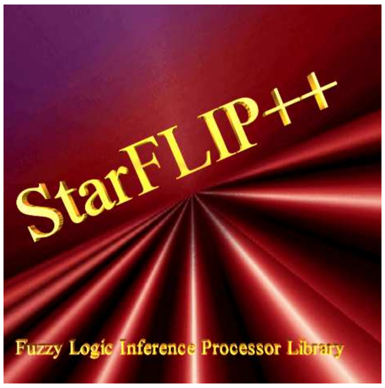


StarFLIP++ Version 1.0

A reusable iterative optimization library
for combinatorial problems with fuzzy constraints.

StarFLIP-Team
Database and Expert Systems Group (E184-2)
Institute of Information Systems
Vienna University of Technology
Paniglgasse 16, A–1040 Wien, Austria
Phone: +43–1–58801–6123 Fax: +43–1–5055304
URL: http://www.dbai.tuwien.ac.at/proj/StarFLIP/
E-Mail: sflip@dbai.tuwien.ac.at

DBAI–TR–97–11

**Christian Doppler Laboratory
for Expert Systems**

**Technische Universität Wien
Institut für Informationssysteme**

Foreword

Preface

This is the first editon of the documentation for the StarFLIP++ Library. It is the summary of one dissertation, around 10 Masters's theses and a multitude of papers presented at international conferences.

Network support

How to give us feedback about our project?

You can write an eMail to sflip@dbai.tuwien.ac.at. All messages will be stored in the StarFLIP++ mail archive (http://www.dbai.tuwien.ac.at/marchives/sflip/).

How to subscribe to the StarFLIP++ (StarFlipPlusPlus) mailing-list?

Announcement and Call for Participation for a SFLIP-MAIL-LIST. An automatic moderated mailing list (Unix-based Listproc) has been established. To participate, send an electronic mail message to listproc@dbai.tuwien.ac.at containing just the following line in the message body (the subject field of the message has no significance):

SUB SFLIP Name

where you replace "Name" by your real name. Your email address will be automatically extracted from the header of your mail. You will then get a welcome mail in response to your "SUBscribe" command above.

All sflip-mail should be sent to sflip@dbai.tuwien.ac.at for distribution amongst all subscribers.

If you prefer getting a digest once a week, send a

SET SFLIP MAIL DIGEST

command to the address listproc@dbai.tuwien.ac.at

Send a message containing only the word HELP to listproc@dbai.tuwien.ac.at to learn more about the commands at your disposition.

By the way, unsubscribing is just as easy as subscribing. Just send the command UNSUB SFLIP in the body of an otherwise empty mail to listproc@dbai.tuwien.ac.at from your mailing-address known to the system (if it change, write directly to sflip-owner@dbai.tuwien.ac.at).

WWW

http://www.dbai.tuwien.ac.at/proj/StarFLIP/

FTP

We have also established an anonymous FTP directory to enable people who are interested in StarFLIP++ to obtain updates, ...

host:	*ftp.dbai.tuwien.ac.at*
name:	*anonymous*
password:	*<your email address>*
directory:	*pub/sw/sflip*

Acknowledgements

We would also like to acknowledge the support received from the staff of Sony Austria (CD-ROM Production), Mr. Frans J. Faase (http://www.cs.utwente.nl/˜faase) for his html2tex converter (we've used a slightly modified version 2.7β), as well as thank all members of the Christian Doppler Laboratory for Expert Systems, of the Database and Expert Systems Group at the Institute of Information Systems of Vienna University of Technology, and our partners at the VOEST Alpine in Linz.

The StarFLIP-Team

Thank you for your interest in the StarFLIP++ Library.

◇

Markus
Bonner,
Friedrich Döll,
Jürgen Dorn, Gerhilde
Egghart, Georg Fischer, Mario
Girsch, Georg Gottlob, Johannes
Hajek, Witold Hendrysiak, Akram Jaro,
Thomas Länger, Joachim Lukasser, Stefan
Mayer, Ali Naraghi, Michael Prah, Andreas Raggl,
Claus Reichel, Bernhard Rhomberg, Wolfgang Rumplmaier,
Ulrich Santa, Johannes Sarg, Karl Schott, Christian
Schwaiger, Günther Skele, Wolfgang Slany,
Wolfgang Snopek, Christian Stary, Wolf-
gang Steindl, Klaus Stohl, Thomas
Tampier, Gabi Thalhammer,
Simone Ulreich, Wolf-
gang Winsauer
and Mazen
Younes

◇

Possible enhancements and future projects in the future

The GUI in the FLIP++ part needs some work (portability). The speed of DynaFLIP++ should be increased, the language needs some redesign. The last 5 % of computations in ConFLIP++ consume 90 % of the time. DomFLIP++ needs some reimplementation. The GUI for CheckFLIP++ is missing. Adding an index to this document.

Claus Reichel
Wolfgang Slany

Contents

List of Tables

List of Figures

Chapter 1

FLIP++

1.1 Knowledge Engineer

Every problem domain needs suitable datatypes and operators to be done efficiently. StarFLIP++ has the special feature of providing fuzzy datatypes and operators to effectively deal with real-world vagueness. Those are defined in FLIP++ together with all instruments needed for doing defuzzification and inference.

- Operators (cf. Section 1.1.1)
- Membership functions (cf. Section 1.1.2)
- Membership functions example (cf. Section 1.1.3)
- Rules (cf. Section 1.1.4)
- Rules example (cf. Section 1.1.5)
- Inferencing in FLIP++ (cf. Section 1.1.6)

1.1.1 Operators

FLIP++ defines a wide range of various operators that are needed to implement fuzziness. They include:

- **fAND** (predefined as: truth(x and y) = minimum (truth(x), truth(y));
- **fOR** (predefined as: truth(x or y) = maximum (truth(x), truth(y));
- **fNOT** (predefined as: truth(NOT X) = 1.0 - truth(x);

With the above operators it is possible to model everything that is usually done in boolean logic in the same fashion but within the extended fuzzy-logic domain. As not everyone agrees on how to define fuzzy operators it is possible to adjust them to ones own conception.

Together with the predefined operators it is possible to create new operators that better fit the problem at hand.

1.1.2 Membership functions

In classical set theory a subset A of a set S maps all elements x of S to the set $\{0, 1\}$ where 0 means x is not a member of A while 1 means that x is a member of A.

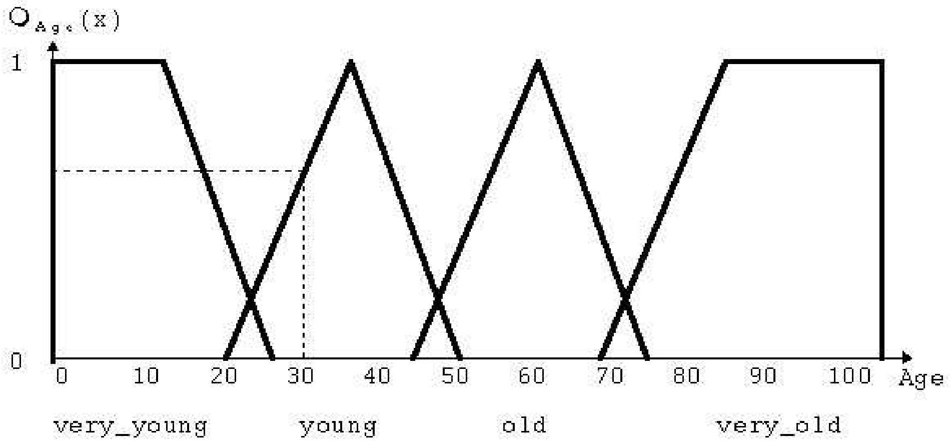
A fuzzy subset F of a set S maps all elements x of S to the interval [0..1] with the meaning of 0 and 1 unchanged but values in-between representing intermediate degrees of membership. This kind of mapping is usually described by a function called the **membership function** (or fuzzy subset) of F.

As membership functions and logic are generalizations of classical set theory there is no conflict between fuzzy and crisp methods and it is possible to use them both in the system.

One major advantage of fuzzy logic is that it is usually less complicated to define membership functions compared with crisp functions and that they are more intuitively and easier to grasp (just think of the famous example of the definition of tallness of a person ...).

In StarFLIP++ you have to define a membership function for every coinage of a linguistic variable. This can be done in different ways:

1. You can define membership functions as shapes with the x-axis defining the measure (for example temperature) and the y-axis defining the degree of membership in the interval [0..1]. Commonly used shapes like triangular or trapezoidal are already present but again it is possible to define a new, individual shape. The following pictures show first an example of a single membership function and then the membership functions for every coinage of a linguistic variable. As you can see membership functions can be of different shapes and usually overlap. It is also possible to mix fuzzy and crisp functions when appropriate.



1

Figure 1.1: A typical membership function

Figure 1.1 on page 3 shows a typical membership function for the famous age example. The dashed line indicates that *Age=30* belongs to the linguistic variable *young* to a degree of *0.7*.

2. The other way to define **all** membership functions for a linguistic variable is to define a 6-tuple consisting of support range (upper end and lower end of the measure) a median its distribution (around the median) its dilatation (this states the degree of fuzziness in the interval [0..1] with higher fuzziness nearer to 1) and its importance (this parameter lies in the [0..1] interval with higher importances nearer to 1).

1.1.3 Example

(-100, 100, 0, 0.5, 1, 0.9) defines all membership functions of a linguistic variable with measures in the range of -100 to 100 a median of 0, a distribution of 0.5 around the median, fuzziness 1 and an importance of 0.9 (the importance belongs of course to the linguistic variable).

Furthermore it is necessary to define the number of coinages for the linguistic variables and their names.

1.1.4 Rules

To actually be able to get something out of the system we need to specify **rules**. Rules consist of an antecedent (or premise) and a conclusion and look like this:

1.1.5 Example

if al_temp is hot and ni_temp is medium then al-ni_heat is acceptable.

al_temp and ni_temp are input variables with hot and medium as membership functions defined on al_temp and ni_temp, respectively. Acceptable is a membership function defined on the output variable al-ni_heat. In the antecedent we determine to what degree the rule applies while in the conclusion the output variable is assigned a value. It is also possible that the conclusion consists of more than one output variable or that there is more than one conclusions for a rule.

In FLIP++ it is also possible to assign a value in the range of [0..1] to a rule to state how accurate (and therefor trustworthy) the rule is

if al_temp is hot and ni_temp is medium then al-ni_heat is acceptable with accuracy 0.7.

1.1.6 Inferencing in FLIP++

FLIP++ generates the following steps during the inference process:

1. **Fuzzification** where the degree of truth for the premise of each rule is determined for an actual value.

2. **Inference** using either max-min or sum-product inferencing. The first method clips off the membership function of the output variable(s) at the computed degree of truth for the premise of the rule (min) and then combines an output fuzzy subset by taking the pointwise maximum for all the fuzzy subsets assigned to a variable (max). The second method scales the membership function of the output variable(s) with the computed degree if truth for the premise of the rule (product) and then combines an output fuzzy subset by taking the pointwise sum for all the fuzzy subsets assigned to a variable (sum).

3. **Defuzzification** converts a fuzzy output set to a crisp number. From the various methods in use FLIP++ uses the CENTROID method at this time, though it is planned to incorporate other popular methods as well as allowing the user to introduce own methods.

1.2 FLIP++ library description

1.2.1 FLIP++ modules

- flippara.h / flippara.cpp (cf. Section 1.2.2)

- flipcvar.h / flipcvar.cpp (cf. Section 1.2.3)
- flipoper.h / flipoper.cpp (cf. Section 1.2.4)
- fliprule.h / fliprule.cpp (cf. Section 1.2.5)

With the use of these four object oriented modules the developer is provided with a tool to develop his/her own fuzzy logic inference processor.

1.2.1.1 Important notions

- linguistic variable
- linguistic term
- range (lower end, upper end)
- median
- distribution
- dilatation
- importance
- (instantiated) membership function
- dilatation length
- crisp value
- defuzzification
- fuzzy operator
- cover
- match value
- certainty

1.2.1.2 Some helpful remarks...

Throughout the description the terms method and attributes are applied to describe a class. The notion method corresponds to the more C++ like term membership function, the attribute notion should be quite clear. There is also a distinction between a class and an object, where class refers to the abstract declaration and definition while object means the instance of a class.

Furthermore, we assume that the reader is familiar with the basics of objecto riented programming, which might help understanding the description and source code as well. It is highly recommended for the reader to go through the following description and reading the source code simultaneously switching back and forth between the description and the source code.

1.2.2 flippara.h / flippara.cpp

This module provides the developer with the most basic functions in order to handle the 'administrative chores' necessary for the definition of linguistic variables. Among these chores are creating, initializing, adding, removing parameter sets and, furthermore, basic functions for input/output. The most important concept of this module is the list. Each new basic class introduced in the module is strongly related to another class which is a simple linear list of objects of that basic class. The list below shows the basic classes/list classes contained in this module. Then a detailed description of the functionality of each class is given.

- Name (cf. Section 1.2.2.1) / NList (cf. Section 1.2.2.2)
- Colour (cf. Section 1.2.2.3) / CList (cf. Section 1.2.2.4)
- ParameterSetBase (cf. Section 1.2.2.5)
- ParameterSet (cf. Section 1.2.2.6) / PSList (cf. Section 1.2.2.7)

1.2.2.1 Name

This class looks quite general and is used to represent the name of a linguistic term (cf. Section 1.2.1.1), made explicit in the attribute *nam*, whose type is a type defined string allowing a maximum of 40 characters to be used in a string. As already mentioned above each basic class is used in a list class, which is why there is the *next* attribute that is the pointer to the next Name (cf. Section 1.2.2.1) object in the list.

The class has constructors either with or without parameter. Using the former a default name is attached to the attribute *nam*, in the latter case the attribute *nam* is initialized with the value of the parameter. The method *show* is used to write the current value of *nam* to the standard output device. The methods *read* and *write* handle input and output to files. The method *copy* sets *nam* to the *nam* value of another Name (cf. Section 1.2.2.1) object. Simple access methods for the attributes round up the class. It is important to note the difference between *name* and *Name*, the former is a type -defined string with a maximum of 40 characters used for the *nam* attribute, whereas the latter is the class described here.

1.2.2.2 NList

This class is a linear list of objects of the Name (cf. Section 1.2.2.1) class. It is used to represent all the different linguistic terms (cf. Section 1.2.1.1) that can be assigned to a specific linguistic variable (cf. Section 1.2.1.1) . In the following description the Name objects are referred to as elements of the list. NList (cf. Section 1.2.2.2) has three attributes *first*, *last* and *number*, where *first* is a pointer to the first element in the list, *last* is a pointer to the last element in the list, and *number* stores the number of elements currently contained in the list.

Again, there are two constructors, one without parameter that provides an empty list and one with an integer parameter creating a list with as many elements as the value of the parameter specifies. For values within the range of 2 to 9 there are default names which are attached to the Name objects. These names can be found and changed in the NList constructor in the flippara.h / flippara.cpp (cf. Section 1.2.2) file. The destructor deletes the elements of the list according to a clean memory management. The method *show* is a simple extension of the *show* method of the Name object and writes the values of the entire list, i.e. the value of each element in the list, to the standard output device. The methods *read* and *write* handle input and output to files by reading/writing the number of elements and each element itself from/to a file. The method *add* adds an element to the list. The method *find* accesses an element with a certain index and returns a pointer to that element, whereas *find_n* accesses an element with a certain index and returns a pointer to the attribute *nam* of that element, the same effect can be attained by using the array-operator []. The method *copy* is used to copy the elements from another NList (cf. Section 1.2.2.2) object by adding new elements to the end of the list using the *add* method. The method *remove_all* deletes all elements of the list and hence supplies an empty list. Furthermore, there are simple methods to set and get the values of the three attributes. In order to avoid any name conflicts, especially when multiple inheritance is used, all the methods of the Name (cf. Section 1.2.2.1) class come in duplicate declaration:

with one name as mentioned above, and the other by putting the method name and the name of its class together seperated by an underscore. This practice makes the names of methods globally distinct and is used throughout the whole system. (e.g. *show* and *show_NList* are virtually the same methods)

1.2.2.3 Colour

This class stores the colour value of another object. It is mainly used for the graphical representation of linguistic terms (cf. Section 1.2.1.1) and linguistic variables (cf. Section 1.2.1.1) respectively. The attributes of the class are *colour*, which stores the value of the colour and is of an enumeration type (*colourtype*), and *next*, which is used to implement the already familiar list concept.

There are two constructors, the default constructor initializes the attribute *colour* with a specifically fixed colour (white), the alternative constructor uses a parameter for the colour to be attached to the object's attribute *colour*. You can look up the list of possible colours in the constructor in flippara.h / flippara.cpp (cf. Section 1.2.2) file or, of course, in the declaration of the enumeration type *colourtype* in the flippara.h / flippara.cpp file. Again, there is a method *show* that writes the current value of the attribute *colour* to the standard output device, the methods *read* and *write* handle input and output to files. The *copy* method copies the *colour* attribute value of another Colour (cf. Section 1.2.2.3) object to the current one. Simple procedures to set the values of the attribute *colour* and to get the values of the attributes *colour* and *next* round up the methods of the class.

1.2.2.4 CList

This class is a linear list of objects of the Colour (cf. Section 1.2.2.3) class. It can be viewed as a list of the various colour values of a linguistic variable (cf. Section 1.2.1.1)'s linguistic terms (cf. Section 1.2.1.1). In the following description the Colour objects of the list are referred to as elements of the list. The three attributes are represented as *first* and *last*, which are pointers to an object of the type Colour, and *number*, an integer variable, giving the number of elements currently contained in the list.

The default constructor without parameters initializes the object with an empty list, whereas the constructor with the integer parameter number creates an object that has as many elements in the list as the value of number indicates. Similar to the NList (cf. Section 1.2.2.2) class, for values between 2 and 9 there are default colours that are attached to the Colour objects of the CList (cf. Section 1.2.2.4) during initialization. These colours can be seen in the constructor in the flippara.h / flippara.cpp (cf. Section 1.2.2) file.The method *show* writes the value of each element in the list to the standard output device. The methods *read* and *write* handle input and output to files by reading/writing the number of elements and each element of the list from/to the file. The *add* method adds a new Colour object as an element to the list, by default the colour attribute of the object is white. The *copy* method copies a complete set of elements from an existing CList (cf. Section 1.2.2.4) object into the current CList object, the elements are copied by the *add* method. The method *remove_all* deletes all elements of the list. The *find* method is used to access an element in the CList object by its index. A modification of the method *find* is the method *find_ct* where instead of a pointer to an element with a specific index a pointer to the element's attribute *colour* is returned, the same effect can be attained by using the array-operator []. Furthermore, there are trivial methods to set and get the values of the CList's attributes. Among them are the methods to set and get the first and last element of the list and the number of elements.

1.2.2.5 ParameterSetBase

This class is a composition of all parameters that are needed to describe a linguistic variable (cf. Section 1.2.1.1) among them are the range (cf. Section 1.2.1.1) with its lower and upper end, the median (cf. Section 1.2.1.1), the distribution (cf. Section 1.2.1.1), the dilatation (cf. Section 1.2.1.1), and the importance (cf. Section 1.2.1.1). The attributes of the object are the parameters, called *range_lower_end*, *range_upper_end*, *median*, *distribution*, *dilatation*, and *importance*. All these attributes are simple float values. Note that the distribution (cf. Section 1.2.1.1) must have a value between 0.1 and 1, while dilatation (cf. Section 1.2.1.1) and importance (cf. Section 1.2.1.1) must be in the interval [0,1].

As usual, there are two constructors. The default constructor initializes all the parameters mentioned above with default values, which can be seen in the flippara.h / flippara.cpp (cf. Section 1.2.2) file. The alternative constructor takes the values of all the parameters as input and initializes the object correspondingly. If there are any inconsistencies in the parameters, the constructor tries to eliminate them. For example, if the *range_lower_end* is not a smaller value than the *range_upper_end* parameter, the two values are swapped. If the *median* is not within the range (cf. Section 1.2.1.1), which is the interval defined by the boundaries *range_lower_end* and *range_upper_end*, then the *median* will be set to the mid of the range. Furthermore, the boundaries of the parameters *distribution*, *dilataton*, and *importance* are the utmost values that can be reached and cannot be exceeded on either side. The method *show* writes the values of the parameters to the standard output device. The methods *read* and *write* do the same, except they read from/write to files instead of standard output. The *copy* method copies the parameters of an existing ParameterSetBase (cf. Section 1.2.2.5) object to the current object. Apart from these methods, there are the trivial set and get methods enabling us to make changes to the value of a specific parameter.

1.2.2.6 ParameterSet

This class is an aggregation of the three basic classes NList (cf. Section 1.2.2.2), CList (cf. Section 1.2.2.4) and ParameterSetBase (cf. Section 1.2.2.5), or expressed more technically, it is derived from these three classes by multiple inheritance. Hence it combines the descriptions of the linguistic terms (cf. Section 1.2.1.1), i.e. names and colours, the parameters of a linguistic variable (cf. Section 1.2.1.1). Again this class is defined as an element of a list, called PSList (cf. Section 1.2.2.7) and will be described in the next class. The attributes of the ParameterSet are *ps_name* containing the name of the ParameterSet (cf. Section 1.2.2.6) and *next* referencing another ParameterSet object as used in PSList. Of course, ParameterSet knows all the attributes from which the classes are derived from.

The default constructor initializes an empty ParameterSet consisting of a default name, a *next* pointer to the next ParameterSet, a default ParameterSetBase (cf. Section 1.2.2.5) object, and the list of objects NList (cf. Section 1.2.2.2) and CList (cf. Section 1.2.2.4), which are both empty. The alternative constructor initializes a ParameterSet object with the values of its input parameters. These parameters are the same as with the ParameterSetBase object plus two pointers that are the reference to an existing NList and CList object and the name of the ParameterSet (cf. Section 1.2.2.6). The method *show* writes the complete ParameterSet to the standard output device, i.e. all the values of its attributes. Similarly the methods *read* and *write* handle input and output to files. The *copy* methods takes a pointer to an existing ParameterSet as input and copies all its attributes to the current one. There are also simple get methods to obtain the values of the attributes, among them are get methods for the attributes *name* and *next*, and get methods to extract the whole ParameterSetBase object, NList (cf. Section 1.2.2.2) object or CList (cf. Section 1.2.2.4) object encapsulated in the object. Moreover, there is a method to set the name of the ParameterSet.

1.2.2.7 PSList

The class introduces a now already familiar concept of the list class. It represents a linear list of ParameterSet (cf. Section 1.2.2.6) objects, which can be viewed as an aggregation of ParameterSets of several linguistic variables (cf. Section 1.2.1.1). The attributes of this class are *first* and *last*, which are pointers to ParameterSet objects, and *number*, which is an integer storing the number of elements in the list.

The constructor initializes an object as an empty list. The destructor deletes all elements of the list according to a clean memory management. The method *show* simply writes all significant attributes of the elements to the standard output device. The same procedure is followed by the methods *read* and *write* by reading/writing the number of elements and each element of the list from/to files. The *add* method without parameter adds an empty ParameterSet (cf. Section 1.2.2.6) to the list, whereas the *add* method with a pointer to a ParameterSet as an argument adds a copy of this ParameterSet to the list. The *add* method with a PSList (cf. Section 1.2.2.7) reference as an argument copies the elements of the complete PSList into the current one. The method *remove_all* deletes all elements of the list. The method *find* is used to access an element by its index. The *remove* method deletes the element with the index specified in the argument. The values of the attributes can be retrieved by simple get methods.

1.2.3 flipcvar.h / flipcvar.cpp

This module focuses on the particulars of a linguistic variable (cf. Section 1.2.1.1). First, the classes FuzzySetBase (cf. Section 1.2.3.5), PlimFuzzySet (cf. Section 1.2.3.6) and PlimList (cf. Section 1.2.3.7) define the characteristic s of linguistic terms (cf. Section 1.2.1.1), especially the handling of membership functions (cf. Section 1.2.1.1) and their instances. Then the classes CombinedVariable (cf. Section 1.2.3.8) and CVList (cf. Section 1.2.3.9) aggregate linguistic terms (cf. Section 1.2.1.1) (PlimList (cf. Section 1.2.3.7)) and linguistic variable (cf. Section 1.2.1.1) parameters (ParameterSetBase (cf. Section 1.2.2.5)). Below you can see a list of the classes contained in this module. After that, a detailed description of each class is provided.

- FloatValue (cf. Section 1.2.3.1) / FList (cf. Section 1.2.3.2)
- Point (cf. Section 1.2.3.3) / PList (cf. Section 1.2.3.4)
- FuzzySetBase (cf. Section 1.2.3.5)
- PlimFuzzySet (cf. Section 1.2.3.6) / PlimList (cf. Section 1.2.3.7)

- CombinedVariable (cf. Section 1.2.3.8) / CVList (cf. Section 1.2.3.9)

1.2.3.1 FloatValue

This represents a float value, which is used as an element of a linear list provided by the FList (cf. Section 1.2.3.2) class. The attribute *value* is a float variable, whereas the attribute *next* is a reference to another FloatValue (cf. Section 1.2.3.1) object which is needed to build a linear list of FloatValue objects as used in the FList class. This technique is the same as the one applied with the following objects Name (cf. Section 1.2.2.1), Colour (cf. Section 1.2.2.3), ParameterSet (cf. Section 1.2.2.6), NList (cf. Section 1.2.2.2), CList (cf. Section 1.2.2.4), and PSList (cf. Section 1.2.2.7) respectively in the flippara.h / flippara.cpp (cf. Section 1.2.2) module.

