oldi;
489      *result = true;
490      return S_OK;
491    }

492    if (lowerBound)
493      Flpt::roundDown();
494    else
495      Flpt::roundUp();
496    double denom = ub-lb;
497    //lb < 0 and ub > 0
498    if (denom == POS_INF) {
499      denom = MAX_DBL;
500    }
501
502    if (lowerBound)
503      Flpt::roundDown();
504    else
505      Flpt::roundUp();
506    double tmp = (double)max-min;
507
508    if (lowerBound)
509      Flpt::roundDown();
510    else
511      Flpt::roundUp();
512    //scaling factor for converting from the real scale to the integer scale
513    double sfrtoi = tmp/denom;
514
515    if (lowerBound)
516      Flpt::roundDown();
517    else
518      Flpt::roundUp();
519    tmp = newr-lb;
520    //lb < 0 , newr > 0
521    if (tmp == POS_INF) {
522      tmp = MAX_DBL;
523    }

524    if (lowerBound)
```

```
525       Flpt::roundDown();
526     else
527       Flpt::roundUp();
528     tmp = sfrtoi * tmp;

529     if (lowerBound)
530       Flpt::roundDown();
531     else
532       Flpt::roundUp();
533     tmp += min;

534     if (lowerBound) {
535       Flpt::roundUp();
536       *newi = (int)ceil(tmp);
537     } else {
538       Flpt::roundDown();
539       *newi = (int)floor(tmp);
540     }
541     *result = true;

542     return S_OK;
543   }

544   /**
545    * Given the bound to increment/decrement the value of lb/ub is
546    * incremented/decremented. The value of lb/ub can be
547    * incremented/decremented such that (lb <= ub) is always true.
548    * Note:
549    *       Takes into consideration the lower and the upper bounds may be
550    * infinities. In such a case:
551    *        When the lower bound is incremented the incremented value would be
552    * equal to the min. representable double according to the IEEE 754 standard.
553    *        When the upper bound is decremented the decremented value would be
554    * equal to the max. representable double according to the IEEE 754 standard.
555    * returns true if the lb/ub was successfully incremented/decremented. false
556    * otherwise
557    */
558   STDMETHODIMP CImpIReal::incrBound(BOOL lowerBound, int incr,
                double min, double max, int range, double initialValue, BOOL *result) {
559    AFX_MANAGE_STATE(AfxGetStaticModuleState())
560
561      if ((lowerBound) && (lb == NEG_INF)) {
562        lb = MIN_DBL;
563        *result = true;
564        return S_OK;
565      }

566    if ((!lowerBound) && (ub == POS_INF)) {
567      ub = MAX_DBL;
```

```
568     *result = true;
569     return S_OK;
570   }
571   if (min == NEG_INF)
572     min = MIN_DBL;
573   if (max == POS_INF)
574     max = MAX_DBL;
575   double left = lb, right = ub;
576   if (left == NEG_INF)
577     left = MIN_DBL;
578   if (right == POS_INF)
579     right = MAX_DBL;
580   if ((isLeftGtEqRight(left, right)) || (isLeftGtEqRight(min, max))) {
581     *result = false;
582     return S_OK;
583   }
584   if (initialValue == NEG_INF)
585     initialValue = MIN_DBL;
586   double *bound, oldBound;
587   if (lowerBound) {
588     oldBound = lb;
589     bound = &lb;
590   } else {
591     oldBound = ub;
592     bound = &ub;
593   }
594
595   double tmp;
596   if (lowerBound)
597     Flpt::roundDown();
598   else
599     Flpt::roundUp();
600   tmp = max-min;
601   if (tmp == POS_INF)
602     tmp = MAX_DBL;
603   if (lowerBound)
604     Flpt::roundDown();
605   else
606     Flpt::roundUp();
607   //scaling factor for converting from integer to real
608   double sfitor = tmp / (double)range;
609   if (*bound < 0) {
```

```
610       tmp = 0;
611       if (*bound != NEG_ZERO) {
612         if (lowerBound)
613           Flpt::roundDown();
614         else
615           Flpt::roundUp();
616         tmp = (incr * sfitor);

617         if (lowerBound)
618           Flpt::roundDown();
619         else
620           Flpt::roundUp();
621         *bound = initialValue + tmp;
622       }
623     } else {
624       if (*bound != POS_INF) {
625         if (lowerBound)
626           Flpt::roundDown();
627         else
628           Flpt::roundUp();
629         tmp = (incr * sfitor);

630         if (lowerBound)
631           Flpt::roundDown();
632         else
633           Flpt::roundUp();
634         *bound = initialValue + tmp;
635       }
636     }
637     *result = true;