The constructor initializes the object with a float value. Furthermore, there are simple set and get methods for the values of the attributes *value* and *next*.

1.2.3.2 FList

This class is a linear list of FloatValue (cf. Section 1.2.3.1) objects.The set of attributes consists of *first* and *last*, which are pointers to the first and last FloatValue objects contained in the list, and *number*, which is an integer variable that stores the numbers of elements contained in the list.

An empty object of this class is initialized by the constructor. According to a clean memory management the destructor deletes all elements of the list. The method *add* adds a new FloatValue object to the list initialized with the value of its float parameter. The *find* method accesses an element of the list by its index. It is a modified version of the *find* method since it does not return a reference to the found element object but returns the value of this object which is a float value. As an alternative the array-operator [] can be used instead of the *find_v* method. The *remove* method deletes an element of the list specified by its index. The *remove_all* method deletes all elements of the list. Apart from these methods, there are simple methods to obtain the values of the attributes.

1.2.3.3 Point

This class represents a Point (cf. Section 1.2.3.3) of a membership function (cf. Section 1.2.1.1), which generally consists of several points describing its shape. The attributes of the class are *x_value* and *y_value* - simple float values that represent the coordinates of the Point - and the *next* attribute, which is a reference to another Point object as needed by the list object PList (cf.

Section 1.2.3.4), an already familiar technique for the attentive reader of the previous sections.

The default constructors initialize the object with a default Point (0, -1). The alternative constructor initializes the object with the Point specified by the parameters for the x and y-coordinates. The *show* method writes the values of the coordinates to the standard output device. The methods *read* and *write* handle input/output on files. Simple get and set methods round up the class.

1.2.3.4 PList

This class is the representation of a linear list of Point (cf. Section 1.2.3.3) objects. The attributes are *first* and *last*, which are pointers to Point (cf. Section 1.2.3.3) objects, and *number*, which is an integer and stores the number of elements of the list.

The constructor creates an empty list, whereas the destructor deletes all elements of the list according to a clean memory management. The method *show* writes the values of all elements to the standard output device. The methods *read* and *write* handle input and output to files by reading/writing the number of elements of the list and their values to files. The *add* method adds a new element to the list, it needs the values of the x- and y-coordinates of the Point object as input, after that the Point object is created and appended to the list. The method *find* accesses an element of the list by its index and returns a pointer to the found Point object. There are two modifications of this *find* method: *find_x* and *find_y*. They also access an element of the list by a specified index and then return its x-coordinate or y-coordinate respectively. Furthermore, the method *find* can be called with an x- and y-coordinate as input, this returns the index of the first Point with these coordinates or the value -1 indicating that no such Point is in the list. Another possibility is to call the *find* method with a reference to a Point (cf. Section 1.2.3.3) as input, then the same procedure is applied as in the previous variant with the coordinates retrieved from the Point object parameter. As an alternative to the *find* method with the index parameter the array-operator [] can be used. The method *insert* can either be used with a Point object as a parameter or with coordinates as parameters. The effect is the same, a new element with the values of the parameters is inserted into the list, the insertion is made in a way that the elements are kept in order with ascending x-coordinates, and in a second order with ascending y-coordinates if x-coordinates are the same. Furthermore the insertion procedure ensures that the same element cannot be contained in the list twice. The method *remove* deletes an element of the list specified by its index. The method *remove_all* initializes the list by deleting all its elements. The *copy* method adds all elements of an existing PList (cf. Section 1.2.3.4) object to the current PList object. Furthermore, there are again simple methods to obtain the values of the attributes.

1.2.3.5 FuzzySetBase

This class contains some basic parameters of a linguistic term (cf. Section 1.2.1.1), which are its name, lower_support_end, upper_support_end, and its colour. The attributes of the object corresponding to the parameters of the FuzzySetBase (cf. Section 1.2.3.5), namely *linguistic_term* of the type Name (cf. Section 1.2.2.1), *lower_support_end* and *upper_support_end* of the type float, and *fs_colour* of the object type Colour (cf. Section 1.2.2.3).

The default constructor initializes the parameters mentioned above with default values, which can be seen in the flipcvar.h / flipcvar.cpp (cf. Section 1.2.3) file. The alternative parameter needs the values of all these parameters as input and sets them accordingly. The method *show* writes the values of the parameters to the standard output device, whereas the methods *read* and *write* do the same, except to a file instead of standard output. The *copy* method copies the parameters of an existing FuzzySetBase object to the parameters of the current object. Furthermore, there are trivial methods to get and set the values of the parameters.

1.2.3.6 PlimFuzzySet

This class is derived from FuzzySetBase (cf. Section 1.2.3.5), as an extension it contains the default and instantiated membership function (cf. Section 1.2.1.1) of a linguistic term (cf. Section 1.2.1.1). The attributes of the class are the two PList (cf. Section 1.2.3.4) objects *msf* and *imsf* for the membership function (cf. Section 1.2.1.1) and instantiated membership function (cf. Section 1.2.1.1), the *match_value*, which is a float value, the *function*, which is the name of membership function and a Name (cf. Section 1.2.2.1) object, and the *next* pointer to another PlimFuzzySet (cf. Section 1.2.3.6).

The default constructor initializes the object, which is the same procedure as with the FuzzySetBase (cf. Section 1.2.3.5) object. Additionally, the attributes *match_value*, *function* and *next* are initialized. *Match_value* represents the value of intersection between the crisp value of a linguistic variable (cf. Section 1.2.1.1) and the membership function (cf. Section 1.2.1.1) of a specific linguistic term. With the alternative constructor it is possible to specify the parameters of the FuzzySetBase (cf. Section 1.2.3.5). The method *show* writes the underlying FuzzySetBase, the match value (cf. Section 1.2.1.1) and the Points (cf. Section 1.2.3.3) of the default and instantiated membership function (cf. Section 1.2.1.1) to the standard output device. The same procedure is applied in the *read* and *write* method, except they handle input and output to files instead of standard output. The method *calculate_match_value* calculates the *match_value* of the membership function for a specific crisp value (cf. Section 1.2.1.1), which is the parameter of the method. The calculation is a simple intersection between the vertical line defined by the crisp value and the polygon defined by the Points of the membership function. The match value (the y-coordinate of the intersection Point (cf. Section 1.2.3.3)) indicates the satisfaction of the linguistic term (cf. Section 1.2.1.1) achieved by a crisp value (cf. Section 1.2.1.1). The method *build_imsf* simply calculates the instantiated membership function (cf. Section 1.2.1.1) according to the current match value. The procedure builds a Point list from the default membership function where only those parts of the polygon below the match value (cf. Section 1.2.1.1) are of interest, so everything above the match value in the default membership function (cf. Section 1.2.1.1) is cut off by a horizontal line. The *add_msf_point* method adds a new Point (cf. Section 1.2.3.3) to the membership function, the Point is specified by its coordinates and appended to the list through the *insert* method of the PList (cf. Section 1.2.3.4) class. The *delete_msf_Point* method removes a Point (cf. Section 1.2.3.3) from the membership function (cf. Section 1.2.1.1), the Point is specified by its index and removed by the *remove* method of the PList (cf. Section 1.2.3.4) class. The methods *get_msf_Point_number* and *get_imsf_Point_number* return the number of Points contained in the membership function (cf. Section 1.2.1.1) and instantiated membership function (cf. Section 1.2.1.1) polygons. The *copy* method copies all the contents from an existing PlimFuzzySet (cf. Section 1.2.3.6) object into the current object. The *get_FSB method* returns a FuzzySetBase object containing the same parameters as the current PlimFuzzySet object. Apart from these methods, there are simple methods to set and get coordinates of the membership function (cf. Section 1.2.1.1)'s Points (cf. Section 1.2.3.3), and to get coordinates of the instantiated membership function (cf. Section 1.2.1.1) Points (cf. Section 1.2.3.3). Besides, a get and set method for the *match_value*, the *name* of the membership function and a get method for the *next* pointer, which is a reference to an other PlimFuzzySet (cf. Section 1.2.3.6) used by the PlimList (cf. Section 1.2.3.7) object, are availabe. Furthermore, references to the membership function and the instantiated membership function can be retrieved as pointers to a PList (cf. Section 1.2.3.4) object by get methods.

1.2.3.7 PlimList

This class is used to create a linear list containing PlimFuzzySet (cf. Section 1.2.3.6) objects. This can be viewed as a linguistic variable (cf. Section 1.2.1.1) consisting of various linguistic terms (cf. Section 1.2.1.1). The three additional attributes apart from the inherited ones are *first* and *last*, the pointers to the first and last element in the list, and *number*, which stores the number of elements contained in the list.

Its constructor builds an empty list, it has no parameters. The destructor deletes all elements of the list according to a clean memory management. The method *show* writes all the characteristic parameters of each element of the list to the standard output device by simply calling the *show* method of the PlimFuzzySet objects. The *read* and *write* methods handle input and output to files by reading/writing the number of elements and their contents from/to files. The method *add* appends a new element to the end of the list. Without parameters a default PlimFuzzySet is added to the list, whereas with parameters it is possible to specify the FuzzySetBase parameters of the new PlimFuzzySet object as well as the dilatation length (cf. Section 1.2.1.1) from both the lower and the upper end, this has an influence on the shape of the membership function (cf. Section 1.2.1.1). The *find* method accesses an element of the list by its index, alternatively the array-operator [] can be used. The method *get_fs_index* accesses an element of the list by the name of the linguistic term (cf. Section 1.2.1.1) and returns its index in case of existence. The method *get_membership* returns the y-coordinate of a membership function's Point specified by its x-coordinate, the element, whose membership function is used, is specified by its index. The method *get_names_of_vars* returns an NList (cf. Section 1.2.2.2) object containing the names of all linguistic terms (cf. Section 1.2.1.1) of the elements in the list. The *check_match* method checks whether the linguistic variable, i.e. the linguistic terms in the linear list, has already instantiated membership functions (cf. Section 1.2.1.1), which in turn means that the linguistic variable (cf. Section 1.2.1.1) has been instantiated and the match values (cf. Section 1.2.1.1) are greater than or equal to zero. The method *set_all_mvs* sets the match values of all elements, i.e. all linguistic terms, to a specific value, which is the parameter of the method, and, after that, it builds for each element the instantiated membership function. The method *add_msf_Point* adds a new Point (cf. Section 1.2.3.3) to the membership function (cf. Section 1.2.1.1) of the element specified by its index, whereas the method *delete_msf_Point* removes a Point specified by its index from the membership function (cf. Section 1.2.1.1) of an element also specified by its index. There is a restriction that a membership function must have at least three Points (cf.

Section 1.2.3.3). The *copy* method appends all elements of an existing PlimList (cf. Section 1.2.3.7) object to the current object. The method *remove_all* deletes all elements of the list. The method *put_imsf_into_cover* is used to build the cover (cf. Section 1.2.1.1) of the instantiated membership functions of all linguistic terms, generally speaking, the cover (cf. Section 1.2.1.1) is the contour of the instantiated linguistic variable (cf. Section 1.2.1.1) which consists of al l the instantiated membership functions of linguistic terms. The function creates a union list of the existing cover (cf. Section 1.2.1.1) and the instantiated membership function (cf. Section 1.2.1.1) to be added, furthermore a cross list containing all intersection Points (cf. Section 1.2.3.3) between the cover (cf. Section 1.2.1.1) and the instantiated membership function is created. Then all Points that are of no interest to the creation of the new cover (cf. Section 1.2.1.1) are systematically removed, among them are the Points of the cover that are below the instantiated membership function (cf. Section 1.2.1.1) and the Points (cf. Section 1.2.3.3) of the instantiated membership function that are below the cover. After that, the cross list and the union list are put together and all redundant Points, i.e. three Points on a straight line, are removed with the new cover as the result of the operation. The methods *above* and *picture* are needed for the calculation of the relative position of a Point and a line, the method *crossPoint* simply calculates the Point (cf. Section 1.2.3.3) of intersection between two lines. Furthermore, there are simple methods to access the attributes of the object. Their implementation is rather straightforward and they are generally augmented methods of the class PlimFuzzySet (cf. Section 1.2.3.6) by the possibility to access the elements of the list by their index.

1.2.3.8 CombinedVariable

Its origin is derived from the ParameterSetBase (cf. Section 1.2.2.5) class and the PlimList (cf. Section 1.2.3.7) class where it is a representation of a linguistic variable (cf. Section 1.2.1.1) consisting of various linguistic terms with their own parameters, membership functions (cf. Section 1.2.1.1) and instantiated membership functions (cf. Section 1.2.1.1). The attributes of the linguistic variable are the *cv_name* (of type Name (cf. Section 1.2.2.1)), the *crisp_value* (of type float), and *next* - which refers to the next linguistic variable in the list.

The default constructor initializes the linguistic variable with a default (empty) PSList (cf. Section 1.2.2.7), a default ParameterSetBase (cf. Section 1.2.2.5) and a default linguistic variable name, the attributes *next* and *crisp_value* are also initialized. The alternative constructor has two parameters, an existing ParameterSet (cf. Section 1.2.2.6) and a name for the linguistic variable. It initializes the ParameterSetBasewith the values of the ParameterSet and the PlimList (cf. Section 1.2.3.7) is cons tructed with as many elements as contained in the ParameterSet objects CListXS (cf. Section 1.2.2.4) and NList (cf. Section 1.2.2.2) respectively. Each element of the PlimList object is initialized with attributes derived from the ParameterSet values, this refers to the *name*, *colour*, *support* and *dilatation length* of the element. The method *show* writes the values of the parameters of the linguistic variable (cf. Section 1.2.1.1) to the standard output device, this includes the *name*, *crisp_value*, all parameters of the ParameterSetBase object, and all parameters of the PlimList and its elements. The same procedure is applied in the methods *read* and *write*, except that they are reading/writing from/to files. The method *set_value* sets a new crisp value (cf. Section 1.2.1.1), and then starts for each linguistic term (cf. Section 1.2.1.1) of the linguistic variable (cf. Section 1.2.1.1), i.e. each element of the PlimList (cf. Section 1.2.3.7), the calculation of the match value (cf. Section 1.2.1.1) and instantiated membership function (cf. Section 1.2.1.1). The method *copy* copies an existing linguistic variable (cf. Section 1.2.1.1) to the current object, i.e. all the parameters of the ParameterSetBase (cf. Section 1.2.2.5), the whole PlimList (cf. Section 1.2.3.7), the linguistic variable (cf. Section 1.2.1.1) name, and the crisp value (cf. Section 1.2.1.1) are copied. This can be achieved by applying the following two methods. The *method get_PSB* returns a ParameterSetBase (cf. Section 1.2.2.5) object identical to the one contained in the current linguistic variable object. The method *copy_PlimList* copies all the elements of an existing PlimList to the actual linguistic variable object. Furthermore, there are simple methods to get and set the values of the name and the crisp value, and to get a reference to the next linguistic variable, since the linguistic variable object can be an element of a linear list, the CVList object.

1.2.3.9 CVList

This class puts together various linguistic variables (cf. Section 1.2.1.1) in a linear list. The attributes of the list are *first* and *last*, which reference linguistic variables, and *number*, which stores the number of elements in the list.

The constructor initializes an empty list, whereas the destructors deletes all elements of the list according to a clean memory management. The method *show* writes the contents of all elements of the list to the standard output device, whereas the method *read* and *write* handle input/output to files. The method *add* without parameters adds a default linguistic variable (cf. Section 1.2.1.1) to the CVList (cf. Section 1.2.3.9) object, i.e. a linguistic variable (cf. Section 1.2.1.1) initialized with its default constructor containing an empty PlimList (cf. Section 1.2.3.7) and a default ParameterSetBase (cf. Section 1.2.2.5). The *copy* method copies all elements of an existing CVList (cf. Section 1.2.3.9) to the current one. The method *add* with a Combined-Variable (cf. Section 1.2.3.8) reference as a parameter adds this linguistic variable to the CVList object. The *add* method with a CVList as a parameter adds only those elements of this list that are not already contained in the current object. The method *find* accesses a linguistic variable in the list by its index. As an alternative to the *find* method the array operator [] can be used. The method *find_cv* accesses a linguistic variable in the list by its name. The methods *find_cv_by_pointer* and *find_cv_by_name* are the reverse of the previous find methods, they have a linguistic variable reference as parameter and return the index of this linguistic variable in case it exists. The method *remove* deletes the element of the list specified by its index. The method *remove_all* deletes all elements of the list.

1.2.4 flipoper.h / flipoper.cpp

This module provides all the necessary classes and methods used for fuzzy operators (cf. Section 1.2.1.1). Below you can see a list of classes contained in this module. A detailed description of the classes can be found further down.

- FuzzyOperator (cf. Section 1.2.4.2)
- DeFuzzyOperator (cf. Section 1.2.4.3)
- ConcatOperator (cf. Section 1.2.4.4)
- TNorm (cf. Section 1.2.4.5)
- TCoNorm (cf. Section 1.2.4.6)
- CompOp (cf. Section 1.2.4.7)
- DefOp (cf. Section 1.2.4.8)
- OperatorSet (cf. Section 1.2.4.9)

1.2.4.1 global functions

- join_mv_with_importance (cf. Section 1.2.4.10)
- aggregate (cf. Section 1.2.4.11)

- certify (cf. Section 1.2.4.12)
- inference (cf. Section 1.2.4.13)
- accumulate (cf. Section 1.2.4.14)

1.2.4.2 FuzzyOperator

This is a simple class that represents the type of a FuzzyOperator (cf. Section 1.2.4.2), which is used for the connection of terms, generally the operators can be interpreted as 'and' and 'or' relations. Of course, the more fuzzy an operator is, the more the otherwise clear difference between 'and' and 'or' begins to blur. The only attribute of the class is the operator type *oper*, which is of the enumeration type *fuzzy_operator*. The enumeration type declaration at the beginning of the file flipoper.h / flipoper.cpp (cf. Section 1.2.4) shows the operators currently supported by FLIP++.

The default constructor initializes an object with the attribute value *no_fop*. The alternative constructor takes the type of operator as input. The method *show* writes the type of the operator to the standard output device. The methods *read* and *write* do the same as *show* using files instead of standard output. The simple methods *get_oper* and *set_oper* are used to set a new operator and get the current operator type.

1.2.4.3 DeFuzzyOperator

This class represents the type of a DeFuzzyOperator (cf. Section 1.2.4.3), which is used to retrieve a defuzzified, i.e. a crisp value (cf. Section 1.2.1.1) from the fuzzy result of the inference process. The only attribute of the class is the operator type *oper*, which is of the enumeration type *defuzzification_operator*. The enumeration type declaration at the beginning of the file flipoper.h / flipoper.cpp (cf. Section 1.2.4) shows the operators currently supported by FLIP++.

The default constructor initializes the object with the attribute value *no_dfop*. The alternative constructor takes the type of operator as input. The method *show* writes the type of the operator to the standard output device. The methods *read* and *write* do the same as *show* using files instead of standard output. The methods *get_oper* and *set_oper* are used to set and get the type of the operator.

1.2.4.4 ConcatOperator

This class is the representation of a ConcatOperator (cf. Section 1.2.4.4), which is used to concatenate several Terms (cf. Section 1.2.5.1). The only attribute of the class is the operator type *oper*, which is of the enumeration type *concat_operator*. The enumeration type declaration at the beginning of the flipoper.h / flipoper.cpp (cf. Section 1.2.4) file shows the operators currently supported by FLIP.

The default constructor initializes the object with the attribute value *no_cop*. The alternative constructor takes the type of operator as input. The method *show* writes the type of the operator to standard output. The methods *read* and *write* do the same as *show* using files instead of standard output.

1.2.4.5 TNorm

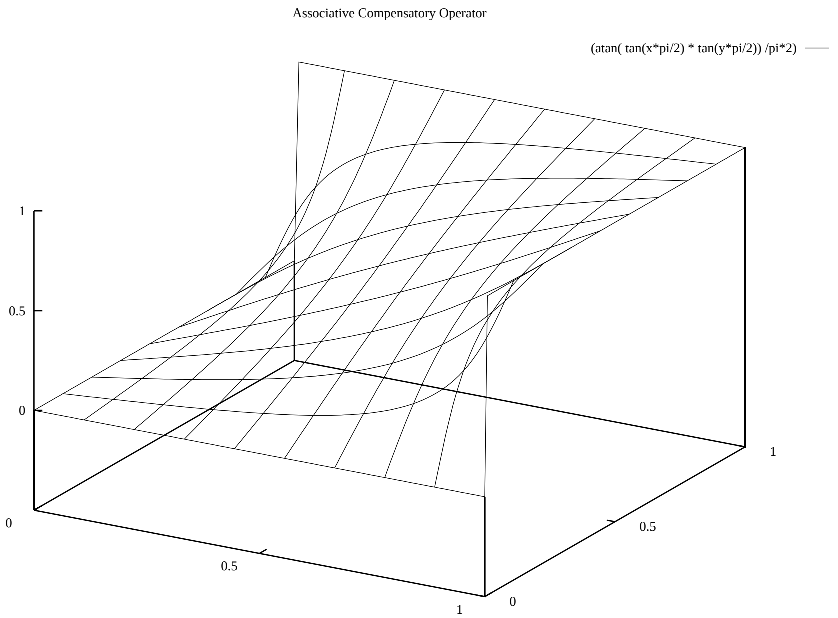
This class has no attributes and provides methods to apply a minimum operator by using a T-Norm calculation. The method *min* with a reference to an FList (cf. Section 1.2.3.2) object as parameter simply calculates the minimum of the float values contained in the FList . The method *min* with the parameters *overall match value* and *certification value* calculates the minimum of these two values. The method *min* with the parameters *overall certified match value* and a reference to a PlimFuzzySet (cf. Section 1.2.3.6) object calculates the minimum of the *overall certified match value* and the *match value* of the PlimFuzzySet object, in case the PlimFuzzySet object does not already have an instantiated membership function (cf. Section 1.2.1.1), the result is the *overall certified match value.* Hereafter the instantiated membership function of the PlimFuzzySet object is calculated by setting the *match value* to the result of the previous computation and calling the *build_imsf* method of the PlimFuzzySet object.

1.2.4.6 TCoNorm

This class has no attributes and provides methods to apply a maximum operator by using a T-Conorm calculation. The method *max* with a reference to an FList (cf. Section 1.2.3.2) object as a parameter simply calculates the maximum of the float values contained in the FList . The method *max* with the parameters *overall match value* and *certification value* calculates the maximum of these two values. The method *max* with the parameters *overall, certified, match, value,* and a reference to PlimFuzzySet (cf. Section 1.2.3.6) object calculates a new *match value* and then instantiates a membership function (cf. Section 1.2.1.1) for the PlimFuzzySet (cf. Section 1.2.3.6) object, referenced by applying the T-CoNorm operator.

1.2.4.7 CompOp

This class is a set of compensatory operators. Similar to the TNorm (cf. Section 1.2.4.5) and TCoNorm (cf. Section 1.2.4.6) each of the operators comes with three variants of a method, the first one with a reference to an FList (cf. Section 1.2.3.2) object as a parameter, the second one with the parameters *overall, match, value* and *certification value*, and the third variant has the *overall certified match value* and a reference to a PlimFuzzySet (cf. Section 1.2.3.6) object as parameters. The *ass_comp_and* and the *ass_comp_or* operators are based on the tangent function, and provide associative and compensatory operators. In order to get a grip on the operator's functionality have a look at the picture below.



2

Figure 1.2: Associative compensatory operator$f(x) = \frac{arctan(tan(x\frac{\pi}{2})tan(y\frac{\pi}{2}))}{2\pi}$

The *fuzzy_and* and the *fuzzy_or* operators are implemented by the use of the so-called *Gamma* operator, therefore it is necessary to add a further argument to the parameters mentioned above, namely the *gamma* value. This value provides the opportunity to modify the extent of compensation and determine how fuzzy the operator should be.

1.2.4.8 DefOp

This class has no attributes and provides methods to apply the defuzzification (cf. Section 1.2.1.1) operator. The procedure used to achieve this is the *center of gravity* method. The method *cog* therefore generates the cover (cf. Section 1.2.1.1) of the CombinedVariable (cf. Section 1.2.3.8) referenced in the parameter and then integrates the cover via the method *integrate*. This method is reponsible for the calculation of the *center of gravity* and returns a float value, that is the crisp value (cf. Section 1.2.1.1) - which is the defuzzified value of the linguistic variable (cf. Section 1.2.1.1).

1.2.4.9 OperatorSet

This class aggregates the operators and methods necessary for FLIP++ to do calculations with a linguistic variable (cf. Section 1.2.1.1). There are four important steps to achieve results from a rulebase: These are *aggregation*, *certification*, *inference* and *accumulation*. According to these four steps there are four functions aggregate (cf. Section 1.2.4.11), certify (cf. Section 1.2.4.12), inference (cf. Section 1.2.4.13) and accumulate (cf. Section 1.2.4.14). The attributes of the OperatorSet (cf. Section 1.2.4.9) are the FuzzyOperators (cf. Section 1.2.4.2) - *and_operator*, *or_operator*, *certainty_operator*, *inference_operator*, *accumulate_operator*, *importance_operator* -, the DeFuzzyOperator (cf. Section 1.2.4.3) *defuzzy_operator*, and the integers *accuracy* and *gamma*.

The default constructor initializes all the operators of the object and sets the *accuracy* and *gamma* to a default value. The method *copy* relocates all the parameters from an OperatorSet (cf. Section 1.2.4.9) object into the current one. The method *show* writes all the parameters of the OperatorSet object to the standard output device. The methods *read* and *write* handle input/output to files. Moreover, there are simple methods to set and get the values of all the attributes of the OperatorSet.

1.2.4.10 join_mv_with_importance

This function combines a *match value* and an importance (cf. Section 1.2.1.1) with the *importance operator* of an OperatorSet (cf. Section 1.2.4.9) object.

1.2.4.11 aggregate

This function aggregates the *match values* of all instantiated membership functions (cf. Section 1.2.1.1) stored in an FList (cf. Section 1.2.3.2) object to an *overall match value* by applying the *concat operator* type specified by the parameter through the operators of the referenced OperatorSet (cf. Section 1.2.4.9) object.

1.2.4.12 certify

This function calculates the *overall certified match value* by applying the *certainty operator* of the OperatorSet object referenced to the *overall match value* and the *certainty value*.

1.2.4.13 inference

This function calculates a new *match value* by applying the *inference operator* to the *overall certified match value* and the old *match value* of the PlimFuzzySet (cf. Section 1.2.3.6) referenced.

1.2.4.14 accumulate

This function accumulates the value of a linguistic variable (cf. Section 1.2.1.1) by calculating its crisp value (cf. Section 1.2.1.1) through the defuzzification (cf. Section 1.2.1.1) process (*center of gravity*).

1.2.5 fliprule.h / fliprule.cpp

- Term (cf. Section 1.2.5.1)
- TList (cf. Section 1.2.5.2)
- Rule (cf. Section 1.2.5.3)
- RuleBlock (cf. Section 1.2.5.4)
- RuleSet (cf. Section 1.2.5.5)

1.2.5.1 Term

This class represents a term consisting of a linguistic variable (cf. Section 1.2.1.1) and an associated linguistic term (cf. Section 1.2.1.1), for example, 'TEMPERATURE is HIGH'. This class also represents the already familiar list concept of FLIP++, it is the element class that is aggregated in a linear list in the TList (cf. Section 1.2.5.2) class. The attributes of the object are a reference to a CombinedVariable (cf. Section 1.2.3.8) object *cv*, a string *lt* for the linguistic term (of type Name (cf. Section 1.2.2.1)), and a reference to another Term (cf. Section 1.2.5.1) which is used in the TList class.

The default constructor initializes an empty object referencing no linguistic variable and a default linguistic term. The destructor empties the object by dissolving any references to a CombinedVariable (cf. Section 1.2.3.8) object and to another Term (cf. Section 1.2.5.1). The method *show* writes the term in a format identical to the above example to the standard output device. The same is done by the methods *read* and *write*, except they read from and write to files, the method *read* also checks whether the read term is formed with a valid linguistic variable (cf. Section 1.2.1.1) given a list of linguistic variables (cf. Section 1.2.1.1). The method *copy* copies an existing term to the current term, i.e. its linguistic variable reference and the linguistic term (cf. Section 1.2.1.1). Furthermore, there are simple methods to get and set the attributes of the class.

1.2.5.2 TList

This class is a linear list of Term (cf. Section 1.2.5.1) objects. The attributes of the list are the references to the *first* and *last* Term object in the list, while *number* indicates the number of elements contained in the list.

The default constructor creates an object as an empty list. The destructor empties the list by deleting all its elements according to a clean memory management. The method *show* without parameters displays all terms contained in the list via the standard output device. Alternatively, the *show* method has a ConcatOperator (cf. Section 1.2.4.4) as parameter and concats all terms of the list with this parameter for output. The method *write* does the same as the latter *show* method, but instead of standard output it uses a file for output. The *read* method reads a complete list of terms from a file. The *add* method adds a new term to the list by taking its linguistic variable (cf. Section 1.2.1.1) and linguistic term (cf. Section 1.2.1.1) as parameters. The *add* method without parameters adds an empty (default) term to the list. The method *find* accesses a term by its index in the list. Furthermore it is possible to retrieve the index of an element containing the linguistic variable specified by the parameter. As an alternative to the former method the array operator [] can be used. Apart from these *find* methods, there are also two other *find* methods *find_cv* and *find_lt* that return references to the CombinedVariable (cf. Section 1.2.3.8) and linguistic term (cf. Section 1.2.1.1) of the element specified by its index respectively. The *copy* method copies the element of an existing TList (cf. Section 1.2.5.2) object to the current object. The method *remove* removes the element specified by its index, whereas *remove_all* deletes all elements of the list. Again, there are simple methods to access the attributes of the class and to set the linguistic variable (cf. Section 1.2.1.1) and linguistic term of an element specified by its index.

1.2.5.3 Rule

This class represents a rule consisting of a list of terms on the left side and a term on the right side. The left side terms are aggregated by a concat operator, additionally, every rule has a certainty attached to it. Again, several rules are combined in a list of rules, see the class RuleBlock (cf. Section 1.2.5.4) for further details. Hence it is also necessary to have a reference to another rule. The format of a rule is: IF Term-List (cf. Section 1.2.5.2) THEN Term (cf. Section 1.2.5.1) WITH CERTAINTY c. Among the attributes are the left side terms stored in a TList (cf. Section 1.2.5.2) object called *ls_t_list*, the right side *y*, the *certainty* (a float value), the ConcatOperator (cf. Section 1.2.4.4) *aggregate_operator* for aggregation of the left side terms, and *next* - the reference to another rule.

The constructor initializes an empty object. The destructor deletes all terms referenced by the rule and initializes the other attributes. The method *show* writes the rule in the format introduced above to the standard output device. In the same way the method *write* handles output to files, and the method *read* input from files. The *copy* method copies an existing rule to the current rule object. Moreover, simple access methods render it possible to retrieve or manipulate the parameters of a rule, in particular, its left side linguistic variables (cf. Section 1.2.1.1) and linguistic terms (cf. Section 1.2.1.1), the right side linguistic variable and linguistic term, the *certainty*, the number of left side terms, and the *concat operator*. Furthermore, the attributes of the rule can be retrieved and set.

1.2.5.4 RuleBlock

This class is the aggregation of several rules to a ruleblock, which is normally used to put all rules with the same right side linguistic variable (cf. Section 1.2.1.1) together. In the next class RuleSet (cf. Section 1.2.5.5) various RuleBlock (cf. Section 1.2.5.4) objects are combined in a linear list. The attributes of the class are the references *first* and *last* to the first and last rule of the list with *number* indicating the number of rules contained in the RuleBlock, and *next* - referencing another RuleBlock.

The constructor creates an object containing an empty list. The destructor deletes all rules contained in the list according to a clean memory management. The method *show* displays all rules of the RuleBlock on standard output by calling the underlying *show* method of each rule. The methods *read* and *write* do the same, except they read/write to files instead of standard output. The method *add* has two variants: when it comes without a parameter it adds an empty rule to the RuleBlock (cf. Section 1.2.5.4) while with a parameter it adds the rule specified in the parameter to the RuleBlock object. The *find* method accesses an element of the list by its index, whereas the *find_cv* method returns a reference to the linguistic variable (cf. Section 1.2.1.1) of the right side term of the rule specified by its index in the list, and the *find_lt* method returns a reference to the linguistic term (cf. Section 1.2.1.1) of the right side term of the rule specified by its index in the list. Alternatively the array operator [] can be used. The *copy* method copies the contents of an existing RuleBlock object to the curent RuleBlock object. The method *remove* deletes an existing element from the list of rules. *Remove_all* deletes all elements of the list. Furthermore, there are simple methods to get and set all the significant parameters of a rule described above in the rule class. Also, the attributes of the class can be set and retrieved by simple methods.

1.2.5.5 RuleSet

This class finally aggregrates all the different RuleBlocks (cf. Section 1.2.5.4) into one linear list, called RuleSet (cf. Section 1.2.5.5), which includes additional parameters. Among them are a list of linguistic variables (cf. Section 1.2.1.1) contained in the RuleSet, a list of ParameterSets (cf. Section 1.2.2.6) and an OperatorSet (cf. Section 1.2.4.9), so this class combines all of the functionality contained in FLIP++. The attributes of this class - apart from the inherited ones - are *rs_name*, the name of the RuleSet of Name (cf. Section 1.2.2.1) type, *ps_list*, the linear list of parameter sets, *cv_list*, a linguistic variable (cf. Section 1.2.1.1) list, *operator_set*, the aggregation of all operators needed, and the attributes for the linear list of RuleSet *first*, *last*, and *number*.

The default constructor initializes a RuleSet object with a default (empty) PSList (cf. Section 1.2.2.7) object, a default CVList (cf. Section 1.2.3.9) object, a default OperatorSet, and no RuleBlocks (cf. Section 1.2.5.4). The alternative constructor takes a PSList, CVList, and OperatorSet object reference as input and initializes an empty (without RuleBlocks (cf. Section 1.2.5.4)) RuleSet (cf. Section 1.2.5.5) accordingly. The destructor deletes all RuleBlocks (cf. Section 1.2.5.4), the PSList (cf. Section 1.2.2.7), the CVList (cf. Section 1.2.3.9) and the OperatorSet (cf. Section 1.2.4.9) of the RuleSet (cf. Section 1.2.5.5) according to a clean memory management. The method *show* displays all elements of the RuleBlock list on standard output. The methods *read* and *write* do the same as *show* except they read from/write to files instead of standard output. The *add* method adds a copy of an existing RuleBlock to the RuleSet. The *add_all_RuleBlocks* method adds a copy of a list of RuleBlocks, which of course is a RuleSet of its own, to the current RuleSet object. The method *add_Rule* adds a single rule consisting of a left-side term list, a right-side term, a concat_operator, and a certainty value to the RuleBlock of the RuleSet containing rules with the same right side linguistic variable (cf. Section 1.2.1.1) as the rule to be added, or, if such a RuleBlock does not exist, to a new RuleBlock. Moreover, it is necessary to check whether this rule puts a new linguistic variable (cf. Section 1.2.1.1) into the RuleSet (cf. Section 1.2.5.5) which makes it necessary to update the list of linguistic variables (cf. Section 1.2.1.1). It is to be noted that a rule must not contain its right side linguistic variable in its left side. Furthermore, the method *add_BasicRule* makes it possible to add a BasicRule consisting of one left-side term (linguistic variable (cf. Section 1.2.1.1) plus linguistic term (cf. Section 1.2.1.1)), one right-side term (linguistic variable (cf. Section 1.2.1.1) plus linguistic term (cf. Section 1.2.1.1)), a nd a certainty to the RuleSet (cf. Section 1.2.5.5). The method *add_ComplexRule* is similar to the *add_BasicRule* method, the difference is that it adds a more complex rule consisting of two left-side terms instead of one to the RuleSet (cf. Section 1.2.5.5). Both *add_BasicRule* and *add_ComplexRule* use the generic method *add_Rule* to add the rule specified by the parameters to the RuleSet. The *find* method accesses a RuleBlock of the RuleSet by its index. As a modification to the *find* method the *find_cv* method returns a reference to the linguistic variable (cf. Section 1.2.1.1) by which the rules were put together to a RuleBlock, it takes also the RuleBlock's index as an argument. An alternative to the former *find* method, the array operator [] can be used. The method *search* returns the index of the RuleBlock in the RuleSet that contains the linguistic variable specified in the parameter in its right side term(s). The method *remove* deletes the RuleBlock (cf. Section 1.2.5.4) specified by its index. Alternatively, the method *remove* can be called with a linguistic variable as parameter which removes the RuleBlock containing this linguistic variable among its right side terms. The method *remove_each_app_of_cv_in_rs_and_cvl* removes all appearance of a linguistic variable (cf. Section 1.2.1.1) in the RuleSet (cf. Section 1.2.5.5), i.e. each rule where the linguistic variable appears in the left side and the RuleBlock (cf. Section 1.2.5.4) that aggregates all rules containing the linguistic variable in the right side, and CVList (cf. Section 1.2.3.9). The method *remove_all* deletes all RuleBlocks contained in the RuleSet.

The method *evaluate* starts the inference process for a specific linguistic variable by calling the *internal_evaluate* method. The process starts with checking whether all the left side linguistic variables in the RuleBlock of the linguistic variable to be evaluated are already instantiated or in turn appear on the left side of a rule, which makes it necessary to evaluate them first. Then each term on the left side of a rule is weighted with an importance (cf. Section 1.2.1.1) through the importance operator, after that all *match values* of the left side terms of a rule are aggregated to an *overall match value* by the use of the aggregation operator. This *overall match value* is linked with the *certainty* of the rule to obtain the *overall certified match value*. Finally, the *overall certified match value* and the old *match value* of the linguistic term (cf. Section 1.2.1.1) (Fuzzy Set) of the right side term are linked with the inference operator to produce the new match value of the right side linguistic term (cf. Section 1.2.1.1) of the linguistic variable (cf. Section 1.2.1.1) to be evaluated. When the inference process has finished the crisp value (cf. Section 1.2.1.1) of the linguistic variable (cf. Section 1.2.1.1) is calculated through the accumulate (cf. Section 1.2.4.14) method (see flipoper.h / flipoper.cpp (cf. Section 1.2.4)) and attached to the crisp value attribute of the linguistic variable. Apart from these complex methods, there are simple methods that enable setting and getting the values of the attributes, and also of those inherited from the parent classes.

1.3 Sample code use

This is an illustration of how to apply the FLIP++ libraries (cf. Section 1.2, page 4). It should provide a knowledge engineer or programmer with the basics of the FLIP++ functionality when intending to include the libraries as part of a more specific system.

Although this example is quite simple, but it should be enough for a demonstration of the library. Suppose we have a heating system that is controlled by an indoor and outdoor temperature sensor. This gives us an idea of what to use for linguistic variables (cf. Section 1.2.1.1): *heating*, *indoor* and *outdoor*. Furthermore, we need a representation of the specific values that the linguistic variables can have. As we are using a fuzzy system it is rather straightforward to define these values called linguistic terms (cf. Section 1.2.1.1).

```
//
//     -----------------------------------
//      F L I P D E M O . C P P
//     -----------------------------------
//

#include "flip.h"
```

```
main() {
// *** declare and allocate io-streams

ifstream *input = new ifstream;
ofstream *output = new ofstream;

// *** declare errorstring

char static error[100]=" ";
```

A distinct name and colour must be defined for each linguistic term (cf. Section 1.2.1.1) of the linguistic variables (cf. Section 1.2.1.1) as shown in the following code sections. Therefore, a CList (cf. Section 1.2.2.4) class instance is used for *outdoor*, *heating* and *indoor* with the parameter indicating the number of linguistic terms.

```
CList *cl_outdoor  = new CList(4);
CList *cl_indoor   = new CList(3);
CList *cl_heating  = new CList(3);
```

Now, the same procedure is followed with the NList (cf. Section 1.2.2.2) class in order to attach names to the linguistic terms.

```
NList *nl_outdoor = new NList;
NList *nl_indoor  = new NList;
NList *nl_heating = new NList;
```

Having the objects instantiated, we can now assign the linguistic terms (cf. Section 1.2.1.1) to the linguistic variables (cf. Section 1.2.1.1) by putting them into the respective NList (cf. Section 1.2.2.2) objects.

```
nl_outdoor->add("icy");
nl_outdoor->add("cool");
nl_outdoor->add("mild");
nl_outdoor->add("hot");
nl_indoor->add("cold");
nl_indoor->add("medium");
nl_indoor->add("warm");
nl_heating->add("no");
nl_heating->add("half");
nl_heating->add("full");
```

So far, we have dealt with the example only on a linguistic level. However, in order to get any results out of the system through the evaluation process we have to specify some parameters. Hence we introduce another object for each linguistic variable (cf. Section 1.2.1.1): the ParameterSet (cf. Section 1.2.2.6). It contains the values of the support range (cf. Section 1.2.1.1), median (cf. Section 1.2.1.1), dilatation (cf. Section 1.2.1.1), distribution (cf. Section 1.2.1.1) and importance (cf. Section 1.2.1.1) of each linguistic variable (cf. Section 1.2.1.1). Furthermore it includes the NList (cf. Section 1.2.2.2) and CList (cf. Section 1.2.2.4) objects defined above.

```
ParameterSet *ps_outdoor,*ps_indoor,*ps_heating;
ps_outdoor = new ParameterSet(
"ps_outdoor", -50, 40, 0, 0.9, 0.7, 1, cl_outdoor, nl_outdoor);
ps_indoor  = new ParameterSet(
"ps_indoor", 0, 40, 15, 0.9, 0.7, 0.9, cl_indoor, nl_indoor);
ps_heating = new ParameterSet(
"ps_heating", 0, 100, 50, 0.9, 0.8, 1, cl_heating, nl_heating);
```

Thus far, we have only defined the parameters of the linguistic variables (cf. Section 1.2.1.1). But there are still no instances of the linguistic variables. The next step is to create a CombinedVariable (cf. Section 1.2.3.8) (=linguistic variable (cf. Section 1.2.1.1)) object for each linguistic variable. The ParameterSet (cf. Section 1.2.2.6) is included in this variable.

```
CombinedVariable *cv_outdoor =
new CombinedVariable(ps_outdoor,"outdoor");
CombinedVariable *cv_indoor =
new CombinedVariable(ps_indoor,"indoor");
CombinedVariable *cv_heating =
new CombinedVariable(ps_heating,"heating");
```

As the *outdoor* and *indoor* temperatures are received from a sensor, they are input variables. Therefore, it is necessary to attach values to them, which starts the matching process. The evaluation process then treats them as matched variables.

```
cv_outdoor->set_value(7);
cv_indoor->set_value(17);
```

With the linguistic variables defined, the next step is to define a set of rules that describes the relationship between the linguistic variables (cf. Section 1.2.1.1). We create an instance of the RuleSet (cf. Section 1.2.5.5) class and specify the inference operator and attach the linguistic variables (cf. Section 1.2.1.1) to it.

```
RuleSet *rule_set = new RuleSet();
rule_set->get_operator_set()->set_inference_operator(maximum);
rule_set->get_ps_list()->add(ps_outdoor);
rule_set->get_ps_list()->add(ps_indoor);
rule_set->get_ps_list()->add(ps_heating);
```

The rules are simply added to the RuleSet (cf. Section 1.2.5.5) with the *add_ComplexRule* method.

```
rule_set->add_ComplexRule(
cv_outdoor, "icy",  cv_indoor, "cold",
cv_heating, "full", AND, 1);
rule_set->add_ComplexRule(
cv_outdoor, "icy",  cv_indoor, "medium",
cv_heating, "full", AND, 0.9);
rule_set->add_ComplexRule(
cv_outdoor, "icy",  cv_indoor, "warm",
cv_heating, "half", AND, 1);
rule_set->add_ComplexRule(
cv_outdoor, "cool",  cv_indoor, "cold",
cv_heating, "full", AND, 0.8);
rule_set->add_ComplexRule(
cv_outdoor, "cool",  cv_indoor, "medium",
cv_heating, "half", AND, 1);
rule_set->add_ComplexRule(
cv_outdoor, "cool",  cv_indoor, "warm",
cv_heating, "half", AND, 0.7);
rule_set->add_ComplexRule(
cv_outdoor, "mild",  cv_indoor, "cold",
cv_heating, "half", AND, 1);
rule_set->add_ComplexRule(
cv_outdoor, "mild",  cv_indoor, "medium",
cv_heating, "half", AND, 0.8);
```

Now we can start the evaluation process of the output variable *heating*

```
rule_set->evaluate(cv_heating);
if (strcmp(error," ")!=0) cout << endl << " ### error: " << error << endl;
```

In order to show the modularity of FLIP++ we write the created RuleSet to a file, then we create a new RuleSet (cf. Section 1.2.5.5) with an additional linguistic variable (cf. Section 1.2.1.1) for the *fuel_flow* and reload the old RuleSet. Finally, the two RuleSets are joined together and put into a more complicated RuleSet.

Below the RuleSet is written to a file.

```
output->open("fliprule.dat");
rule_set->write(output);
output->close();
```

Here a new RuleSet is created.

```
RuleSet *rule_set2 = new RuleSet();
```

As we want to use another linguistic variable (cf. Section 1.2.1.1) for the *fuel_flow*, we must first build a CombinedVariable (cf. Section 1.2.3.8) object and all part objects that are required. This follows the same scheme shown at the beginning.

```
CList *cl_fuel_flow  = new CList(3);
NList *nl_fuel_flow = new NList;
nl_fuel_flow->add("zero");
nl_fuel_flow->add("low");
nl_fuel_flow->add("high");
ParameterSet *ps_fuel_flow = new ParameterSet("ps_fuel_flow",
0, 50, 20, 0.9, 0.9, 1, cl_fuel_flow, nl_fuel_flow);
rule_set2->get_ps_list()->add(ps_fuel_flow);
CombinedVariable *cv_fuel_flow =
new CombinedVariable(ps_fuel_flow,"fuel_flow");
```

Below, we add new rules connecting the new variable to the existing ones. This is also the reason why we have not deleted the old RuleSet (cf. Section 1.2.5.5) yet as we need it for the rule definitions.

```
rule_set2->add_BasicRule( cv_heating, "no",   cv_fuel_flow, "zero", 1);
rule_set2->add_BasicRule( cv_heating, "half", cv_fuel_flow, "low",  1);
rule_set2->add_BasicRule( cv_heating, "full", cv_fuel_flow, "high", 1);
```

Now, we put the two RuleSets together. This is done by loading the old RuleSet and then adding it to the extended one.

```
delete rule_set;
RuleSet *rule_set_temp = new RuleSet();
input->open("fliprule.dat");
rule_set_temp->read(input);
input->close();
rule_set2->get_ps_list()->add(rule_set_temp->get_ps_list());
rule_set2->get_cv_list()->add(rule_set_temp->get_cv_list());
rule_set2->add_all_RuleBlocks(rule_set_temp);
delete rule_set_temp;
```

The evaluation procedure for the *fuel_flow* variable can now be started as usual.

```
rule_set2->evaluate(cv_fuel_flow);
};
```

1.4 FLIP++ User Interface

In the following we will use a simple example, which is actually identical to the one used in the FLIP++ Sample Code (cf. Section 1.3, page 29) which is a part of DocuFLIP++. Although very simple and not too realistic it should give you a brief impression of how to use the FLIP++ user interface.

Suppose we have a heating system that is controlled by an indoor and outdoor temperature sensor. This gives us an idea of what to use for linguistic variables (cf. Section 1.2.1.1): heating, indoor and outdoor. Furthermore, we need a representation of the specific values the linguistic variables can have. As we use a fuzzy system it is rather straightforward to define these values called linguistic terms (cf. Section 1.2.1.1). Apart from the linguistic level we also need some parameters to tell the system how to transform the linguistic level into a sound mathematical representation. Finally, we can add some rules and thereby provide the system with information about the interrelationship of the linguistic variables, i.e. in terms of our example the relationship between the outdoor and indoor temperature on the one side and the level of heating on the other side. After the values of the input variables have been set, an evaluation can be done.

1.4.1 Main application window

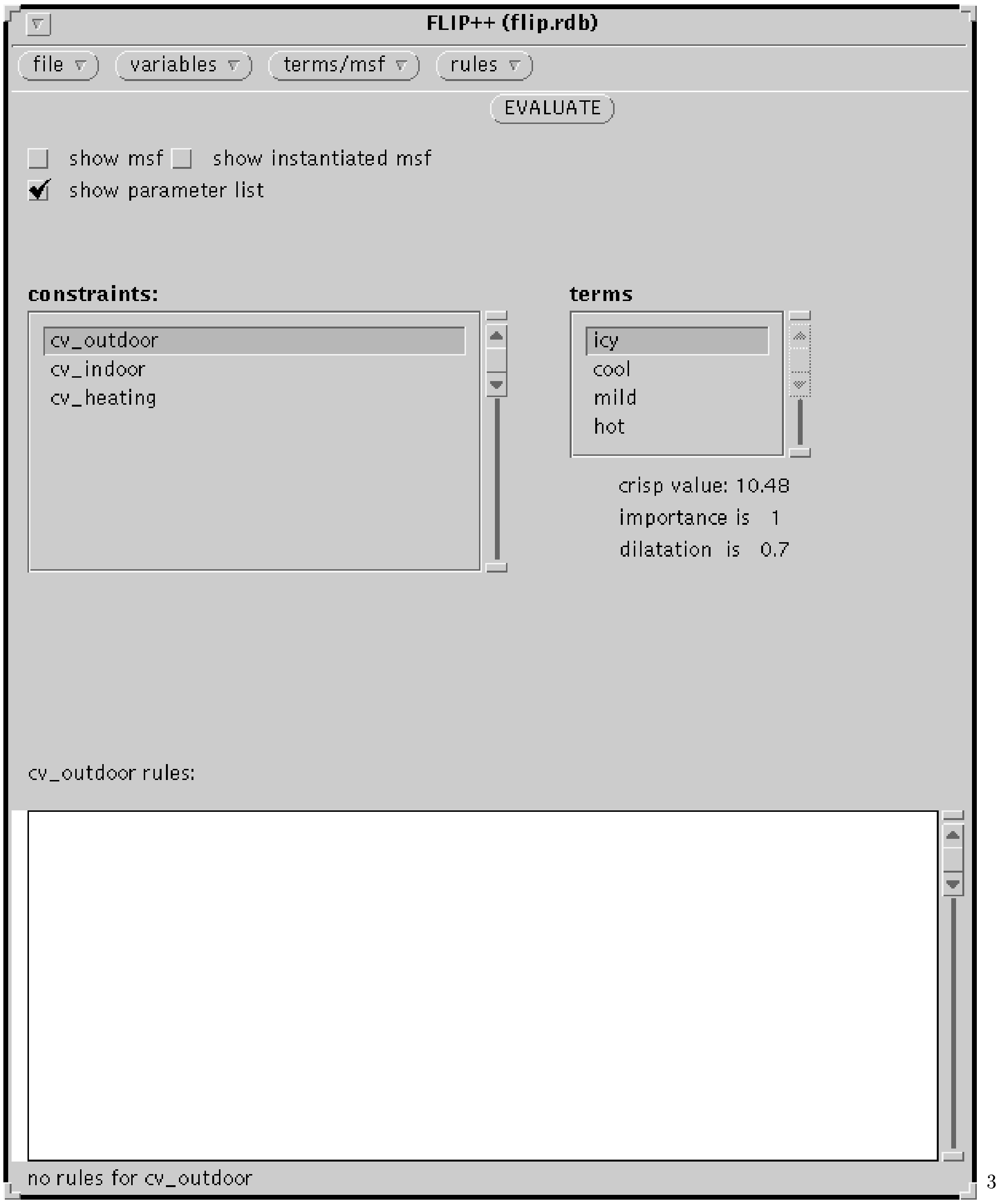


Figure 1.3: FLIP++ Main Application Window

Initially, we find ourselves in the main application window. It encompasses all the features of FLIP++: the user can define linguistic variables (cf. Section 1.2.1.1), attach a set of parameters to them, add rules to put knowledge into the system, and evaluate a well defined sytem of rules, linguistic variables and parameters. Another important feature is the definition of the membership functions (cf. Section 1.2.1.1) and their graphical representation.