638     if ((isLeftGtRight(*bound, initUB)) || (isLeftGtRight(lb, ub)) ||
639         (ABS(tmp) <= EPSILON)) {
640       *bound = oldBound;
641       *result = false;
642     }

643     return S_OK;
644   }

645   STDMETHODIMP CImpIReal::setLB(double lb) {
646     AFX_MANAGE_STATE(AfxGetStaticModuleState())
647
648       this->lb = lb;
649     return S_OK;
650   }

651   STDMETHODIMP CImpIReal::setUB(double ub) {
652     AFX_MANAGE_STATE(AfxGetStaticModuleState())
```

```
653
654     this->ub = ub;
655   return S_OK;
656 }

657 STDMETHODIMP CImpIReal::setBounds(double lb, double ub) {
658   AFX_MANAGE_STATE(AfxGetStaticModuleState())
659     this->lb = lb;
660   this->ub = ub;

661   return S_OK;
662 }

663 STDMETHODIMP CImpIReal::resetBounds() {
664   AFX_MANAGE_STATE(AfxGetStaticModuleState())
665
666     Real::resetBounds();

667   return S_OK;
668 }

669 STDMETHODIMP CImpIReal::getLB(double *pLB) {
670   AFX_MANAGE_STATE(AfxGetStaticModuleState())
671
672     *pLB = lb;
673   return S_OK;
674 }

675 STDMETHODIMP CImpIReal::getUB(double *pUB) {
676   AFX_MANAGE_STATE(AfxGetStaticModuleState())
677
678     *pUB = ub;
679   return S_OK;
680 }

681 STDMETHODIMP CImpIReal::getInitialBounds(double *pLB, double *pUB) {
682   AFX_MANAGE_STATE(AfxGetStaticModuleState())
683     *pLB = initLB;
684   *pUB = initUB;

685   return S_OK;
686 }

687 STDMETHODIMP CImpIReal::getBounds(double *pLB, double *pUB) {
688   AFX_MANAGE_STATE(AfxGetStaticModuleState())
689     *pLB = lb;
690   *pUB = ub;

691   return S_OK;
```

```
692 }

693 STDMETHODIMP CImplReal::toDecString(BSTR *pDec) {
694   AFX_MANAGE_STATE(AfxGetStaticModuleState())

695     ostrstream outBuf;
696   outBuf << "[ " << Flpt::toDecString(lb) << ", "
697          << Flpt::toDecString(ub) << " ]" << ends;
698   CString cs(outBuf.str());
699   *pDec = cs.AllocSysString();

700   return S_OK;
701 }

702 STDMETHODIMP CImplReal::toHexString(BOOL bias, BSTR *pHex) {
703   AFX_MANAGE_STATE(AfxGetStaticModuleState())

704     ostrstream outBuf;
705   outBuf << "[ " << Flpt::toHexString(bias, lb) << ", "
706          << Flpt::toHexString(bias, ub) << " ]" << ends;
707   CString cs(outBuf.str());
708   *pHex = cs.AllocSysString();

709   return S_OK;
710 }

711 STDMETHODIMP CImplReal::getHexLB(BOOL bias, BSTR *pLB) {
712   AFX_MANAGE_STATE(AfxGetStaticModuleState())
713
714     CString cs(Flpt::toHexString(bias, lb));
715   *pLB = cs.AllocSysString();
716   return S_OK;
717 }

718 STDMETHODIMP CImplReal::getHexUB(BOOL bias, BSTR *pUB) {
719   AFX_MANAGE_STATE(AfxGetStaticModuleState())
720
721     CString cs(Flpt::toHexString(bias, ub));
722   *pUB = cs.AllocSysString();
723   return S_OK;
724 }

725 STDMETHODIMP CImplReal::getDecLB(BSTR *pLB) {
726   AFX_MANAGE_STATE(AfxGetStaticModuleState())
727
728     CString cs(Flpt::toDecString(lb));
729   *pLB = cs.AllocSysString();

730   return S_OK;
```

```
731 }

732 STDMETHODIMP CImpIReal::getDecUB(BSTR *pUB) {
733   AFX_MANAGE_STATE(AfxGetStaticModuleState())
734
735     CString cs(Flpt::toDecString(ub));
736   *pUB = cs.AllocSysString();

737   return S_OK;
738 }
```

VITA

Surname: Parvataneni ***Given Names:*** Seshu

Place of Birth: Andhra Pradesh, India

Educational Institutions Attended

University of Victoria	1998 to 2003
Osmania University	1992 to 1996

Degrees Awarded

BE	Osmania University	1996

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: INTERACTIVE EXPLORATION OF NUMERICAL CONSTRAINT SATISFACTION PROBLEMS.

Author: ____________________

SESHU PARVATANENI
February XX, 2003