1.4.2 How to define a ParameterSet

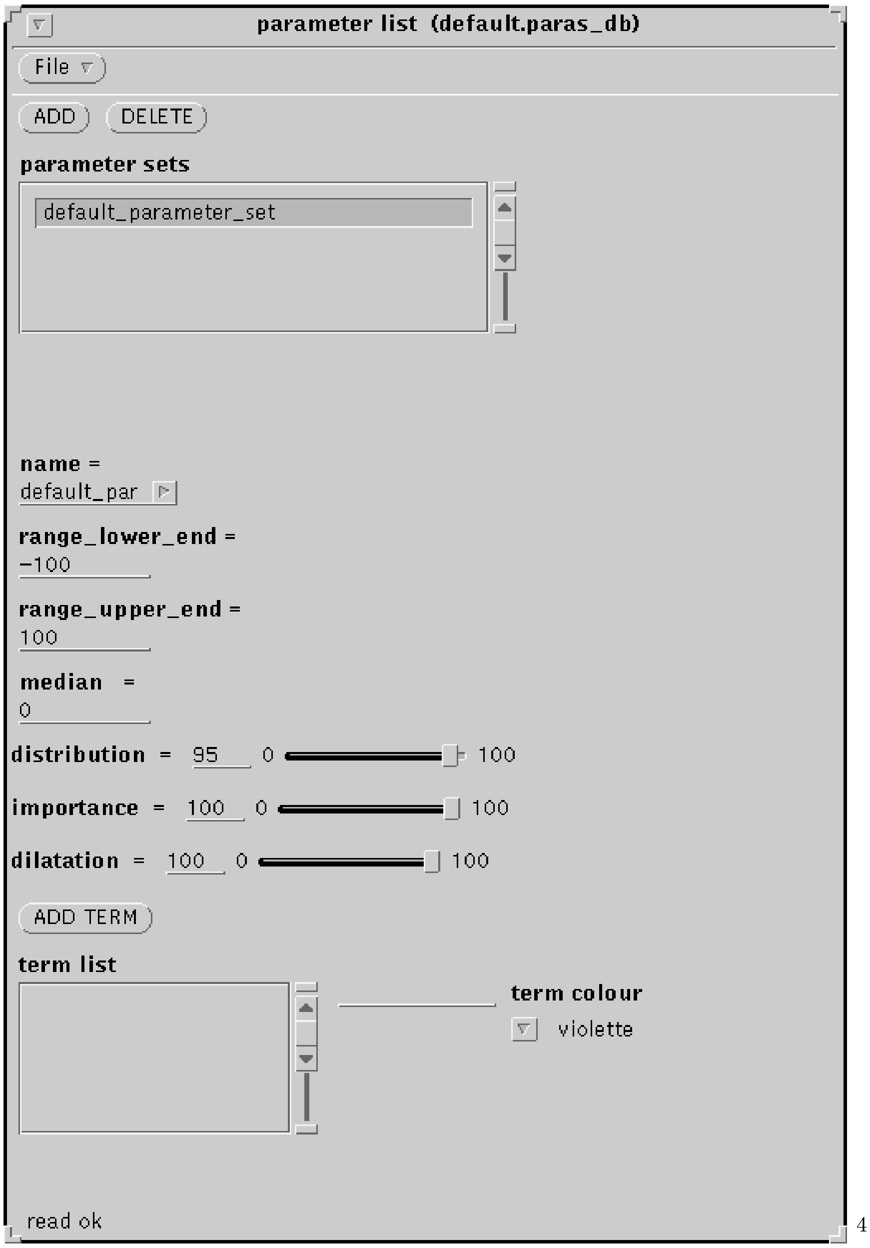


Figure 1.4: How to define a ParameterSet

Before we start defining our linguistic variable we must define the ParameterSets (cf. Section 1.2.2.6) which are fundamental for every linguistic variable (cf. Section 1.2.1.1). This can be done by clicking the *show parameter list* button in the main application window. A new window *parameter list* is shown on the screen that enables the user to define various parameters concerning a linguistic variable. It shows all parameter sets available. They can be selected by clicking the mouse button. With the buttons *ADD* and *DELETE* it is possible to add or remove ParameterSets. Furthermore, it is possible to read ParameterSets from and write to files by clicking the *File* Button. When writing to a file, all ParameterSets currently displayed and hence stored in memory are written to the file. Among the individual parameters of a ParameterSet are the name, the range with its lower and upper end, the median, the distribution, the importance, and the dilatation. This is exactly the same as in the FLIP++ Sample Code (cf. Section 1.3, page 29).

A ParameterSet also refers to a specific set of linguistic terms (cf. Section 1.2.1.1), which has to be defined. These terms are displayed in the *term list*. New terms can be added to the list by clicking *ADD TERM*. Moreover, the *term colour* used in the graphical representation can be modified.

Coming back to our example, we learn from the short description above that we need three parameter sets corresponding to the three linguistic variables (cf. Section 1.2.1.1). We start with the *ps_outdoor* ParameterSet (cf. Section 1.2.2.6). First, we use *ADD* to enter the new ParameterSet. A new window *new set of parameter* comes up asking for all the parameters, The number of linguistic terms (cf. Section 1.2.1.1) specified in the item *how much terms* determines the entries in the *term list* of the *parameter list* window. After entering the same values for the parameters as in the FLIP++ Sample Code (cf. Section 1.3, page 29) and clicking the *OK* Button, the programm returns to the *parameter list* window. Here it is possible to rename the linguistic terms as they are given some default values which might not be appropriate for a specific example. In our example the linguistic terms are again identical to the ones used in the FLIP++ Sample Code (cf. Section 1.3, page 29).

1.4.3 How to define a linguistic variable

Now we return to the main application window. The menu *variables* offers three items that enable us to add, edit and delete a linguistic variable (cf. Section 1.2.1.1). As we want to use three linguistic variables in our example, we have to apply the *ADD variable* item three times. Each time a small request box is shown asking for the name of the linguistic variables and the ParameterSet (cf. Section 1.2.2.6) that should be attached to the variable.

We start with *cv_outddor* and attach *ps_outdoor* to it, and so on. Editing and deleting a variable are straightforward operations. After a new linguistic variables has been created, it can be found in the *constraints* list.

1.4.4 Membership functions

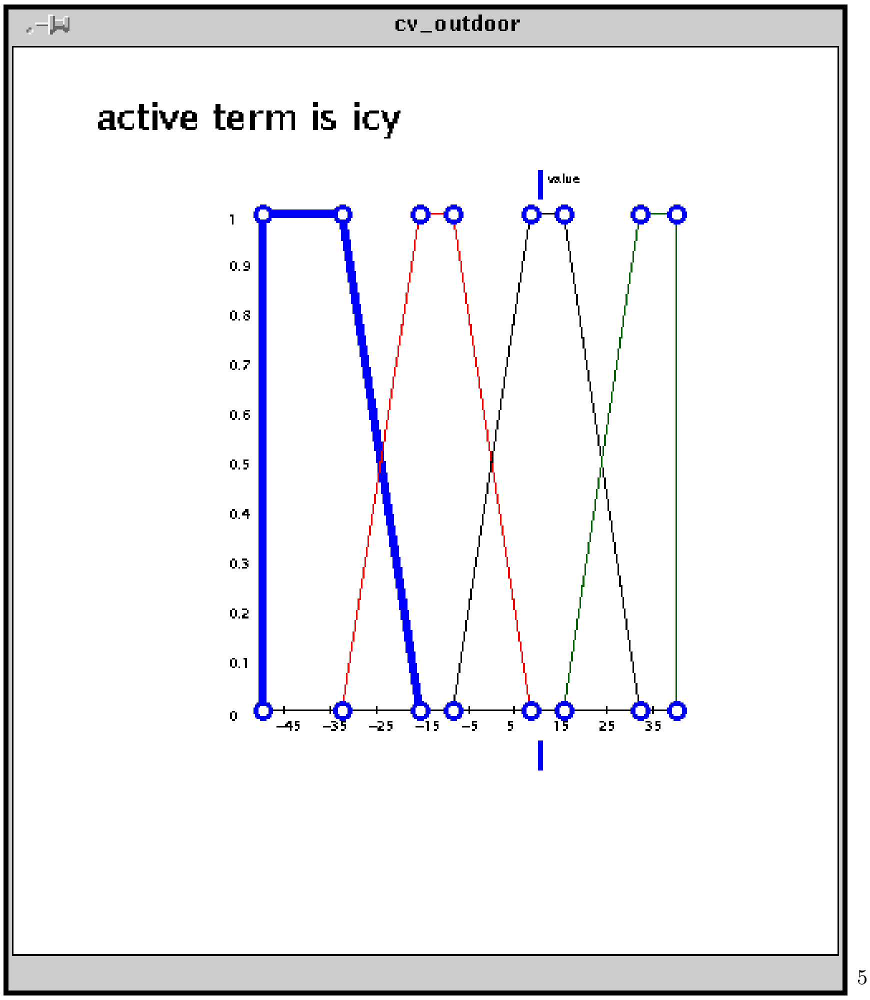


Figure 1.5: Membership functions

The menu *terms/msf* provides features to modifiy the shape of the membership function (cf. Section 1.2.1.1) of the linguistic variables (cf. Section 1.2.1.1) selected in the *constraints* list. When clicking the *show msf* button you can see the shape of the default membership function (cf. Section 1.2.1.1) for the currently selected linguistic variables (cf. Section 1.2.1.1) in the *constraints* list. The membership function (cf. Section 1.2.1.1) of the currently selected linguistic terms (cf. Section 1.2.1.1) of that variable is drawn in bold lines. Now, it is possible to modifiy the membership function by dragging the points with the mouse pointer, adding a point afer selecting the menu item *ADD a Node to msf*, and deleting a point after selecting the menu item *DEL a Node from msf.* Furthermore, the menu item *SET value of variable* allows to set a specific crisp value for the variable just by clicking at the appropriate position in the graphical representation. Also, it is possible to change the colour of a specific linguistic term.

1.4.5 How to define the rules

```
IF cv_outdoor  = icy  and  cv_indoor  = cold THEN  cv_heating  =  full  (1)
IF cv_outdoor  = icy  and  cv_indoor  = medium THEN  cv_heating  =  full  (0.9)
IF cv_outdoor  = icy  and  cv_indoor  = warm THEN  cv_heating  =  half  (1)
IF cv_outdoor  = cool  and  cv_indoor  = cold THEN  cv_heating  =  full  (0.8)
IF cv_outdoor  = cool  and  cv_indoor  = medium THEN  cv_heating  =  half  (1)
IF cv_outdoor  = cool  and  cv_indoor  = warm THEN  cv_heating  =  half  (0.7)
IF cv_outdoor  = mild  and  cv_indoor  = cold THEN  cv_heating  =  half  (1)
IF cv_outdoor  = mild  and  cv_indoor  = medium THEN  cv_heating  =  half  (1)
```

6

Figure 1.6: Example of Rules

Defining the rules in FLIP++ is an easy task. You just position your mouse pointer in the rules window at the bottom of the main application window. Then you can write the rules. Of course certain syntactical limitations must be observed when designing rules. Each rule follows the structure

IF (term-list) THEN (term) WITH CERTAINTY (x).

where term-list is

(term) (concat_operator) [(term) (concat_operator) ...]

and term is

(linguistic variable) = (linguistic term)

and concat_operator is

{ AND | OR }

The item *UPDATE edits* in the menu *rules* checks whether any changes made in the rules window are syntactically sound and adds the rules to the rulebase. The item *show how much rules* allows to display either only the rules of the currently selected linguistic variable (cf. Section 1.2.1.1) in the rules windows or all the rules of all linguistic variables. It is also possible to change the output style using *change output rule style*. Then, '=' is used instead of IS and the certainty of a rule is displayed in parenthesis. Of course, it is possible to edit the rules in either style just depending on which is currently selected.

1.4.6 How to select operators

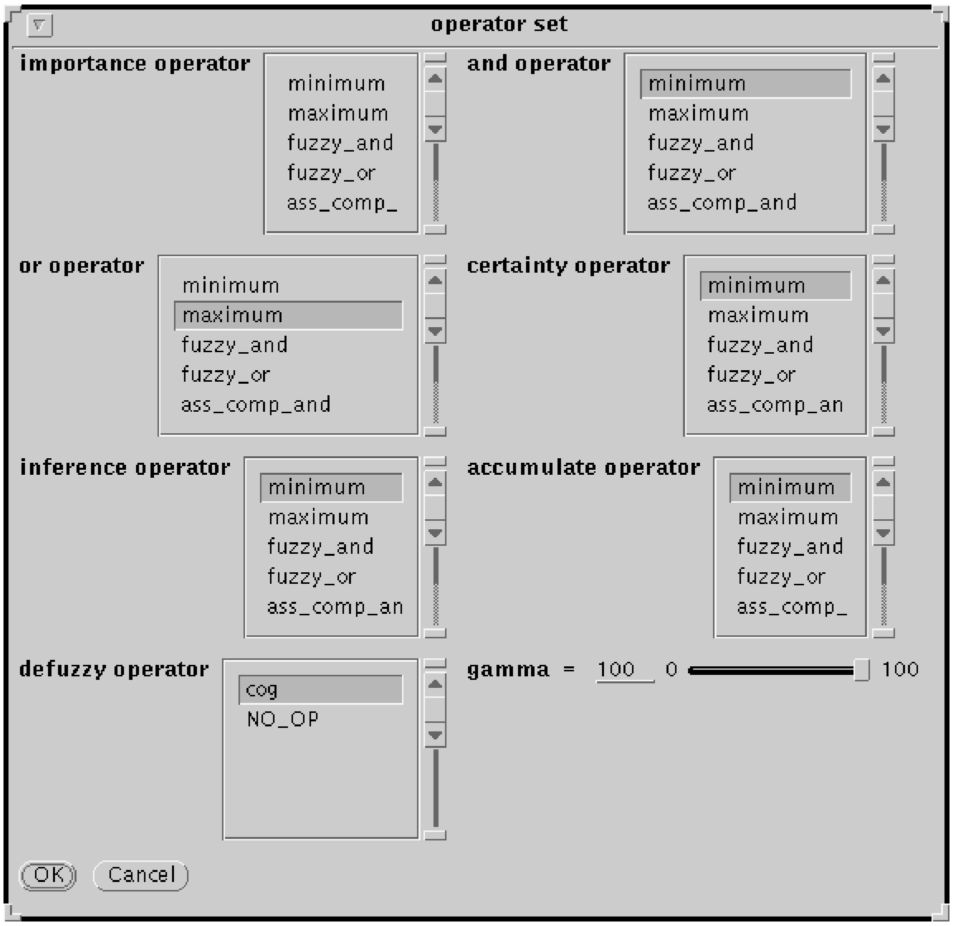


Figure 1.7: Selection of Operators

As we have seen above, within the rules there are operators used to concatenate the various terms. FLIP++ offers some operators the user can choose from. By clicking the item *EDIT operator set* in the *rules* window a new window *operator set* is displayed on the screen. Here you can specify the operators used for the *and*, *or*, *importance*, *certainty*, *inference*, *accumulate*, and *defuzzification* operations. For a more detailed description of these various operations have a look to the FLIP++ library description (cf. Section 1.2, page 4).

1.4.7 How to evaluate and interpret the results

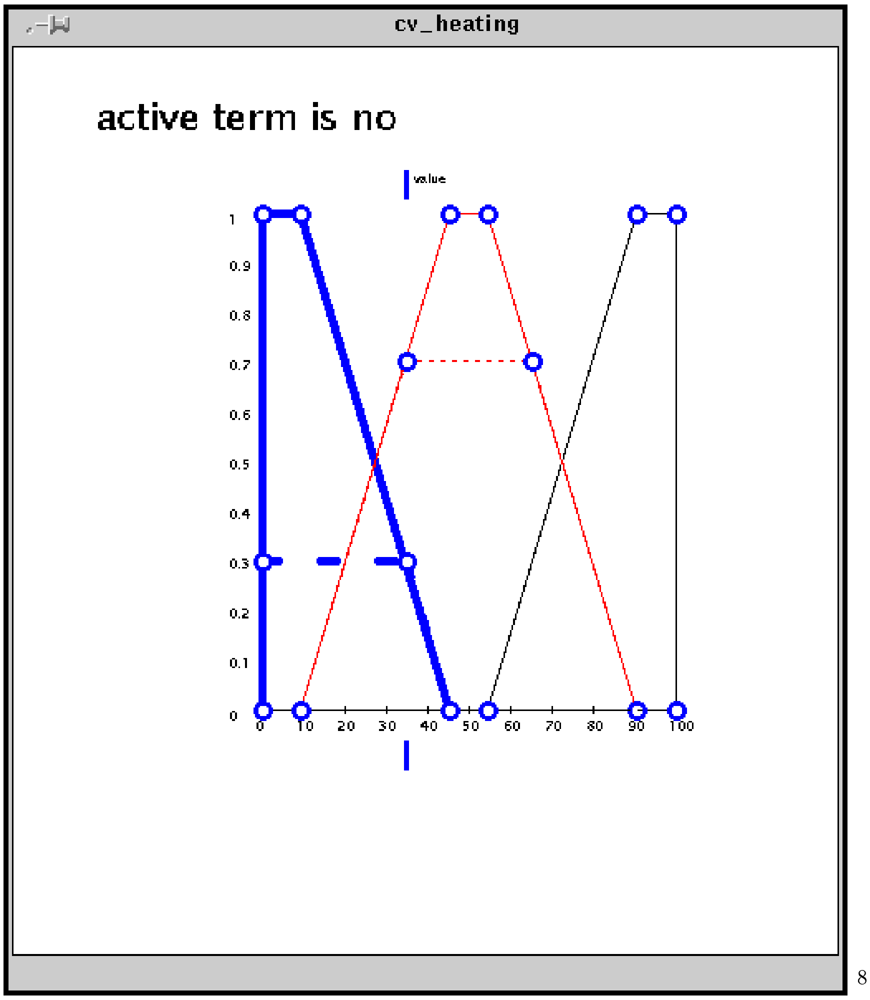


Figure 1.8: How to evaluate and interpret the results

After all parts of a system are defined, we can move on to start the computation. This is done by clicking the *EVALUATE* button in the main application window, which results in computing the value of the currently selected linguistic variable (cf. Section 1.2.1.1). So, we must first select our output variable *cv_heating*. After the evaluation has been completed, the crisp value is displayed below the *terms* window. To get a better impression of the result and be able to make some interpretations , it is necessary to see the fuzzy value of the result. FLIP++ also allows the user to do this by clicking the *show instantiated msf* button. In the same manner as the membership function (cf. Section 1.2.1.1) the instantiated membership function generated by the evaluation is displayed. For each of the linguistic terms (cf. Section 1.2.1.1) that b elong to the selected and evaluated linguistic variable (cf. Section 1.2.1.1) the instantiation shows how well it is satisfied. Of course, the instantiated membership function (cf. Section 1.2.1.1) can not be edited since it is created by the calculation. Often it provides the user with a good view of whether the system is working to its needs or not and therefore delivers a basis for fine-tuning steps.

Chapter 2

ConFLIP++

2.1 Knowledge Engineer

Every real-life task imposes constraints of different kinds and different importances that should not be violated to achieve viable solutions. Typical constraints that have to be dealt with in schedulingsystems include deadlines or the exchange of machine parts before they wear out. As it is not possible to know beforehand if all constraints can be satisfied in any given system the notion of importance of a given constraint is a very important measure that can help as a guideline if constraint violations are inevitable.

It is easy to see that constraints can come in two ways that can be identified as crisp constraints (e.g. a timeline that has to be met) on the one hand and fuzzy constraints (e.g. a chemical compatibility has to be a member of a specified set in a membership function) on the other hand.

For any given system at hand all constraints are specified in the ConFlip++ module by the means of simple constraints (also called compare constraints) and composed constraints (also called concatenated constraints).

- Compare Constraints (cf. Section 2.1.1)
- Compare Constraints example (cf. Section 2.1.2)
- Composed Constraints (cf. Section 2.1.3)
- Composed Constraints example (cf. Section 2.1.4)

2.1.1 Compare Constraints

This is an atomic type of constraint that can be used as a standaloneconstraint or to compose a more complex concatenated constraint.

In the following the parameters of this constraint together with a short description are given:

Parameter	**Description**
name	specifies the unique name of a constraint
importance	a real number in the range of [0..1] denoting the relative importance of a constraint.
dilatation	denotes if the constraint (more exact: the variable of the constraint) is of the type fuzzy, crisp or mixed (though a mixed constraints is fuzzy in essence)
comment	this optional parameter can be used for a short description by the user
variable	the unique name of the variable a constrained is designed for
compare_operator	any operator out of the operator set $\{=;;<;>\}$
compare_value	the value to which the variable is compared to

Table 2.1: Parameters of Compared Constraints

2.1.2 Example

Parameter	**Real-World Instance**
name	lu-c1
importance	0.9
dilation	fuzzy
comment	'Aluminum constraint 1'
variable	alu (has to be defined already in FLIP++)
compare_operator	$<$
compare_value	0.01

Table 2.2: Example of a Compared Constraint

2.1.3 Composed Constraints

Composed constraints are the combination of either two compare constraints, one compare constraint and one composed constraint or two composed constraints. They are useful to create constraint trees by the means of aggregating constraints as well as for modeling antagonistic criteria. The complexity and type of the constraintsin a system to be modeled depends solely on the user. In the following the parameters of this constraint together with a short description are given:

Parameter	**Description**
name	specifies the unique name of the constraint
importance	a real number in the range of [0..1] denoting the relative importance of a constraint.
dilatation	denotes if the constraint (more exact: the variables of the constraint) is of the type fuzzy, crisp or mixed (though a mixed constraints is fuzzy in essence)
comment	this optional parameter can be used for a short description by the use
constraint_1	name of the 1st-component constraint
constraint_2	name of the 2nd-component constraint
concat_operator	a conjunction operator out of the operator set {and; or}

Table 2.3: Parameters of Composed Constraints

Note that as usual the meaning of the concat_operator can be defined by what the user finds meaningful - the default operators are fuzzy and and fuzzy or.

2.1.4 Example

Suppose for the example that we have two compare constraints alu-c1 (the one of the example above) and another compare constraint called ni-c1 (that would look something like the alu-c1 costraint though that is not really importance). The concatenation constraint could look like this:

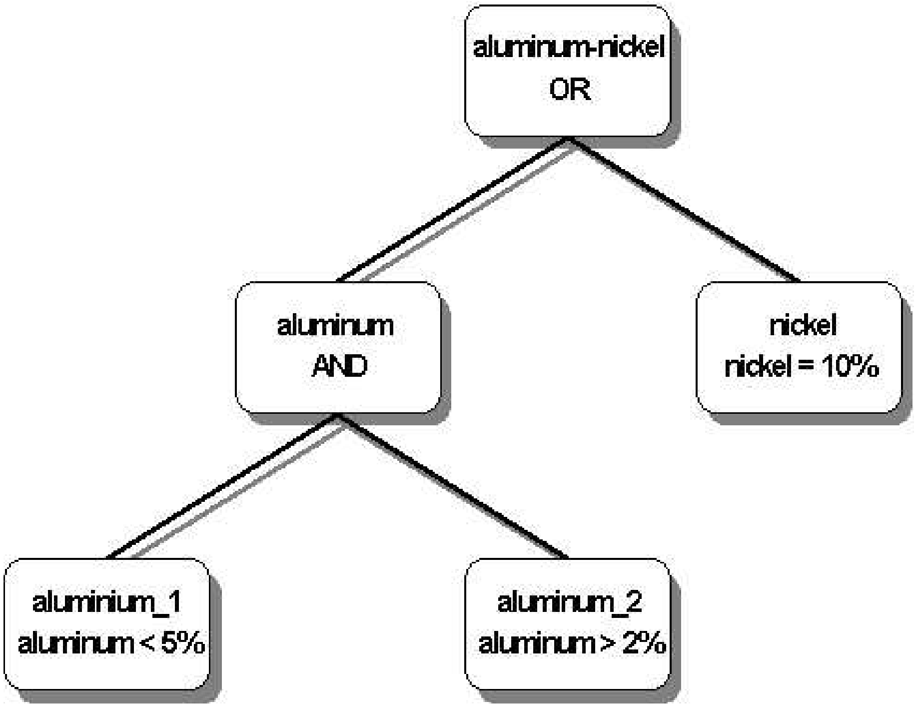
Parameter	**Real-World Instance**
name	alu-c1-ni-c1
importance	0.8
dilatation	fuzzy
comment	combined aluminum-nickel constraint
constraint_1	alu-c1
constraint_2	ni-c1
concat_operator	and

Table 2.4: Example of a Composed Constraint

Of course the question arises about a ruleset for any concatenated constraints. The answer is that whenever one creates a concatenated constraint Conflip++ automatically creates a new ruleset out of default or user-defined values. This automatically created ruleset can of course be fine-tuned by the human expert.

Another very important notion about ConFlip++ is *SetofConstraints*. Each SetofConstraints consists of a name and a list of constraints (that can either be of type compare or concat). Furthermore it has a ruleset-object, tables for concatenating and comparing constraints with the appropriate operators defined in the OperatorSet, and a parameter set describing fuzzy linguistic variables. This makes it obviously easy to group and administer constraints that belong together.

A typical constraint-tree that is in fact part of a real-world steelmaking-scheduling system using StarFlip++ is shown intable 2.1.



1

Figure 2.1: A typical Set of Constraints

Besides of being the place for declaring and defining the constrains that are inherent in the system to be modelled ConFlip++ is as well the place where the **evaluation** of a constraint takes place. The three steps of constraint evaluation are as follows:

1. All free linguistic variables are assigned numbers that are either defuzzified real numbers or possibility distributions.

2. The evaluation functon returns a value (by default a defuzzified real number) that describes the degree of satisfaction of a constraint with its assigned number(s).

3. Evaluated constraints are retained in a *SetOfEvalConstraints* for further operations.

Of course there can be more than one SetOfEvalConstraints with the same idea of grouping constraints that somehow belong together as we have seen above for SetOfConstraints.

2.2 ConFLIP++ Library description

2.2.1 ConFLIP++ modules

- cflplist.h / cflplist.cpp (cf. Section 2.2.2)
- cflplib.h / cflplib.cpp (cf. Section 2.2.3)
- conflip.h / conflip.cpp (cf. Section 2.2.4)
- cflp_dyn.h / cflp_dyn.cpp (cf. Section 2.2.5)

2.2.2 cflplist.h / cflplist.cpp

- Node1d (cf. Section 2.2.2.1)
- GenList (cf. Section 2.2.2.2)
- GenSled (cf. Section 2.2.2.3)

2.2.2.1 Node1d

This class is used to generate an item of the GenList class. It is the most generic way to describe an element of a linear list by storing a reference to the item, its predecessor, and its successor. The item is a void pointer and is set to NULL when a new Node1d object is initialized.

2.2.2.2 GenList

The GenList class is a generic list type used for all kinds of linear cyclical lists in ConFLIP++. It provides the generic framework for cyclical lists used for derivations of specific lists. The attributes of the class are the *root* node, which is always empty and therefore indicates the start of the list, with the integer attribute *no_of_items* storing the number of items contained in the list, and the attribute *flag* showing the status of the list.

There are three possibilities to generate a GenList object: Either one uses the constructor without parameters, which causes an empty list to be generated via the protected *init* method, otherwise it is possible to construct a new GenList object as a copy from an existing GenList object. Alternatively to the latter form of initialization it is also possible to use the constructor which takes a GenSled (cf. Section 2.2.2.3) object associated to a GenList object as an argument and hence generates a new GenList object as a copy of the GenList associated with the specified GenSled object.

The attribute *flag* is set to *lf_init* with all three constructors. The destructor deletes the *root* node. The method *clear* empties the list by deleting all its elements with the help of the *del* method which needs to be specified according to the specific item of the list, the *flag* is also set to *lf_init*. The method *copy* copies all the items of an existing GenList object to the current GenList object using the item specific *cpy* method.

The overloaded operator = enables the user to assign one GenList object to another, just as it is with normal variables. The same is valid for the = operator with a GenSled object on the right side, which assigns the GenList object associated to the GenSled object to the GenList object on the left side. The overloaded compare operator == renders it possible to compare two GenList objects for identity. Two lists are considered to be identical if they have the same number of items, which in turn have to be identical. The comparison of two items is made through the use of the *cmp* method where the terms of identity between items must be specified.

The method *weak_equal* checks whether two lists consist of the same set of elements, this can be interpreted as a weak form of identity. The method *read* reads a GenList from a file via the protected method *rd* and adds them with the *add_quick* method described below to the current GenList object. Similarly, the method *write* writes the contents of the GenList to a file via the protected method *wr*. The method *get_how_full* returns the number of elements contained in the list. The method *get_flags* retrieves, the methods *set_flags* and *clr_flags* manipulate the *flag* attribute of the list. The method *show* writes the list, i.e. its nodes with their items, via the *write* method to the standard output device. The methods *del*, *key*, *look*, *cmp*, *cpy*, *rd*, and *wr* depend on the specific details of the item contained in each node object, which makes it necessary to design them according to the needs of the specific items.

Hence it follows that for the generic and abstract classes described here they do not have any functionality, and must be rewritten in any derived class. The method *insert* puts a new item (info) into the list after the node stated in the parameter node, through the *InfoCopy* parameter it is possible to indicate whether the new item should be copied or moved. The success of the insertion procedure is shown in the boolean return value. The method *kick_out* deletes a node from the list by simply deleting its item with the *del* method, unlinking it from its predecessor and successor, and then deleting the node itself. The *kick_out* method has also an alternative variant which makes it possible for us to specify the item of a node object, so that the node object containing this item is removed from the list.

The *get_info* method returns the item contained in a list node specified in the parameter. The *add* method puts a new item into the list via the *add_quick* method described below, but only if no other node with the same key for the item exists. The key of an item in the list is determined by the protected method *key*.

The method *add_quick* adds a new item just before the root node of the cyclic list, which is the end position of the list. The insertion is in fact executed by the *insert* method. The *lookup* method comes in three variants. The first one tries to find an item of the list starting from the node stated in the parameter node. If the item was found the count parameter holds the distance from the starting node to the node containing the found item. The second *lookup* leaves out the count parameter and therefore simply calls the first *lookup* with a dummy variable for the count parameter. The third *lookup* has only the item to be searched as parameter and calls the second variant of *lookup* with the first node of the list as starting node, i.e. the successor of the *root* node. This variant checks the whole list whether the specified item exists or not.

2.2.2.3 GenSled

This class is always associated with a GenList (cf. Section 2.2.2.2) and offers various kinds of handling the GenList object and its contents. By setting the flags of the GenList object it is possible to allow or disallow certain operations to the GenSled object. Among these operations are the *insert*, *add*, *add_quick*, and *kick_out* methods, all of which can make changes to the contents of the list. The class has three attributes: a pointer to a Node1d (cf. Section 2.2.2.1) object referring to the currently positioned node in the list, a pointer to a GenList object and an integer variable *index* stating the position in the list.

There are three different constructors for the GenSled class. The first one simply initializes a GenSled object with the pointers set to *NULL* and the *index* to zero. The second constructor has a reference to a GenList object as a parameter and sets the *list_ptr* to this reference, the *index* to the first element (zero when no element) and the node pointer to the first node (*root* when no element). The third variant of the constructor is exactly the same as the second one except for the parameter, which is not a reference but a pointer argument.

There are several overloaded operators in order to support navigation in the list. The *++* operator following its assigned object accesses the next item in the list in a forward direction and returns a pointer to the new item of the list. Whereas, the *++* operator preceding its assigned object returns a pointer to the old item of the list after accessing the next item in the list. This is just the same as in C++ with the i++ and ++i statements. Of course, the same procedure applies to the – operator. Furthermore, the [] operator tries to access an element specified by an integer index. This makes it possible to treat the list as an array, if the list is empty or the specified index is zero or less then the list is positioned on the root node. In any case the item of the positioned node is returned. The += operator advances several steps specified by the parameter after the operator. The same applies to the -= operator which goes backwards instead of forwards. The *()* operator returns the item of the currently positioned node.

The method *isvalid* checks whether the currently positioned node is valid, i.e. different from the *root* pointer.The *islast* method checks whether the currently positioned node is the last in the list, similarly the *isfirst* method checks whether the currently positioned node is the first in the list. The methods *first* and *last* return the item of the first and last node respectively. The method *position* returns the *index* of the currently positioned node. The lookup *method* checks whether an item is contained in the list or not by the use of the *lookup* method of the GenList object. In case the item is in the list it positions on the node containing the item and returns the item, otherwise it positions on the *root* node and returns *NULL*. The method *insert* first checks whether it is possible to insert an item into the list, this is indicated by the *lf_insert* flag set in the associated GenList object. If so, the item specified in the parameter is inserted into the list by the use of the *insert* method of the GenList object. The same procedure is valid for the *add* and the *add_quick* method.

Similarly the *kick_out* method is also accessible through the GenSled object after the *flag* (*lf_kick_out*) of the GenList (cf. Section 2.2.2.2) object has been checked. If the *flag* setting allows the GenSled object to kick out then the currently positioned item is removed and the GenSled is positioned to its predecessor. The method *get_how_full* returns the number of items in the list.

2.2.3 cflplib.h / cflplib.cpp

- ListOfStrings (cf. Section 2.2.3.1)
- SledOfStrings (cf. Section 2.2.3.2)
- ListOfLingTerms (cf. Section 2.2.3.3)
- SledForLingTerms
- SetOfLingTerms (cf. Section 2.2.3.4)
- ListOfSetsOfLingTerms (cf. Section 2.2.3.5)

- SledForSetsOfLingTerms (cf. Section 2.2.3.6)
- Table (cf. Section 2.2.3.7)
- TableConcat (cf. Section 2.2.3.9)
- ListOfTables (cf. Section 2.2.3.10)
- SledForTables (cf. Section 2.2.3.11)
- ListOfParameterSets (cf. Section 2.2.3.12)
- SledForParameterSets (cf. Section 2.2.3.13)

2.2.3.1 ListOfStrings

This class is derived from the GenList (cf. Section 2.2.2.2) class and is especially used for the handling of lists of strings.

The constructors of the object are the same as with the GenList object. The destructor clears the whole list via the use of the inherited *clear* method. To specify the class according to the use of string items it is necessary to modify the internal protected methods of the GenList class. Among these are the *del*, *key*, *look*, *cmp*, *cpy*, *rd* and *wr* method. The *del* method simply deletes an item of the list, which is referred to by a pointer, with the delete command and casting the pointer to a character pointer indicating that it is a reference to a string. The *key* method returns its argument meaning that with a string item the key is identical to the string item. The *look* and *cmp* method check whether two items of the list are equal or not by doing a string comparison. The *cpy* method copies a string value into a list item with the length set according to the actual string length. This is used in the *copy* method (see GenList (cf. Section 2.2.2.2)). The *rd* method reads a single list item from a file into the list, whereas the *wr* method writes a single list item to a file.

The remaining methods of the class are implemented by the direct use of GenList methods, among them are the operators =, == and the methods *weak_equal*, *add*, *add_quick*, *kick_out*, and *lookup*. They are overwritten in order to modify the parameter types to strings.

2.2.3.2 SledOfStrings

This class does exactly the same as the generic GenSled (cf. Section 2.2.2.3) class, except that the parameters used and the results returned are character pointers instead of void pointers, which is necessary for a string list. The class operates as sled for the ListOfStrings (cf. Section 2.2.3.1) class.

2.2.3.3 ListOfLingTerms

This class is derived from the GenList (cf. Section 2.2.2.2) class and is especially used for the handling of lists of linguistic terms.

The constructors of the object are the same as with the GenList object. The destructor clears the whole list via the use of the inherited *clear* method. To specify the class according to the use of linguistic term items it is of course necessary to modify the internal protected methods of the (cf. Section 2.2.2.2) class. Among these are the *del*, *key*, *look*, *cmp*, *cpy*, *rd* and *wr* method.

As linguistic terms are also strings the methods *del*, *key*, *look*, *cmp* are the same as those in the ListOfStrings (cf. Section 2.2.3.1) class. Futhermore, the method *cpy* copies a string value, which is a linguistic term, into a list item. Contrary to the *cpy* method of the ListOfStrings, the length of the string is set to the maximum linguistic term length independent of the actual term length. The method *rd* reads a single list item from a file and checks prior to the insertion of the data into the item whether it forms a valid linguistic term. The check is done in the function *check_ling*. Only strings within the length boundaries of linguistic terms and containing no illegal characters are considered to be valid. The method *wr* writes a single list item to a file. The method *insert* also checks whether a valid linguistic term is inserted into the list and then makes the insertion through the inherited *insert* method. The method *rename* replaces all items of the list containing a specific linguistic term with another linguistic term.

The remaining methods of the class are implemented by the direct use of GenList (cf. Section 2.2.2.2) methods. These are the methods *add*,*kick_out*, *lookup*, *weak_equal*, and the operators = and ==. Their parameter types are modified according to the specific classes. This class does exactly the same as the generic GenSled class, except that the parameters used and the results returned are character pointers instead of void pointers, which is necessary for string (linguistic term) items. The class operates as a sled for the ListOfLingTerms (cf. Section 2.2.3.3) class.

2.2.3.4 SetOfLingTerms

This class stores a whole set of linguistic terms. It is derived from the ListOfLingTerms (cf. Section 2.2.3.3) class. The main difference between the basic class and the derived class is that the attribute *name_of_set*, which stores the name of the set of linguistic terms, is added to the derived class.

The constructor without parameters initializes a set with a default set name and an empty list of linguistic terms. The constructor with another SetOfLingTerms object specified as parameter initializes the object with the same values as the object in the argument. The destructor is an empty function. The method *clear* deletes all elements of the linguistic term's list by the *clear* method of the ListOfLingTerms class. The methods *copy*, operator =, operator ==, and *weak_equal* need to be adapted due to the additional *name_of_set* attribute. The method *check_equality* is identical to the method *weak_equal*. The methods *read*lesswrite need also to be adapted, so that the additional attribute apart from the list of linguistic terms will be read andwritten.

Furthermore, the methods *get_name_of_set* and *set_name_of_set* enable the getting and setting of the *name_of_sest* attribute. The *set_name_of_set* checks via the *check_set* function whether the name of the set is a valid name. The *set* method sets the *name_of_set* attribute and the ListOfLingTerms object according to its parameters, which makes it possible to combinean existing ListOfLingTerms with a name to form a SetOfLingTerms object.

2.2.3.5 ListOfSetsOfLingTerms

Several sets of linguistic terms, i.e. several linguistic variables, are combined in this class. It is derived from the GenList (cf. Section 2.2.2.2) class.

The three types of constructors are implemented in the same way as the three mentioned in the GenList class description. One initializing a default (=empty) list, one initializing the list as a copy of another list, and finally one initializing the list with the values of a list associated to a specified sled. The destructor simply empties the list using the *clear* method. The method *del* deletes one item of the list, which is a SetOfLingTerms (cf. Section 2.2.3.4) object. The method *key* returns the key value of a list item, which is the name of the *SetOfLingTerms* object. The methods *cmp* and *look* compare the key value of the specified list item with the specified string value and return the result of the comparison in a boolean value. The method *cpy* copies a specified SetOfLingTerms object into the specified item. The methods *rd* and *wr* read/write an item of the list from/to a file. The remaining methods are implemented by using the GenList methods. The methods *add*, *kick_out*, *lookup*, *weak_equal* and the operators =, == are modified for the specific types of parameters.

2.2.3.6 SledForSetsOfLingTerms

This class does exactly the same as the generic GenSled (cf. Section 2.2.2.3) class, except that the parameters used and the results returned are SetOfLingTerms (cf. Section 2.2.3.4) pointers instead of void pointers, which is necessary for this type of list. The class operates as sled for the ListOfSetsOfLingTerms (cf. Section 2.2.3.5) class.

2.2.3.7 Table

This class is the basic class for the description of the operator functionality.It is an abstract class, which is used for the two table types concat and compare described below. It contains the table of possible results when an operator is applied to a comparison between a linguistic variable and a value or when it is used for the concatenation of constraints. The attributes arse the *sname* and the *type* of the Table.

The constructor without parameters initializes a default Table with the use of the *init* method. The constructor with another Table object as argument initializes a Table object with the same contents as the specified Table object via the *copy* method. The destructor is an empty function. The method *init* generates a default table name using the function *get_default_table_name*. The method *clear* is an empty function. The method *copy* copies the attributes of the specified Table object into the object. The "=" operator enables the assignment operation between Table objects. The "==" operator compares two Table objects using the attributes *name* and *type* as comparison criterion, and returns the result of the comparison as a boolean value. The *get_name* and *get_type* methods return the values of the respective attributes. The *set_name* method sets the name of the Table object, after it has checked the name by the use of the *check_table* function. The method *write* writes the attributes of the Table object to a file. The method *read* reads from a file, and prior to assigning the name attribute it checks whether the read name is a valid Table object name by using the *check_table* function.

2.2.3.8 TableCompare

This class is derived from the Table (cf. Section 2.2.3.7) class and represents the compare operator functionality. This table is used when a compare expression is evaluated (e.g. temperature , >=, = and *!=*. The attributes are the three ListOfLingTerms (cf. Section 2.2.3.3) arrays *in*, *out*, and *table_compare*, which contain the relationship between the linguistic terms of the input linguistic variable and the linguistic terms of the output linguistic variable. For the compare operators the input and output linguistic variables can either be of fuzzy or crisp dilatation type. Therefore with each method it is necessary to specify the dilatation type. The arrays contain entries for each of the dilatation types.

The default constructor initializes a default table with the *type* attribute set to *t_comp* indicating a compare table. The alternative constructor sets attributes of the TableCompare object according to the contents of the TableCompare object specified in the argument. The destructor is an empty function. The method *clear* deletes the contents of all the attributes. The method *copy* copies all the attributes of another TableCompare object to the current object.

The = operator enables the assignement between TableCompare objects or between a TableCompare and a Table (cf. Section 2.2.3.7) object. The == operator compares two TableCompare objects using all the attributes as comparison criterion, and returning the result of the comparison in a boolean value. The methods *read* and *write* read/write all attributes of the object from/to a file.

The method *set_in_vars* sets the linguistic terms of the input linguistic variable for the compare expression. It checks whether the dilatation type is crisp or fuzzy and wether the output linguistic variable has already been set or not . Then the entries in the table of compare operators for the type of input variable are set to the corresponding output variable. For each linguistic term of the input variable there must be a corresponding entry in the table of compare operators. Compare operators exist only for crisp and fuzzy dilatation types and have one input and one output variable.

The method *set_out_vars* is rather similar to the previous method. It inserts the linguistic terms of the output linguistic variable and sets the table of compare operators accordingly. The method *set_in_to_out_vars* sets the table of a specific compare operator to the linguistic terms given in the argument. The specified linguistic terms must be valid linguistic terms of the output linguistic variable. The method *set* enables to set one single linguistic term, which corresponds to a linguistic term of the input linguistic variable, in the table of a specific compare operator.

The method *add_in_var* adds one specified linguistic term to the input linguistic variable and extends the table of compare operators with one entry for each operator corresponding to the new input linguistic term. The method *add_out_var* extends the output linguistic variable with one linguistic term. The *kick_out_in_var* deletes a linguistic term of the input linguistic variable and the corresponding entries in the table of compare operators. The method *kick_out_out_var* deletes a linguistic term of the output linguistic variable. The method *rename_in_var* renames a linguistic term of the input linguistic variable. The method *rename_out_var* renames a linguistic term of the output linguistic variable.

The methods *get_in*, *get_out*, and *get_in_to_out* return sleds for the input linguistic variable, the output linguistic variable, and the table of compare operators respectively.

2.2.3.9 TableConcat

This class is derived from the Table (cf. Section 2.2.3.7) class and represents the concat operator functionality. This table is used when a concat expression is evaluated (e.g. temperature_constraint AND humidity_constraint). Contrary to the compare operator in the class TableCompare (cf. Section 2.2.3.8), the concat operator needs two input linguistic variables for the concatenation. The concat operators are *and* and *or*. The attributes are the three ListOfLingTerms (cf. Section 2.2.3.3) arrays *in*, *out*, and *table_concat*, which contains the relationship between the linguistic terms of the input linguistic variables and the linguistic terms of the output linguistic variable. For the concat operators the input linguistic variables can either be of fuzzy or crisp dilatation type, whereas the output linguistic variable can additionally be of mixed dilatation type. Therefore with each method it is necessary to specify the dilatation type. The arrays contain entries for each of the dilatation types.

The default constructor initializes a default table with the *type* attribute set to *t_conc* indicating a concat table. The alternative constructor sets attributes of the TableConcat object according to the contents of the TableConcat object specified in the argument. The destructor is an empty function. The method *clear* deletes the contents of all the attributes. The method *copy* copies all the attributes of another TableConcat object to the current object. The = operator enables the assignement between TableConcat objects or between a TableConcat and a Table (cf. Section 2.2.3.7) object. The == operator compares two TableConcat objects using all the attributes as comparison criterion, and returning the result of the comparison in a boolean value. The methods *read* and *write* read/write all attributes of the object from/to a file. The method *set_in_vars* sets the linguistic terms of the specified input linguistic variable for the concat expression.

It is checked whether the the dilatation type is crisp or fuzzy and the output linguistic variable has already been set. Then the entries in the table of concat operators for the type of input variable are set to the corresponding output variable. For each combination of linguistic terms of the input variables there must be a corresponding entry in the table of concat operators. The method *set_out_vars* is rather similar to the previous method. It inserts the linguistic terms of the output linguistic variable and sets the table of concat operators accordingly. As the output linguistic variable can also be of mixed type, which is the result of a concatenation of a crisp and a fuzzy input linguistic variable, this must also be considered when initializing the table of concat operators. The method *set_in_to_out_vars* sets the table of a specific compare operator to the linguistic terms given in the argument. The specified linguistic terms must be valid linguistic terms of the output linguistic variable. The method *set* enables to set one single linguistic term, which corresponds to a combination of linguistic terms of the input linguistic variables, in the table of a specific concat operator.

The method *add_in_var* adds one specified linguistic term to the specified input linguistic variable and extends the table of concat operators with one entry for each linguistic term of the other input linguistic variable and each operator corresponding to the new input linguistic term. The method *add_out_var* extends the output linguistic variable with one linguistic term. The *kick_out_in_var* deletes a linguistic term of the specified input linguistic variable and the corresponding entries in the table of compare operators. The method *kick_out_out_var* deletes a linguistic term of the output linguistic variable. The method *rename_in_var* renames a linguistic term of the specified input linguistic variable. The method *rename_out_var* renames a linguistic term of the output linguistic variable.

The methods *get_in*, *get_out*, and *get_in_to_out* return sleds for the specified input linguistic variable, the output linguistic variable, and the table of compare operators respectively.

2.2.3.10 ListOfTables

This class is derived from the abstract class GenList (cf. Section 2.2.2.2). It represents a collection of several operator Tables.

The default constructor generates an empty list. The other two constructors generate a list with the contents copied from the ListOfTables object specified in the parameter. The parameter is either a reference to the ListOfTables object or a reference to a SledForTables (cf. Section 2.2.3.11) which is associated to a ListOfTables object. The destructor deletes all elements of the list. The method *del* deletes an item of the list, which either can be a TableCompare (cf. Section 2.2.3.8) or a TableConcat (cf. Section 2.2.3.9) object. The method *key* returns the name of the referenced Table (cf. Section 2.2.3.7) askey.

The method *look* compares a string with the name of the referenced Table. The method *cmp* checks whether two referenced Tables are identical. The method *cpy* copies one Table object into another one. The method *rd* reads a Table from a file into an item. The method *wr* writes a Table onto a file. The = and == operators, and the *weak_equal*, *add*, *kick_out*, and *look_up* methods use the inherited methods of GenList (cf. Section 2.2.2.2) and operate in the same way.

2.2.3.11 SledForTables

This class does exactly the same as the generic GenSled (cf. Section 2.2.2.3) class, except that the parameters used and the results returned are Tables (cf. Section 2.2.3.7) pointers instead of void pointers, which is necessary for this type of list. The class operates as sled for the ListOfTables (cf. Section 2.2.3.10) class.

2.2.3.12 ListOfParameterSets

This class is a collection of ParameterSets, which are objects of the underlying FLIP++ library. It is derived from the abstract class GenList (cf. Section 2.2.2.2). It represents a collection of several linguistic variables.

The default constructor generates an empty list. The other two constructors generate a list with the contents copied from the ListOfParameterSets object specified in the parameter. The parameter is either a reference to the ListOfParameterSets object or a reference to a SledForParameterSets (cf. Section 2.2.3.13) which is associated with a ListOfParameterSets object. The destructor deletes all elements of the list while the method *del* deletes an item of the list. The method *key* returns the name of the referenced ParameterSet as a key. The method *look* compares a string with the name of the referenced ParameterSet. The method *cmp* is not available. The method *cpy* copies one ParameterSet object into another . The method *rd* reads a ParameterSet from a file into an item. The method *wr* writes a Parameter-Setfrom an object into a file. The = and == operators, and the *weak_equal*, *add*, *kick_out*, and *look_up* methods use the inherited methods of GenList (cf. Section 2.2.2.2) and operate in the same way.

2.2.3.13 SledForParameterSets

This class does exactly the same as the generic GenSled (cf. Section 2.2.2.3) class, except that the parameters used and the results returned are ParameterSets pointers instead of void pointers, which is necessary for this type of list. The class operates as sled for the ListOfParameterSets (cf. Section 2.2.3.12) class.

2.2.4 conflip.h / conflip.cpp

- InfoB (cf. Section 2.2.4.1)
- Info (cf. Section 2.2.4.2)
- InfoCompare (cf. Section 2.2.4.3)

- InfoConcat (cf. Section 2.2.4.4)
- ListOfInfos (cf. Section 2.2.4.5)
- SledForInfos (cf. Section 2.2.4.6)
- Constraint (cf. Section 2.2.4.7)
- ConstraintCompare (cf. Section 2.2.4.8)
- ConstraintConcat (cf. Section 2.2.4.9)
- ListOfConstraints (cf. Section 2.2.4.10)
- SledForConstraints (cf. Section 2.2.4.11)
- SetOfConstraints (cf. Section 2.2.4.12)

2.2.4.1 InfoB

This is an internal abstract class which provides the framework for the InfoCompare (cf. Section 2.2.4.3) and InfoConcat classes. The main purpose is to provide information about a constraint, which is then used in the SetOfConstraints (cf. Section 2.2.4.12) class. The information encompasses a *name*, a *type* indicating the type of constraint (compare or concat), an *importance* attached to a constraint, the *dilatation* (crisp or fuzzy), and a *comment* about the constraint.

The default constructor initializes an object with the use of the internal *init* method. The *init* method sets a default *name* generated by the function *get_default_info_name*, sets the *importance* to zero, the *dilatation* to crisp, and the comment to an empty string. As an alternative, an object can be created as a copy of another InfoB object by applying the *copy* method. The destructor and the method *clear* are empty functions, as this is an abstract and non-dynamic class. The method *copy* is used by the above mentioned constructor and copies the contents of another InfoB object to the current one. The operator = is used for assignment. The operator == checks whether two InfoB objects have the same contents in the attributes. Also, the methods *read* and *write* handle input/output from/to files.

Furthermore, there are trivial get and set methods to retrieve and set the values of the attributes. For the *type* attribute there is only a get method because this attribute is set internally by assignment in the InfoCompare (cf. Section 2.2.4.3) and InfoConcat (cf. Section 2.2.4.4) object respectively. There also exists an aggregate *set* method where all attributes, again with *type* as exception, that can be set with one call. The *name* attribute must also fulfill length and character requirements, which are observed by the *check_info* function.

2.2.4.2 Info

This class is derived from the virtual class InfoB (cf. Section 2.2.4.1), it does not add any funcionality. So it is identical with the InfoB class. In fact, it is not treated as an abstract class, and can therefore be used as an object of its own.

2.2.4.3 InfoCompare

This class is derived from the InfoB (cf. Section 2.2.4.1) class and it is used to provide information about a compare constraint (e.g. temperature >= 30). The attributes are *var*, which is the linguistic variable contained in the constraint ('temperature' in the above example), *var_dilatation* which indicates whether the variable is crisp or fuzzy, *op* specifies the type of compare operator applied ('>=' in the above example), and *comp_value*, which is the linguistic variable that has to be compared with.

The default constructor initializes an object using the *init* method. The *init* method sets the *type* to *compare*, generates a default name for the linguistic variable, sets the linguistic variable type to crisp, the operator to equal, and the compare value to zero. There also exists a constructor which takes another InfoCompare object as an argument and sets the attributes of the current object correspondingly. The destructor and the method *clear* are empty functions. The *copy* method copies the values of an existing object to the current one and is used by the above mentioned constructor. The operators = and == are used for assignment and comparison. The methods *read* and *write* read/write the contents of the object from/to files. Futhermore, it is also possible to indicate in a boolean parameter in the methods *clear*, *copy*, *read*, and *write*, whether inherited parts from Info (cf. Section 2.2.4.2) should also be processed or not. Again, there are trivial get and set methods to retrieve and set the values of the attributes.

There also exists an aggregate *set* method which enables the setting of all attributes including the inherited ones at once.

2.2.4.4 InfoConcat

This class is derived from the InfoB (cf. Section 2.2.4.1) class and is used to provide information about a concat constraint (e.g. temperature_constraint AND humidity_costraint). The attributes are *const1* and *const2*, which are the names of the two constraints to be concatenated ('temperature_constraint' and 'humidity_constraint' in the above example), *op* which specifies the type of concat operator applied ('AND' in the above example).

The default constructor initializes an object using the *init* method. The *init* method sets the *type* to *concat*, generates default names for the constraints, and sets the operator to 'c_and'. There also exists a constructor which takes another InfoConcat object as an argument and sets the attributes of the current object correspondingly. The destructor and the method *clear* are empty functions. The *copy* method copies the values of an existing object to the current one and is used by the above mentioned constructor. The operators = and == are used for assignment and comparison. The methods *read* and *write* read/write the contents of the object from/to files.

Furthermore, it is also possible to indicate in a boolean parameter found in the methods *clear*, *copy*, *read*, and *write*, whether inherited parts from Info (cf. Section 2.2.4.2) should be processed too or not. Again, there are trivial get and set methods to retrieve and set the values of the attributes. There also exists an aggregate *set* method which enables the setting of all attributes including the inherited ones at once. The names of the constraints and the concat constraint must be unique.

2.2.4.5 ListOfInfos

This class collects several objects of the above described Info (cf. Section 2.2.4.2), InfoCompare (cf. Section 2.2.4.3), and InfoConcat (cf. Section 2.2.4.4) classes. It is derived from the generic class GenList (cf. Section 2.2.2.2).

The default constructor generates an empty list. The other two constructors generate a list with the contents copied from the ListOfInfos object specified in the parameter. The parameter is either a reference to the ListOfInfos object or a reference to a SledForInfos (cf. Section 2.2.4.6) which is associated to a ListOfInfos object. The destructor deletes all elements of the list. The method *del* deletes an item of the list. The method *key* returns the name of the referenced Info (cf. Section 2.2.4.2) as a key. The method *look* compares a string with the name of the referenced Info. The method *cmp* compares two referenced Infos for identity. The method *cpy* copies one Info object into another one. The method *rd* reads an Info from a file into an item. The method *insert* inserts an Info object into the list and in case of the concat constraint checks whether the underlying constraint are existent.

Also the method *kick_out*, with a node parameter, deletes all appearances of a constraint in other concat constraints of the list prior to deleting the constraint itself. The method *wr* writes an Info from an object to a file. The = and == operators, and the *weak_equal*, *add*, *kick_out*, and *look_up* methods use the inherited methods of GenList (cf. Section 2.2.2.2) and operate in the same way.

2.2.4.6 SledForInfos

This class does exactly the same as the generic GenSled (cf. Section 2.2.2.3) class, except that the parameters used and the results returned are Infos (cf. Section 2.2.4.2) pointers instead of void pointers, which is necessary for this type of list. The class operates as sled for the ListOfInfos (cf. Section 2.2.4.5) class.

2.2.4.7 Constraint

This is a virtual class derived from the InfoB (cf. Section 2.2.4.1) class and represents a framework for the two constraint classes ConstraintCompare (cf. Section 2.2.4.8) and ConstraintConcat, which are described below. The attributes of the class are *satisfaction*, which says how well a constraint is satisfied *cv_for_constraint*, which is the linguistic variable that is attached to the constraint, and *where_do_i_belong_to*, which refers to the SetOfConstraints (cf. Section 2.2.4.12) object the constraint belongs to.

The default constructor initializes an empty constraint. First it calls the constructor of InfoB (cf. Section 2.2.4.1), then it calls the method *init*. This method sets the satisfaction to zero and the pointers *cv_for_constraint* and *where_do_i_belong_to* to NULL. Again, there are alternative constructors. One initializing the Constraint object with the contents of another Constraint object. Another one initializing the inherited parts with the contents of the specified InfoB object. The destructor deletes the linguistic variable associated with the constraint. The method *clear* is an empty method. The method *copy* copies the contents of another Constraint object to the current one and is used by the constructor described above. The operators = is used for assignment, whereas the operator == is not implemented for this class.

The methods *read* and *write* handle input/output from/to files. The methods *read*, *write*, and *copy* have a boolean parameter indicating whether the inherited attributs should be processed too or not. The attributes *satisfaction* and *set_cv_for_const* can be set and retrieved with trivial methods. Furthermore, the set methods for the inherited attributes *name* and *importance* need to be modified. With *set_name* not only the inherited *name* attribute is set to the specified name, but also the name of the linguistic variable *cv_for_const*.

Furthermore, all appearances in concat constraints must be updated. With *set_importance* the importance of the linguistic variable is also set. The method *make_instance* creates a new linguistic variable (CombinedVariable object) for the constraint, where the number of linguistic terms is determined by the compare operator and dilatation type used. The method *bind_to_set* connects the Constraint object to the specified SetOfConstraints. The method *bind_to_flip* connects the Constraint object, which is already bound to a SetOfConstraints (cf. Section 2.2.4.12) object, to the FLIP++ library by assigning the linguistic variable *cv_for_const*. The appropriate variable is found by the name of the constraint, which is also the name of the variable, by searching in the rules of the SetOfConstraints object.

2.2.4.8 ConstraintCompare

This class is derived from the classes Constraint (cf. Section 2.2.4.7) and InfoCompare (cf. Section 2.2.4.3). It represents a compare constraint (e.g. temperature >= 30). The attribute *cv_for_var* is a reference to a linguistic variable used in the constraint.

The default constructor calls the method *init* and the constructors of the mother classes. The other two constructors have either an existing ConstraintCompare or InfoCompare object as reference and initialize the object accordingly. The destructor and the method *clear* are empty methods. The method *init* sets the attribute *cv_for_var* to NULL. The method *copy* copies the contents of another ConstraintCompare object to the current one. The = operator is used for assignment, on the left side either a ConstraintCompare or a Constraint (cf. Section 2.2.4.7) object can appear. The == operator is not implemented. The methods *read* and *write* handle input/output from/to files. The methods *clear*, *copy*, *read*, and *write* have a boolean parameter which determines whether the inherited attributes should be processed too or not. For the attribute *cv_for_var* exist trivial set and get methods. The method *set_name* sets not only the *name* for the Constraint (cf. Section 2.2.4.7), but also the name for the linguistic variable *cv_for_var*, which forms the Constraint (cf. Section 2.2.4.7).

The method *set_var_name* sets the name of the linguistic variable *cv_for_var*. The methods *set_var_dil* and *set_op* enable setting the dilatation type used for the linguistic variable and the compare operator applied in the Constraint. The last two methods also use the *generate_ruleset* method to adapt to the rules to the new operator or constraint variable respectively after the old rules for the Constraint have been removed. This is simply done by using the corresponding (cf. Section 2.2.3.8) object for the operator and dilatation type, where for each entry in the table consisting of an input and output value a rule is generated. So, if the *cv_for_var* variable has a specific linguistic term as value then the *cv_for_const* variable is set to the linguistic term value that corresponds to the input value found in the table. The method *make_instance* assigns a CombinedVariable object to the *cv_for_var* variable, where the number of linguistic terms is determined by the compare operator and dilatation type used. The method *bind_to_flip* connects the ConstraintCompare object, which is already bound to a SetOfConstraints (cf. Section 2.2.4.12) object, to the FLIP++ library by first connecting via the inherited Constraint (cf. Section 2.2.4.7) object and then assigning the linguistic variable *cv_for_var*. The appropriate variable is found by the name of the linguistic variable used in the constraint by searching in the rules of the SetOfConstraints object.

2.2.4.9 ConstraintConcat

This class is derived from the classes Constraint (cf. Section 2.2.4.7) and InfoConcat (cf. Section 2.2.4.4) and represents a concat constraint (e.g. temperature_constraint AND humiditity_constraint).

The default constructor calls the method *init* and the constructors of the mother classes. The other two constructors have either an existing ConstraintConcat or InfoConcat object as reference and initialize the object accordingly. The destructor, the method *clear*, and the method *init* are empty methods. The method *copy* copies the contents of another ConstraintConcat object to the current one. The = operator is used for assignment, on the left side either a ConstraintConcat or a Constraint object can appear. The == operator is not implemented. The methods *read* and *write* handle input/output from/to files. The methods *clear*, *copy*, *read*, and *write* have a boolean parameter which determines whether the inherited attributes should be processed too or not. The methods *set_const1* and *set_const2* assign the two constraints used in the concatenation, whereas the method *set_op* defines the concat operator used in the Constraint.

The previous three methods also use the *generate_ruleset* method to adapt the rules to the new operator or constraints respectively after the old rules concerning the Constraint have been removed. This is simply done by using the corresponding (cf. Section 2.2.3.9) object for the operator and dilatation type, where for each entry in the table consisting of two input and one output value a rule is generated. So, if the two constraint variables have specific linguistic terms as value then the *cv_for_const* variable is set to the linguistic term value that corresponds to these input values found in the table. The method *make_instance* assigns Constraints to the constraint variables and adapts the rules correspondingly. The method *bind_to_flip* connects the ConstraintConcat object, which is already bound to a SetOfConstraints (cf. Section 2.2.4.12) object, to the FLIP++ library by connecting via the inherited Constraint (cf. Section 2.2.4.7) object.

2.2.4.10 ListOfConstraints

This class collects several objects of the above described Constraint (cf. Section 2.2.4.7), ConstraintCompare (cf. Section 2.2.4.8), and ConstraintConcat (cf. Section 2.2.4.9) classes. It is derived from the generic class GenList (cf. Section 2.2.2.2).

The default constructor generates an empty list. The other two constructors generate a list with the contents copied from the ListOfConstraints object specified in the parameter. The parameter is either a reference to the ListOfConstraints object or a reference to a SledForConstraints (cf. Section 2.2.4.11) which is associated to a ListOfConstraints object. The destructor deletes all elements of the list. The method *del* deletes an item of the list. The method *key* returns the name of the referenced Constraintas key. The method *look* compares a string with the name of the referenced Constraint. The method *cmp* compares two referenced Constraints for identity. The method *cpy* copies one Constraint object into another. The method *rd* reads a Constraint (cf. Section 2.2.4.7) from a file into an item. The method *wr* writes a Constraint to a file. The method *insert* inserts an Constraint object into the list and in case of the concat constraint checks whether the underlying constraints are existent. The method *kick_out* with a node parameter deletes all appearances of a constraint in other concat constraints of the list prior to deleting the constraint itself. The = and == operators, and the *weak_equal*, *add*, *kick_out*, and *look_up* methods use the inherited methods of GenList (cf. Section 2.2.2.2) and operate in the same way.

2.2.4.11 SledForConstraints

This class does exactly the same as the generic GenSled (cf. Section 2.2.2.3) class, except that the parameters used and the results returned are Constraints (cf. Section 2.2.4.7) pointers instead of void pointers, which is necessary for this type of list. The class operates as sled for the ListOfConstraints (cf. Section 2.2.4.10) class.

2.2.4.12 SetOfConstraints

This class is derived from the ListOfConstraints (cf. Section 2.2.4.10). It is the master class combining Constraints (cf. Section 2.2.4.7) with Tables (cf. Section 2.2.3.7), RuleSets and ParameterSets. The attributes are the *name* of the SetOfConstraints, *correspondings_for_compare* and *correspondings_for_concat*, which are the Tables, *standard_ps*, which is a standard ParameterSet used for the generation of linguistic variables (CombinedVariables), and *rules*, which is a RuleSet and hence a connection to the FLIP++ libraries.

The default constructor initializes an empty set with a default name. Alternatively a SetOfConstraints can be instantiated as a copy from an existing SetOfConstraints object. The destructor is an empty method. The method *clear* is used to empty the ListOfConstraints (cf. Section 2.2.4.10). The method *copy* copies the values from an existing SetOfConstraints object to the current one. The method *insert* inserts a new Constraint (cf. Section 2.2.4.7) into the set. The operators = and == are used for assignment and comparison. The methods *read* and *write* handle input/output from/to files.

There are also simple get methods for the retrieval of *name*, the Tables (cf. Section 2.2.3.7), Rules, and ParameterSet. Furthermore an aggregate *set* method enables setting the *name*, Tables, and the ParameterSet. The method *set_name* sets the name of the SetOfConstraints. The method *get_lowest_level* looks recursivley for the lowest occurence of a Constraint (cf. Section 2.2.4.7) in a hierarchy of Constraints (cf. Section 2.2.4.7) and returns a list of all Constraints (cf. Section 2.2.4.7) visited when descending down the Constraints (cf. Section 2.2.4.7) tree. The method *get_value* retrieves the crisp value of the linguistic variable contained in a ConstraintCompare (cf. Section 2.2.4.8) object, whereas the method *set_value* reversely sets the value of a linguistic variable contained in a ConstraintCompare object.

Hence this method instantiates the linguistic terms of the linguistic variable according to the value specified. The method *evaluate* starts the evaluation of a Constraint (cf. Section 2.2.4.7) by evaluating the corresponding constraint variable with the use of the FLIP++ libraries.

2.2.5 cflp_dyn.h / cflp_dyn.cpp

- EvalConstraint (cf. Section 2.2.5.1)

- ListOfEvalConstraints (cf. Section 2.2.5.2)

- SledForEvalConstraints (cf. Section 2.2.5.3)

- SetOfEvalConstraints (cf. Section 2.2.5.4)

2.2.5.1 EvalConstraint

This class is used for storing the results of the evaluation of a Constraint (cf. Section 2.2.4.7). This enables the user to combine evaluation results by aggregation or to defuzzy them, or to reuse them in other computations.

The attributes are *name*, which is the name of the EvalConstraint, *cv_name*, the name of the CombinedVariable used, *result_cv*, the CombinedVariable object that stores the evaluation result, and *importance*, which is the importance attached to the *result_cv*.

The default constructor uses the *init* method to initialize the object. The destructor is empty. The *init* method sets a default *name*, the importance to zero, and the *result_cv* to NULL. The method *clear* deletes the *result_cv* CombinedVariable. The method *copy* copies the contents of another EvalConstraint object to the current one. The = operator is used for assignment, whereas the == comparison operator is not supported. The methods *read* and*write* are used for retrieving and storing the data contents of an object. Simple *get* and *set* methods allow to retrieve and modify the values of the attributes.

2.2.5.2 ListOfEvalConstraints

This class collects several objects of the above described EvalConstraints (cf. Section 2.2.5.1) class. It is derived from the generic class GenList (cf. Section 2.2.2.2).

The default constructor generates an empty list. The other two constructors generate a list with the contents copied from the ListOfEvalConstraints object specified in the parameter. The parameter is either a reference to the ListOfEvalConstraints object or a reference to a SledForEvalConstraints (cf. Section 2.2.5.3) which is associated to a ListOfEvalConstraints object. The destructor deletes all elements of the list.

The method *del* deletes an item of the list. The method *key* returns the name of the referenced EvalConstraint (cf. Section 2.2.5.1) as key. The method *look* compares a string with the name of the referenced EvalConstraint. The method *cmp* compares two referenced EvalConstraints for identity. The method *cpy* copies one EvalConstraint object into another one. The method *rd* reads an EvalConstraint from a file into an item. The method *wr* writes an EvalConstraint to afile. The = and == operators, and the *weak_equal*, *add*, *kick_out*, and *look_up* methods use the inheritedmethods of GenList (cf. Section 2.2.2.2) and operate in the same way.

2.2.5.3 SledForEvalConstraints

This class does exactly the same as the generic GenSled (cf. Section 2.2.2.3) class, except that the parameters used and the results returned are EvalConstraint (cf. Section 2.2.5.1) pointers instead of void pointers, which is necessary for this type oflist. The class operates as sled for the ListOfEvalConstraints (cf. Section 2.2.5.2) class.

2.2.5.4 SetOfEvalConstraints

This class is derived from ListOfEvalConstraints (cf. Section 2.2.5.2) and aggregates a ListOfEvalConstraints together with a Table, a ParameterSet, and a RuleSet. It is used to aggregate and defuzzify EvalConst raints (cf. Section 2.2.5.1). The attributes are *name*, *rules*, *table*, and *ps*.

Both the constructor and the destructor are empty functions. The method *insert* inserts a new EvalConstraint into the list and binds its *result_cv* to the corresponding CombinedVariable in the *rules*. The methods read and write retrieve and store the data contents of the object from/to files.

Simple *get* methods allow us to retrieve the values of the attributes. The method *get_eval_const_store* returns the whole ListOfEvalConstraints object. The method *set_name* sets the name of the object, whereas themethod *set* sets the attributes *tables*, *ps*, and *name*. The method *defuzzify* uses the FLIP++ method *accumulate* to get a defuzzified result of the specified linguistic variable. The method *aggregate* is used to concatenate two EvalConstraints in order to form an overall EvalConstraint. The *rules* and the *ps* are adapted to the newly created constraint.

2.3 Sample code use

This is an illustration of how to apply the ConFLIP++ libraries (cf. Section 2.2, page 48). The following sample code is a by-product of the ConFLIP++ development process and should provide a good overview of how to apply the specific features of the librariers in an appropriate way. Furthermore, it is strongly recommended to study the code sections concurrent with the description of the ConFLIP++ libraries.

```
/********************************************************************/
/* This file is a testing program; it shows all                     */
/* important features of ConFLIP++                                  */
/********************************************************************/

#include <stdlib.h>
```

```
#include <stdio.h>
#include <string.h>
#include <fstream.h>
#include "cflp_dyn.h"

conflip_error_t conflip_error;

/*****************************************************************/
/*   DEMO  : HOW TO USE ConFLIP++                                */
/*****************************************************************/

int main (void)
{
  int i,j;
  cout<<endl<<"WELCOME TO DEMO1"<<endl<<endl;
  ifstream infile;
  ofstream outfile;
```

Initially, some predefined values are read from files: The ListOfTables (cf. Section 2.2.3.10), the ListOfSetsOfLingTerms (cf. Section 2.2.3.5) and the ListOfParameterSets (cf. Section 2.2.3.12). So the system is initialized with default Tables (cf. Section 2.2.3.7) for the operators, default linguistic terms, and default ParameterSets.

```
  /*****************************************************************/
  /* CREATE HELP-ARRAYS                                            */
  /*****************************************************************/
  ListOfTables corrs;
  ListOfSetsOfLingTerms lings;
  ListOfParameterSets paras;

  infile.open("df_lings.cfp",ios::in);
  if(infile.is_open())
  {
    if(!lings.read(infile))
      cout<<conflip_error.get_txt_eval()<<endl;
    infile.close();
  }
  infile.open("df_corr.cfp",ios::in);
  if(infile.is_open())
  {
    if(!corrs.read(infile))
      cout<<conflip_error.get_txt_eval()<<endl;
    infile.close();
  }
  infile.open("df_paras.cfp",ios::in);
  if(infile.is_open())
  {
```

```
    if(!paras.read(infile))
      cout<<conflip_error.get_txt_eval()<<endl;
    infile.close();
  }

  cout<<"corrs : "<<endl;
  corrs.show();
  cout<<"lings : "<<endl;
  lings.show();
  cout<<"paras : "<<endl;
  paras.show();
```

In this code section, we have shown how to create a new SetOfLingTerms (cf. Section 2.2.3.4) and how to modify it using the associated Sled object. Afterwards, a new ParameterSet is created and both the ParameterSet and the SetOfLingTerms are attached to the lists defined above.

```
  /*****************************************************************/
  /* EDIT HELP-ARRAYS                                              */
  /*****************************************************************/

  cout<<"EDIT HELP-ARRAYS"<<endl;

  static name new_vars1[MAX_LING_VARS]={ "EXCELLENT", "FINE", "OK",
  "POOR", "MISERABLE" };

  SetOfLingTerms my_set;
  if(!my_set.set_name_of_set("my_fuzzy_set"))
    cout<<conflip_error.get_txt_eval()<<endl;
  for(i=0;i<5;i++)
    if(!my_set.add(new_vars1[i]))
      cout<<conflip_error.get_txt_eval()<<endl;
  my_set.show();
  SledForLingTerms my_set_sled=my_set;
  my_set_sled[2];
  cout<<my_set_sled()<<endl;
  strcpy(my_set_sled(),"GREAT");

  static char *temp="my_parameters";

  NList namelist;
  CList colorlist;
  colorlist.add(blue);
  colorlist.add(violet);
  ParameterSet my_ps(temp,0.2,0.3,0.4,0.5,1,1,&colorlist,&namelist);

  if(!lings.add(&my_set))
    cout<<conflip_error.get_txt_eval()<<endl;
  if(!paras.add(&my_ps))
```

```
    cout<<conflip_error.get_txt_eval()<<endl;

  cout<<"lings : "<<endl;
  lings.show();
  cout<<"paras : "<<endl;
  paras.show();
```

Now, we can proceed with putting the parts together to a SetOfConstraints (cf. Section 2.2.4.12) object and a (cf. Section 2.2.5.4) object respectively. Furthermore a ListOfInfos (cf. Section 2.2.4.5) is created that will be used to store information about constraints.

```
  /*****************************************************************/
  /* CREATE ListOfInfos, SetOfConstraints and SetOfEvalConstraints    */
  /*****************************************************************/

  cout<<"CREATE SetOfConstraints"<<endl;

  ListOfInfos infos;
  SetOfConstraints consts;
  if(!consts.set("my_consts",
 (TableCompare *)corrs.lookup(DEFAULT_COMP),
 (TableConcat *)corrs.lookup(DEFAULT_CONC),
 paras.lookup(DEFAULT_PARA)))
    cout<<conflip_error.get_txt_eval()<<endl;

  SetOfEvalConstraints evals;
  if(!evals.set("my_eval",
 (TableConcat *)corrs.lookup(DEFAULT_CONC),
 paras.lookup(DEFAULT_PARA)))
    cout<<conflip_error.get_txt_eval()<<endl;
```

In ConFlip++ each constraint carries some information which is explicitly stored in the Info (cf. Section 2.2.4.2) objects. In our example, we use some constraints concerning chemical compatibilities, which are relevant for the process of steelmaking. For example, the first constraint means that the aluminium value (a detailed explanation of this value is not relevant for this documentation) should be smaller than 0.08. This is a typical ConstraintCompare (cf. Section 2.2.4.8). On the other hand, the ConstraintConcat (cf. Section 2.2.4.9) is the concatenation of two other constraints and therefore implicitly generates a hierarchy of constraints. For instance, the *alu_alu* constraint is a concatenation of the *alu1* and *alu2* constraints. Moreover, it is possible to combine constraints with different dilatation types: the concatenation of a *fuzzy* and *crisp* constraint result in a *mixed* constraint as it is the case with the alus_ni1 constraint.

```
/*****************************************************************/
/* CREATE INFOS                                                  */
/*****************************************************************/

cout<<"CREATE Infos"<<endl;

InfoCompare alu1;
if(!alu1.set("alu-c1", 0.9, fuzzy, "", "alu", fuzzy, smaller, 0.08))
  cout<<conflip_error.get_txt_eval()<<endl;
if(!infos.add(&alu1))
  cout<<conflip_error.get_txt_eval()<<endl;

InfoCompare alu2;
if(!alu2.set("alu-c2", 0.2, fuzzy, "", "alu",fuzzy, equal, 0.08))
  cout<<conflip_error.get_txt_eval()<<endl;
if(!infos.add(&alu2))
  cout<<conflip_error.get_txt_eval()<<endl;

InfoConcat alu_alu;
if(!alu_alu.set("alu-alu", 0.9, fuzzy, "", alu1.get_name(),
alu2.get_name(), c_and))
  cout<<conflip_error.get_txt_eval()<<endl;
if(!infos.add(&alu_alu))
  cout<<conflip_error.get_txt_eval()<<endl;

InfoCompare ni1;
if(!ni1.set("ni-c1", 0.4, crisp, "", "ni", crisp, bigger, 0.4))
  cout<<conflip_error.get_txt_eval()<<endl;
if(!infos.add(&ni1))
  cout<<conflip_error.get_txt_eval()<<endl;

InfoCompare ni2;
if(!ni2.set("ni-c2", 0.4, crisp, "", "ni", crisp, not_equal, 0.4))
  cout<<conflip_error.get_txt_eval()<<endl;
if(!infos.add(&ni2))
  cout<<conflip_error.get_txt_eval()<<endl;

InfoConcat ni_ni;
if(!ni_ni.set("ni-ni", 0.23, crisp, "",
ni1.get_name(),ni2.get_name(), c_or))
  cout<<conflip_error.get_txt_eval()<<endl;
if(!infos.add(&ni_ni))
  cout<<conflip_error.get_txt_eval()<<endl;

InfoConcat alus_ni1;
if(!alus_ni1.set("alus-ni1", 0.3, mixed, "",
alu_alu.get_name(),ni1.get_name(), c_and))
  cout<<conflip_error.get_txt_eval()<<endl;
if(!infos.add(&alus_ni1))
```

```
    cout<<conflip_error.get_txt_eval()<<endl;

  InfoConcat alus_ni1_alu2;
  if(!alus_ni1_alu2.set("alus-ni1-alu2", 0.8, mixed, "",
  alus_ni1.get_name(), alu2.get_name(), c_and))
    cout<<conflip_error.get_txt_eval()<<endl;
  if(!infos.add(&alus_ni1_alu2))
    cout<<conflip_error.get_txt_eval()<<endl;

  cout<<"infos : "<<endl;
  infos.show();
```

With the infos created above it is now possible to fill the SetOfConstraints (cf. Section 2.2.4.12) object with the constraints by using the Info objects. At this time also the rules that correspond to the constraint definitions are generated by using the corresponding Tables (cf. Section 2.2.3.7). The generation of rules is a part of the FLIP++ libraries.

```
  /*****************************************************************/
  /* PUT INFOS INTO SETOFCONSTRAINTS                               */
  /*****************************************************************/

  cout<<"PUT INFOS INTO SetOfConstraints - create everything that does
  not exist"<<endl;
  ConstraintCompare hlpcomp;
  ConstraintConcat hlpconc;

  hlpcomp=(ConstraintCompare)ni1;
  if(!consts.add(&hlpcomp))
    cout<<conflip_error.get_txt_eval()<<endl;
  hlpcomp=(ConstraintCompare)alu1;
  if(!consts.add(&hlpcomp))
    cout<<conflip_error.get_txt_eval()<<endl;
  hlpcomp=(ConstraintCompare)alu2;
  if(!consts.add(&hlpcomp))
    cout<<conflip_error.get_txt_eval()<<endl;
  hlpconc=(ConstraintConcat)alu_alu;
  if(!consts.add(&hlpconc))
    cout<<conflip_error.get_txt_eval()<<endl;
  hlpcomp=(ConstraintCompare)ni2;
  if(!consts.add(&hlpcomp))
    cout<<conflip_error.get_txt_eval()<<endl;
  hlpconc=(ConstraintConcat)ni_ni;
  if(!consts.add(&hlpconc))
    cout<<conflip_error.get_txt_eval()<<endl;
  hlpconc=(ConstraintConcat)alus_ni1;
  if(!consts.add(&hlpconc))
    cout<<conflip_error.get_txt_eval()<<endl;
```

```
hlpconc=(ConstraintConcat)alus_ni1_alu2;
if(!consts.add(&hlpconc))
  cout<<conflip_error.get_txt_eval()<<endl;

cout<<"consts : "<<endl;
consts.show();
RuleSet *rul_help1=consts.get_rules();
rul_help1->show();
```

This code section shows how to modify a constraint or more specifically how to change its name and importance.

```
/*****************************************************************/
/* EDIT CONSTRAINTS                                              */
/*****************************************************************/

cout<<"EDIT Constraints"<<endl;

ConstraintCompare *coco;
if((coco=(ConstraintCompare *)consts.lookup("alu-c1"))==NULL)
  cout<<conflip_error.get_txt_eval()<<endl;
if(!coco->ConstraintCompare::set_name("ALUMINIUM"))
  cout<<conflip_error.get_txt_eval()<<endl;

cout<<"consts : "<<endl;
consts.show();

if(!coco->set_importance (0.99))
  cout<<conflip_error.get_txt_eval()<<endl;

ConstraintCompare *con;
if((con=(ConstraintCompare *)consts.lookup("ALUMINIUM"))==NULL)
  cout<<conflip_error.get_txt_eval()<<endl;
cout<<"new importance : "<< con->get_importance()<<endl;
```

Next comes the evaluation part. The constraint to be evalutated is the alus-ni1 constraint, which was defined above. Before the evaluation is started it is necessary to specify some input values. These input values are all the ConstraintCompares (cf. Section 2.2.4.8) that are contained in the constraint tree of the constraint to be evaluated. The method *get_lowest_level* retrieves exactly these constraints for the specified constraint in a ListOfStrings (cf. Section 2.2.3.1) object. Now, these linguistic variables can easily be set to a certain input value. After this, the evaluation can be started and the result is returned in a CombinedVariable. This result can then be stored in an EvalConstraint (cf. Section 2.2.5.1) for further calculations. AnEvalConstraint can also be defuzzified or aggregated with other EvalConstraint. In our example, also the ni-ni constraint is evaluated and its result stored in an

EvalConstraint. At the end, the two EvalConstraint are aggregated and an overall defuzzified result is calculated.

```
/*******************************************************************/
/* EVALUATE constraints to get CombinedVars, create EVALCONSTRAINTS  */
/*******************************************************************/

cout<<"EVALUATE"<<endl;

ListOfStrings names_alus;
SledForStrings mysled;
consts.get_lowest_level("alus-ni1",names_alus);
names_alus.show();
mysled=names_alus;
consts.set_value(mysled++,40.0);
consts.set_value(mysled++,-70.0);
consts.set_value(mysled++,80.0);

CombinedVariable *sat_cv_alus= new CombinedVariable;
sat_cv_alus = consts.evaluate("alus-ni1");

sat_cv_alus->show();
EvalConstraint ec1;
if(!ec1.set("eval1",sat_cv_alus))
  cout<<conflip_error.get_txt_eval()<<endl;
if(!evals.add(&ec1))
  cout<<conflip_error.get_txt_eval()<<endl;

float s1=evals.defuzzify("eval1");
cout<<"defuzzyfied value  for alus-ni1 :"<<s1<<endl;

ListOfStrings names_ni;
consts.get_lowest_level("ni-ni",names_ni);
names_ni.show();
mysled=names_ni;
consts.set_value(mysled++,-20.0);
consts.set_value(mysled++,50.0);
CombinedVariable *sat_cv_ni = new CombinedVariable;
sat_cv_ni=consts.evaluate("ni-ni");
sat_cv_ni->show();

EvalConstraint ec2;
if(!ec2.set("eval2",sat_cv_ni))
  cout<<conflip_error.get_txt_eval()<<endl;
if(!evals.add(&ec2))
  cout<<conflip_error.get_txt_eval()<<endl;

float s3=evals.defuzzify("eval2");
cout<<"defuzzyfied value  for ni-ni :"<<s3<<endl;
```

```
cout<<"aggregate"<<endl;

evals.aggregate("all", &ec1, &ec2);

RuleSet* rul_help=evals.get_rules();
rul_help->show();
evals.show();

float s2=evals.defuzzify("all");
cout<<"defuzzyfied value for  all :"<<s2<<endl;
```

This code section simply shows how infos, constraints, and evaluated constraints are written in a file for later usage.

```
/*****************************************************************/
/* WRITE                                                         */
/*****************************************************************/

cout<<"WRITE"<<endl;

outfile.open("t_infos.cfp",ios::out|ios::trunc);
if(outfile.is_open())
{
  if(!infos.write(outfile))
    cout<<conflip_error.get_txt_eval()<<endl;
  outfile.close();
}
outfile.open("t_consts.cfp",ios::out|ios::trunc);
if(outfile.is_open())
{
  if(!consts.write(outfile))
    cout<<conflip_error.get_txt_eval()<<endl;
  outfile.close();
}
outfile.open("t_evals.cfp",ios::out|ios::trunc);
if(outfile.is_open())
{
  if(!evals.write(outfile))
    cout<<conflip_error.get_txt_eval()<<endl;
  outfile.close();
}
```

Kick_out methods are used to remove items from a list object, which is illustrated below.

```
/*****************************************************************/
/* KICK OUT                                                      */
/*****************************************************************/
```

```
cout<<"KICK OUT"<<endl;

if(!infos.kick_out("alu-c1"))
if(!consts.kick_out("ALUMINIUM"))
if(!lings.kick_out("my_fuzzy_set"))

cout<<"infos : "<<endl;
infos.show ();
cout<<"consts : "<<endl;
consts.show();
cout<<"lings : "<<endl;
lings.show();
```

Similarly the write methods and the *read* methods are used to retrieve previously used and calculated data from files.

```
/********************************************************************/
/* READ                                                             */
/********************************************************************/

cout<<"READ"<<endl;

ListOfInfos new_infos;
SetOfConstraints new_consts;
SetOfEvalConstraints new_evals;

infile.open("t_infos.cfp",ios::in);
if(infile.is_open())
{
  if(!new_infos.read(infile))
    cout<<conflip_error.get_txt_eval()<<endl;
  infile.close();
}
infile.open("t_consts.cfp",ios::in);
if(infile.is_open())
{
  if(!new_consts.read(infile))
    cout<<conflip_error.get_txt_eval()<<endl;
  infile.close();
}
infile.open("t_evals.cfp",ios::in);
if(infile.is_open())
{
  if(!new_evals.read(infile))
    cout<<conflip_error.get_txt_eval()<<endl;
  infile.close();
}
```

```
cout<<"new infos : "<<endl;
new_infos.show ();
cout<<"new consts : "<<endl;
new_consts.show();
cout<<"new evals : "<<endl;
new_evals.show();
```

Finally, the just read data is used to retrieve some objects from it. This demonstrates the flexibility of ConFLIP++ as far as reusing already built or calcualted structures for new computations are concerned.

```
/******************************************************************/
/* FINAL TEST                                                     */
/******************************************************************/

cout<<"FINAL TEST"<<endl;

InfoCompare *c1=(InfoCompare *)new_infos.lookup("alu-c2");
if(c1!=NULL)
  cout<<"1. name: "<<c1->get_name()<<"    dil: "
    <<(int)c1->get_dilatation()<<endl;

ConstraintCompare *co=(ConstraintCompare *)new_consts.lookup("ni-c1");
if(co!=NULL)
  cout<<"2. name: "<<co->get_name()<<"    var: "
    <<co->get_var_name()<< "   dil : " <<(int)co->get_dilatation()
    <<endl;

ConstraintConcat *cc=(ConstraintConcat *)new_consts.lookup("alu-alu");
if(cc!=NULL)
  cout<<"3. name: "<<cc->get_name()<<"     1.const: "
    <<cc->get_const1()<<"  dil: "<<(int)cc->get_dilatation()<<endl;

ConstraintConcat *cc1=(ConstraintConcat *)new_consts.lookup("alus-ni1");
if(cc1!=NULL)
  cout<<"4. name: "<<cc1->get_name()<<"    1.const: "
    <<cc1->get_const1()<<"  dil: "<<(int)cc1->get_dilatation()<<endl;

/******************************************************************/
/* END                                                            */
/******************************************************************/

cout<<endl<<"DEMO1 SUCCESSFULLY RUN - thank you, see you"<<endl;

/******************************************************************/
/* CLEAN-UP                                                       */
/******************************************************************/

return(1);
```

```
}
```

2.4 ConFLIP++ User Interface

In the following we will use a simple example, which is actually identical to the one used in the ConFLIP++ Sample Code (cf. Section 2.3, page 69) part of DocuFLIP++. Although very simple and not too realistic it should give you a short impression of how to use the ConFLIP++ user interface. It is also recommended, although not necessary, to go through the documentation of FLIP++ in DocuFLIP++. There are many similarities between ConFLIP++ and FLIP++ as ConFLIP++ is based on FLIP++.

One of the features of ConFLIP++ is that is goes a little bit farther than FLIP++ when it comes to the representation of constraints. Whereas in FLIP++ you simply define linguistic variables with their linguistic terms and the rules that show the interrelationship between the variables, ConFLIP++ also allows a direct constraint definition and then generates the appropriate rules automatically. Generally speaking, the constraints can be divided into two groups: The compare constraints and the concat constraints. The former allows us to compare a linguistic variable with a specific value and to evaluate how well this comparison is satisfied, the latter constraint type is an aggregation of two other constraints. One can easily see, that the concat constraint hence enables us to build a hierarchy of constraints or put in another way, a constraints tree. This is especially useful when you have a system where a specific order is inherent and where some constraints are underlying constraints for various other constraints.

Similarly to the ConFLIP++ Sample Code (cf. Section 2.3, page 69) we build our example system by using some predefined parameters and then create the hierarchy of constraints that refer to chemical compatibilities, which are relevant to the process of steelmaking.

2.4.1 Main application window

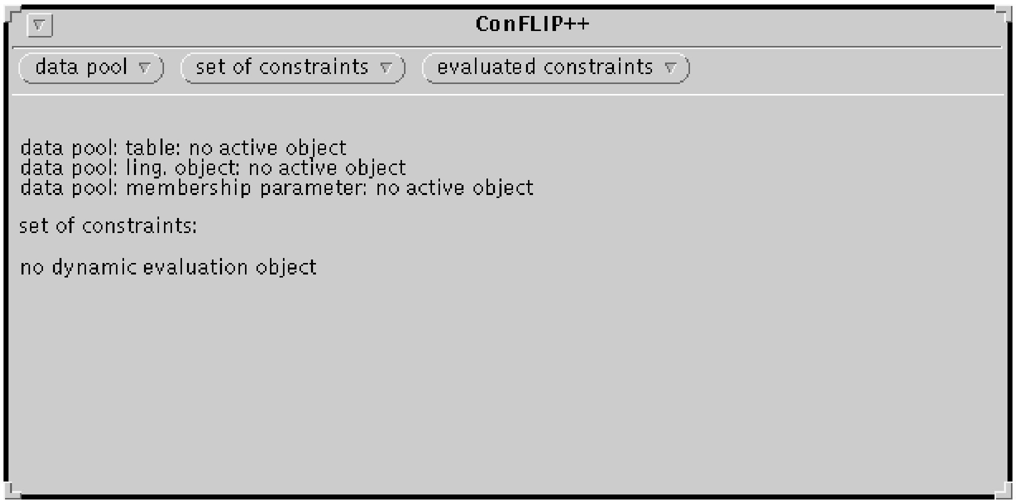
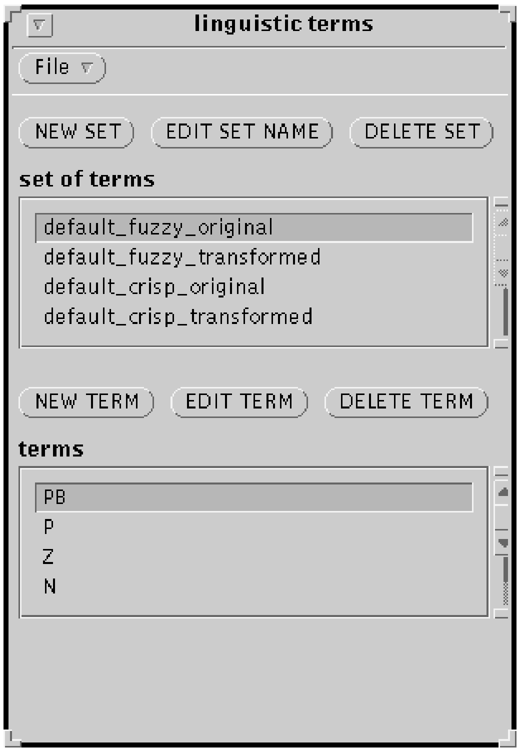


Figure 2.2: ConFLIP++ Main Application Window

Initially, we find ourselves in the main application window. It shows the states of the data pool, which is formed by the Table (cf. Section 2.2.3.7) that defines the operators functionality with the linguistic terms used by the linguistic variables and the ParameterSet defining the shape of membership functions. Also, the set of constraints that are currently defined is displayed.

2.4.2 How to define linguistic terms

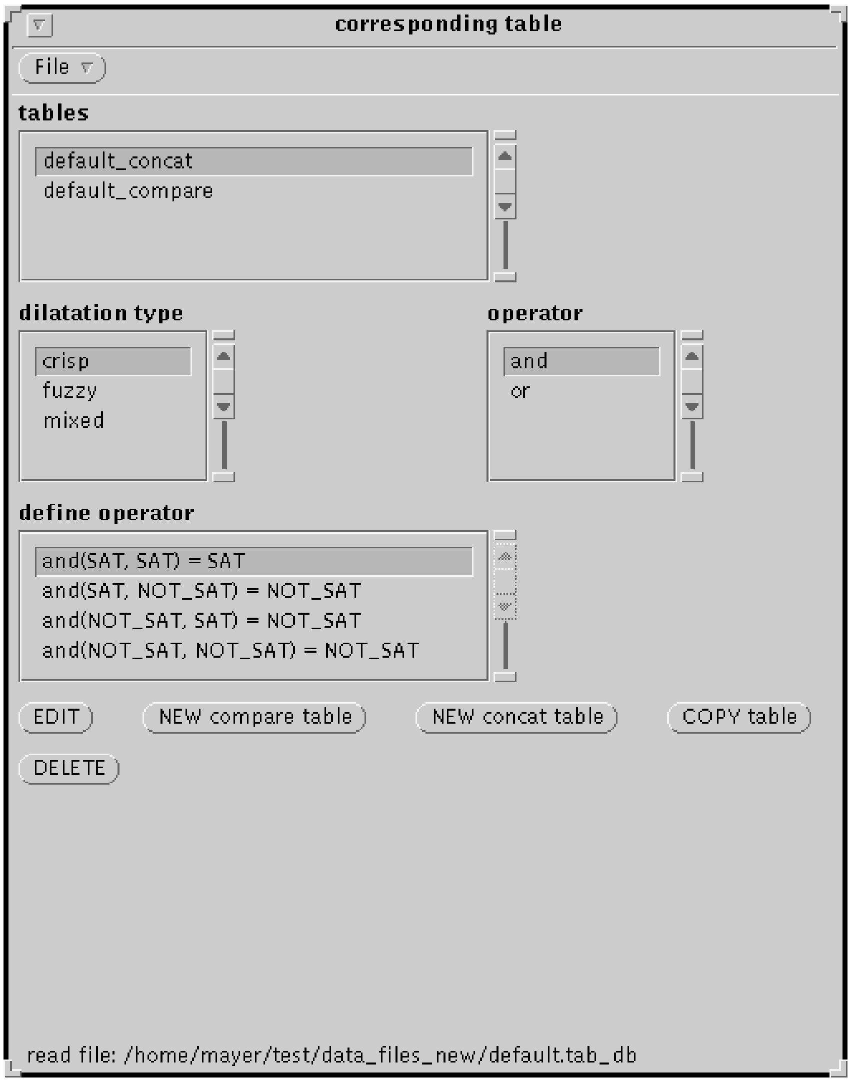


[3]

Figure 2.3: How to define linguistic terms

First, we start with the linguistic terms after selecting the item *lingusitc terms* in the menu *data pool* the *linguistic terms* window is displayed on the front. Here new linguistic variables (set of terms) and linguistic terms can be added, edited, and deleted. Furthermore it is possible to read and write from/to a file.

2.4.3 How to define operator functionality



4

Figure 2.4: How to define operator functionality

Now we proceed with specifying a table (cf. Section 2.2.3.7), which is necessary in order to tell ConFLIP++ how an operator should be applied to input values. Having selected the item *tables* in the *data pool* window brings us to the *corresponding table* window. There, a list of tables (cf. Section 2.2.3.7) is displayed. Moreover, for the currently selected table the dilatation types, operators, and functionality of the selected operator for the selected dilatation are shown. Tables have to specified for both types of constraints.

Besides, we must distinguish between a fuzzy and crisp input/output value in the case of compare constraints. With concat constraints it is also possible to have mixed values because there are always two input parameters, i.e. the two constraints that are to be concatenated. To create a new table (cf. Section 2.2.3.7) one either chooses the *NEW compare table* or the *NEW concat table* button at the bottom of the window. Then a windows prompts for the name of the table and the particular fuzzy/crisp/mixed input and output terms, which are the sets of linguistic terms defined in the previous paragraph.

2.4.4 How to define sets of parameters

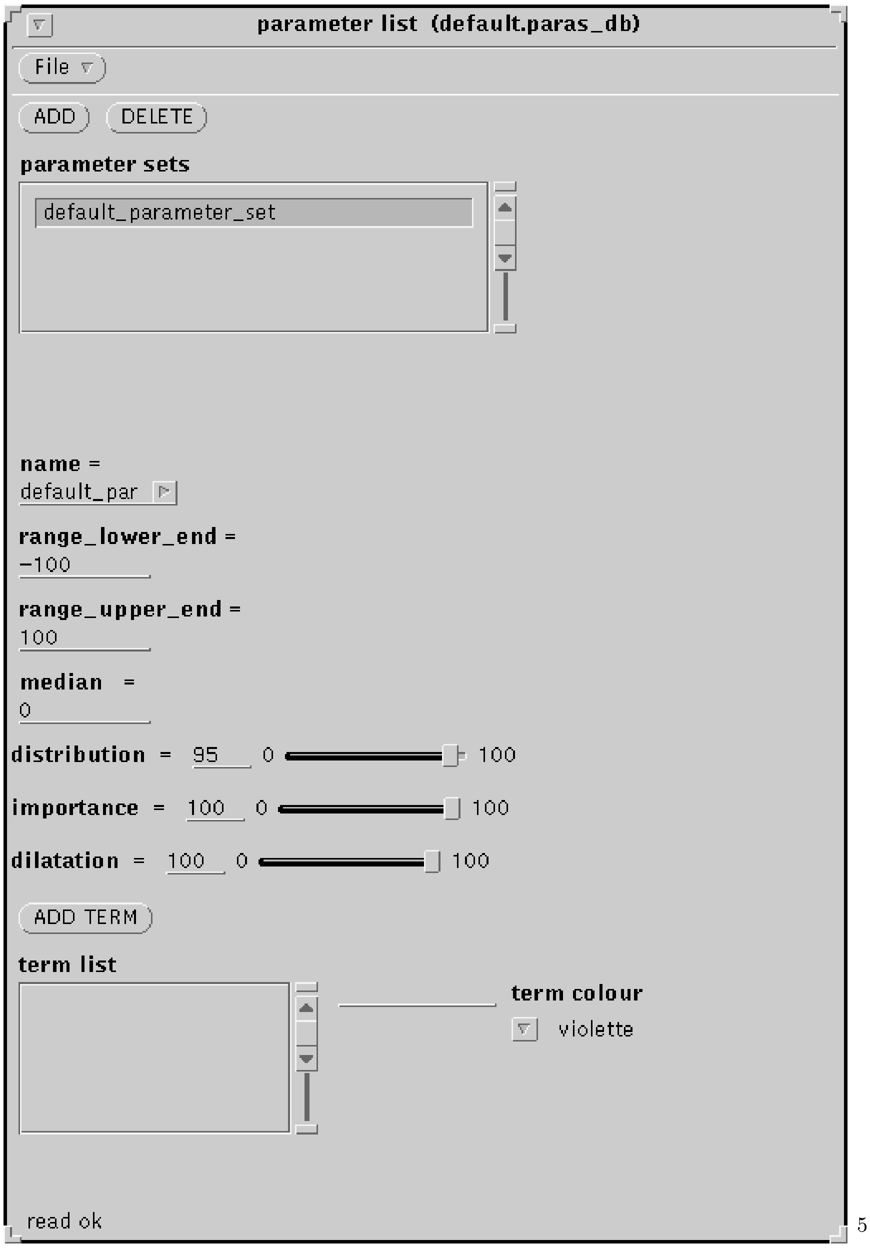


Figure 2.5: How to define sets of parameters

In order to specify the shape of the membership functions of linguistic variables it is also necessary to have paramter sets that describe the functions. A parameter set can be created by clicking the item *membership parameters* of the *data pool* window. The window that appears on the screen is exaclty the same is in the FLIP++ User Interface. First a list of parameter sets is displayed, then the parameters of the currently selected set. The values of the parameter set are identical to the ones used in the ConFLIP++ Sample Code (cf. Section 2.3, page 69). These parameters are used as default values for the linguistic variables that are created for the representation of a constraint. So, contrary to FLIP++, where linguistic variables are created explicitly by the user, in ConFLIP++ the user creates constraints and the system makes implicitly linguistic variables that are attached to these constraints.

2.4.5 How to define constraint information

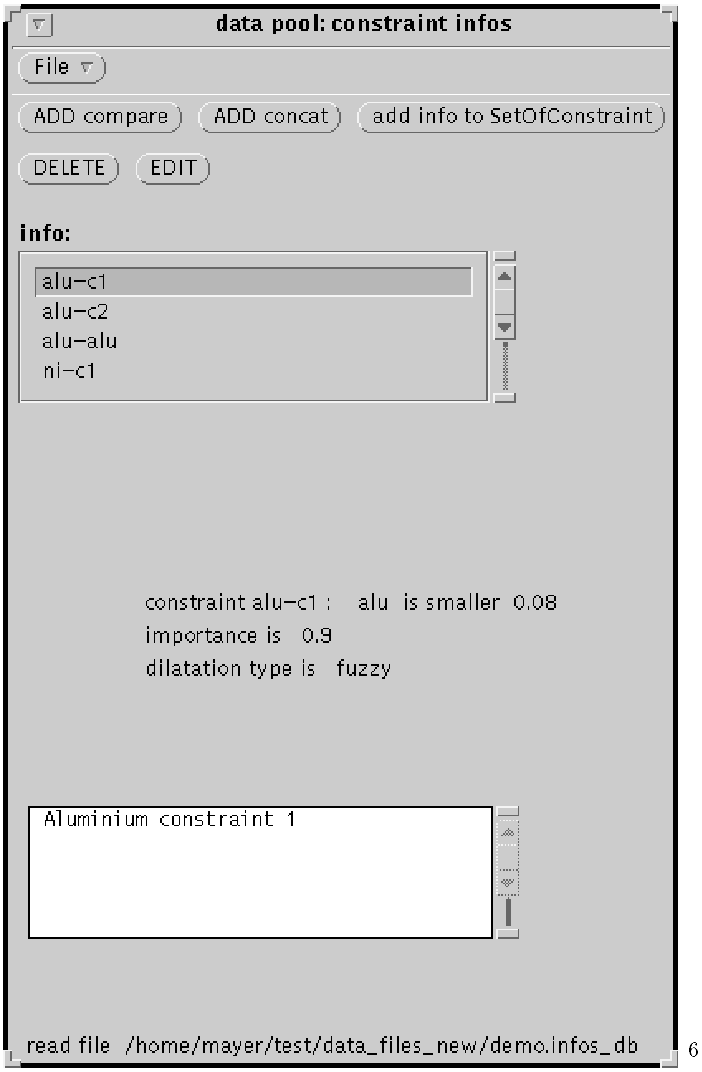


Figure 2.6: How to define constraint information

The last item of the *data pool* menu are the *constraint infos*, these are the descriptions of the constraints. In the specific window you can either choose to add a compare or a concat constraint. The creation is rather straightforward, you just have to enter the particular parameters of the constraints. In our example, we created exactly the same infos for the constraints as in the ConFLIP++ sample code (cf. Section 2.3, page 69). Furthermore, the *add info to SetOfConstraint* button enables the user to attach an info to certain, already defined, set of constraints.

2.4.6 How to define constraints

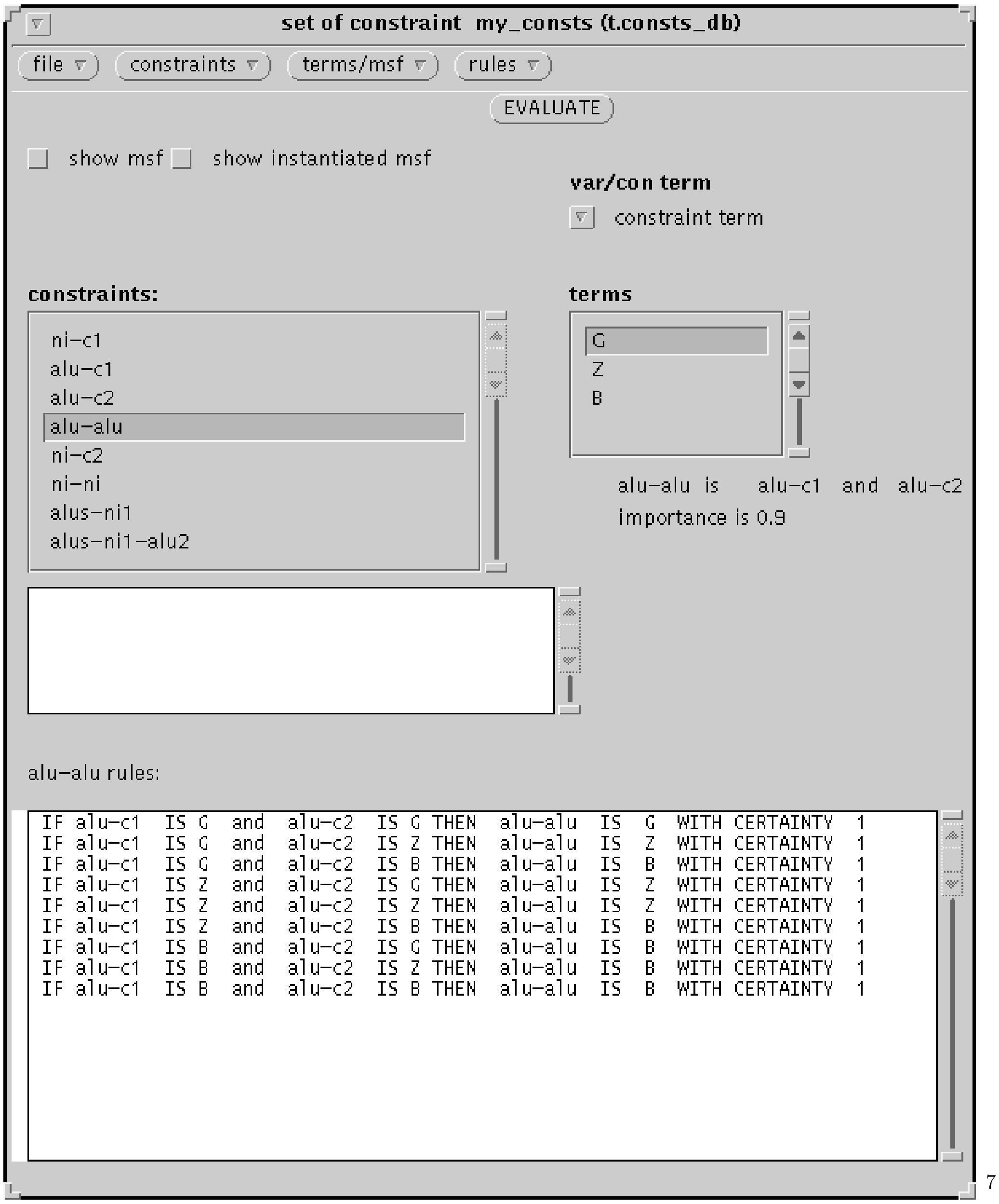


Figure 2.7: How to define constraints

The menu *set of constraints* is devoted to the handling of the constraints. Until now, we have only set the framework for the constraints. Now if we want to create them, we can either load already defined constraints from a file, or we create new ones by using the items *NEW* or *NEW with defaults.*

Furthermore, an existing set of constrains can be edited when clicking the item *constraint handling*. All these options lead to the same window, which is almost exactly the same as the FLIP++ user interface. For that reason we will be refering to the description given for any particulars.

Roughly speaking, the main differences to the FLIP user interface are: instead of linguistic variables you create constraints (either concat or compare), internally these are in turn represented as linguistic variables (CombinedVariable objects). Furthermore, it is possible to derive constraints directly from the definition of the constraint infos with the item *NEW info from container*. Of course the created constraints can be edited and deleted. The menus *terms/msf* and *rules* behave in the same way as in the FLIP++ user interface. Also, slightly different from FLIP++ the switch *var/con term* chooses between either displaying the terms of the variable contained in a compare constraint or the terms of the compare constraint variable itself. For the concat constraint this switch does not have a meaning.

Again, please note that you always get a default set of rules for each constraint, which of course can be modified. One can easily see that in principle the rules created by ConFLIP++ according to the table, parametersets, linguistic terms for a specific constraint could also be modified with FLIP++. But this would mean a lot of tedious work, which can otherwise be done automatically. Modifying the rules generated by ConFLIP++ is just the fine-tuning work that might be necessary to obtain reasonable results.

2.4.7 How to evaluate constraints and to define evaluated constraints

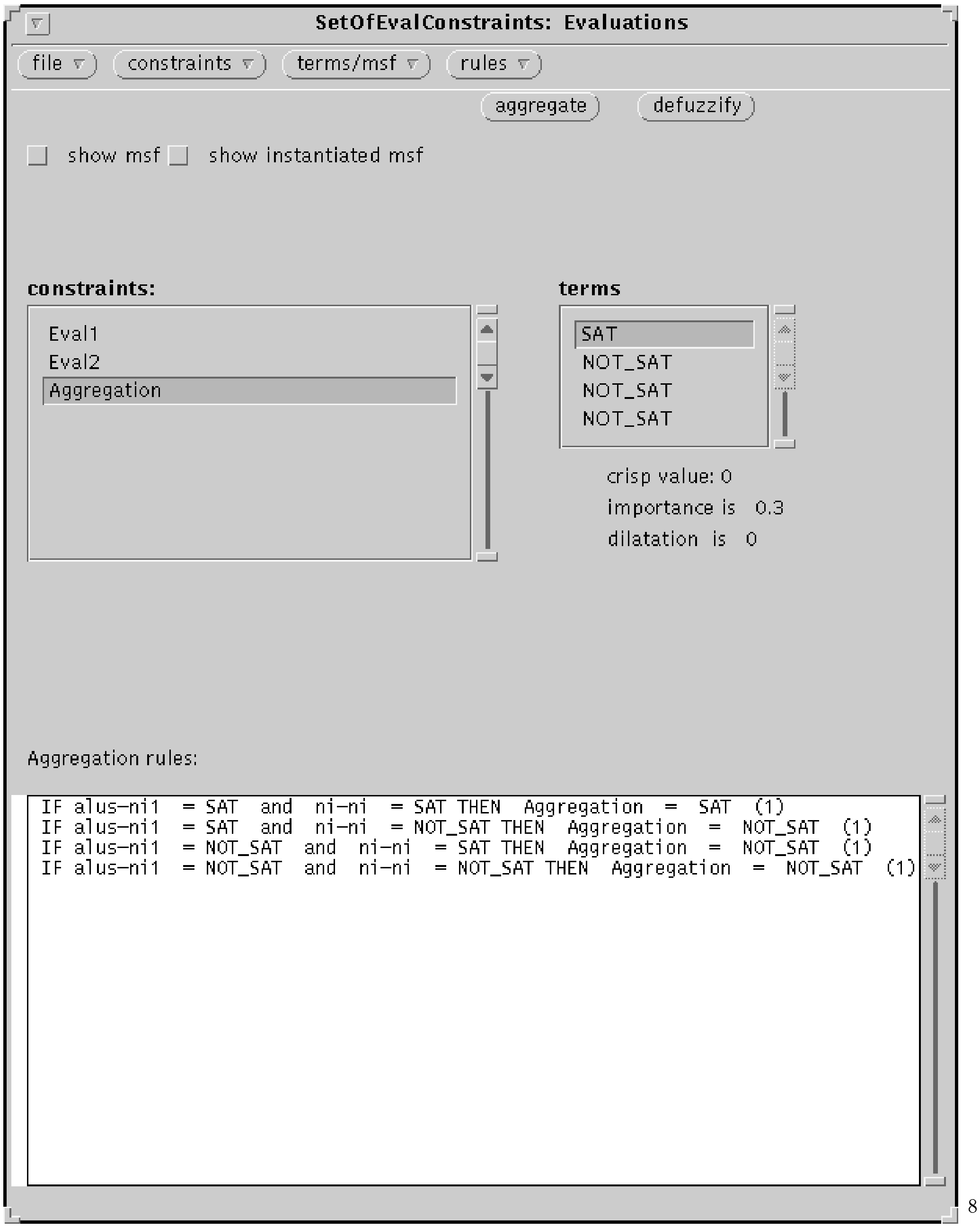
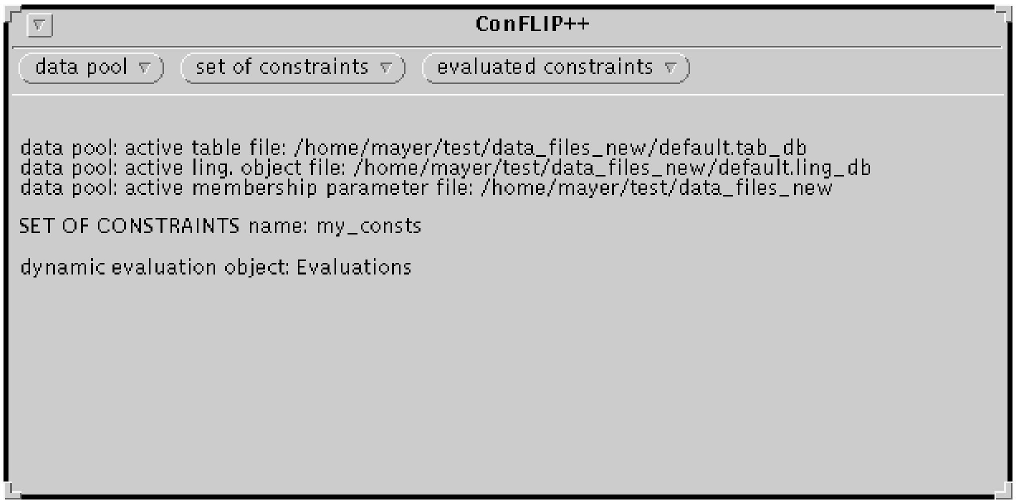


Figure 2.8: How to evaluate constraints and to define evaluated constraints

Now, before proceeding with evaluating a constraint it is necessary to define a set of evaluated constraints, where each evaluation of a constraint will be stored. This is either achieved by choosing the items *READ*and*NEW*, or *NEW with defaults* in the menu *evaluated constraints*. The upcoming window looks similar to the *set of constraints* window. As in the ConFLIP++ Sample Code (cf. Section 2.3, page 69) two evaluations can be aggregated to an overall evaluation, which again shows the hierarchical composition of different evaluations. The other components like terms, rules, and membership functions are handled in the same way as in the *set of constraints* window.

Going back to the *set of constraints* window, we can start the evaluation of a constraint by choosing it in the list of constraints and clicking the *evaluate* button. This brings a window to the screen prompting for the name of the evaluation and the values which can be set for the input variables. Prior to evaluation it is necessary to set values for all the variables used in the underlying constraints which the ConFLIP++ always prompts the user for . Then an evaluated constraint is created, which can be seen in the *evaluated constraints* window.



9

Figure 2.9: The Evaluated Constraints Window

Chapter 3

DomFLIP++

3.1 Knowledge Engineer

DomFLIP++ is one of the central parts of the StarFLIP++ environment. The complete domain knowledge representation is defined in this module. You have also to provide any initial solution for the job to be optimized.

In the following the notions of jobs, aggregates and rules are introduced.

- Job (cf. Section 3.1.1)
- Job example (cf. Section 3.1.2)
- Aggregates (cf. Section 3.1.3)
- Aggregates example (cf. Section 3.1.4)
- Rules (cf. Section 3.1.5)
- Rules example (cf. Section 3.1.6)
- Initial Schedule (cf. Section 3.1.7)

3.1.1 Job

The task at hand is the initial job. This job can of course be sub-divided into ever smaller junks until one reaches a point where further sub-division is no longer possible in a useful manner. The resulting pieces are called **atomic jobs**. It is these **atomic jobs** that are to be scheduled later in a meaningful way. Each of these jobs has different properties like the time it takes or its priority for example.

This decomposition of the task should be done with a lot of deliberation. It is as well important to identify as many stand-alone jobs as possible (better as there are) as it is of no good to divide logical steps that belong together into different jobs just because it is possible.

3.1.2 Example

Our initial job is to produce n tons of steel of thickness t and quality q. Some atomic jobs for this task are:

1. chemical treatment for n tons of quality q

2. rolling of n tons of steel to thickness t

If we could distinguish different steps in the treatment of the chemicals on the assembly line we could of course further sub-divide job 1) though it would be of no use because you cannot change assembly lines in the course of the chemical treatment. In fact this could even prove counter-productive as you would have to introduce some additional rules and constraints later on that would explicitly state that a change of assembly lines is not possible.

3.1.3 Aggregates

An aggregate is an unit (in steel producing usually a machine) on which jobs of the same kind with respect of its use are scheduled. We can again start with a big aggregate that serves us for the whole job and then sub-divide to the extent that we reach smaller aggregates of suitable functionality. These jobs are handled in the same way (e.g. on a paper-cutter) by the aggregate at least for a specific amount of time (though the handling of jobs can also differ if you have as an aggregate a machine that can do different things). It seems clear that there usually will be a rather strong dependence between jobs that are handled on the same aggregate with little time between them. The nearer they are - on the time line - the greater is their dependence i.e. they are treated in a very similar way.

As with jobs you should also take great care while identifying the aggregates. To little leaves you very inflexible while too much could prove counter-productive.

3.1.4 Example

In our steel-making process among other aggregates we could distinguish:

1. Assembly lines (A1 and A2; those are two different aggregates that serve the same function)

2. Rolling mills (R1 and R2; also two different aggregates as above)

As in the job-example we could easily sub-divide one assembly line into different, smaller aggregates with the same result as above - a system that is more complex without giving more flexibility to optimizations.

3.1.5 Rules

After the identification of jobs and aggregates as a next important step we have to define rules. These rules state how jobs (atomic or concatenated ones) or/and aggregates can be interchanged or substituted (if it is not possible to find such rules there is nothing you can optimize). The more rules you can identify the better. Each additional rule adds some more flexibility to your system. An important tag of a rule is how favorable it is to use it.

Correct identification and classification of rules is of utmost importance as they build the foundation for the optimizing cycle.

3.1.6 Example

Some rules for our steel-making example include:

1. An atomic job that was previously scheduled on a rolling mill (R1 or R2) can be rescheduled only on a rolling mill.

2. An atomic job that was previously scheduled on an assembly line (A1 or A2) can be rescheduled only on an assembly line.

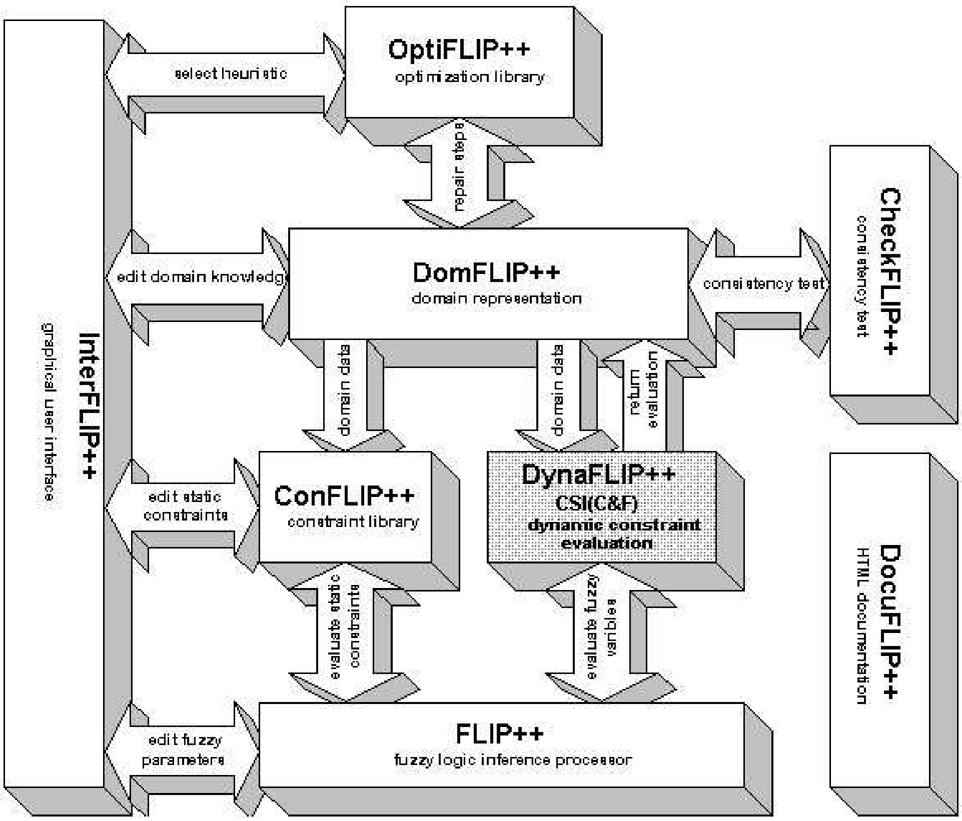
It is important to note that while inadequate dividing of jobs and aggregates could lead to an overly complex or inflexible system that can nevertheless produce results the definition of rules that are wrong will lead to no results in the best case or faulty ones as an alternative.

3.1.7 Initial Schedule

For any optimizations to take place you have to provide the system with an initial schedule as a starting point for its optimizations. The better the schedule is you provide (with regard to constraint violations) the faster the optimization cycle will be finished though a good initial schedule is not obligatory.

At this time of the development of StarFLIP++ DomFLIP++ is the starting point of the system. From here every other function of the environment is evoked.

Every other unit is either incorporated in DomFLIP++ or called upon from this point (see figure 3.1 on page 96).



1

Figure 3.1: Overview on StarFLIP++

Chapter 4

DynaFLIP++

In this page we can find two different terms for one subject: *DynaFLIP++* and *CSI(C&F)*. At this point we would like to remark that DynaFLIP++ stands for all conceptual issues stated in this thesis. CSI(C&F) is a *constraint script interpreter* on *crisp and fuzzy sets*, i.e., the implementation of all objectives and functionality of DynaFLIP++. This linguistic distinction may seem not to make a lot of sense, but it is based on a 'historical' motivation. Several different approaches of implementing DynaFLIP++ have been done, and, finally, CSI(C&F) seemed to be the best solution. Earlier implementations adapting the static constraint library ConFLIP++ (cf. Section 2, page 43) did not perform as expected, and for this CSI(C&F) was developed.

DynaFLIP++ interacts with FLIP++ (cf. Section 1, page 1), the latter being a general purpose fuzzy inference processor library, needed for all computation concerning fuzzification, defuzzification, membership functions, linguistic variables, distributions, etc. DynaFLIP++ wraps additional functionality around these objects and reaches this functionality up to DomFLIP++ (cf. Section 3, page 93) where the actual domain knowledge is processed and optimization is performed. This additional functionality has become necessary because of factors encountered in many different real-world optimization problems, of which we only became aware of after trying to handle two different problems, namely steel production scheduling and classroom roostering. Since special consideration is given to the reusability aspect of all libraries, we found that *static fuzzy constraints* are good to model the aspects associated with partial satisfaction, compromising, and relative importance of constraints. However, other aspects require dynamic generation of constraints from template constraints. For instance, the actual number and kind of constraints often depend upon the current instantiation of the problem that is to be optimized. With each search step in the optimization process, the structure of this instantiation may change. Therefore, the structure of the constraint evaluation tree has also to change. For example, it is possible that a job with an associated delivery date is exchanged with another job that has no delivery date, therefore this constraint must not be evaluated for the second schedule.

Another aspect is that it is normally not useful to tune each constraint separately. Instead, a static constraint is tuned for a selected reference value, and DynaFLIP++ then uses this static constraint to generate a dynamic constraint adapted to the actual situation. Again, looking at the delivery date example, this implies that the template is a static constraint that is tuned around the value zero, with an appropriate fuzzy distribution around it. DynaFLIP++ then specializes this to an actual time in the scheduling horizon, for instance 10 a.m. on April 22, 1996.

This dynamic adaptation is mostly harmless for simple constraints such as delivery dates, but becomes more tricky when complex constraints are involved. A more complex example, also taken from the steelmaking domain, would be the duration of the tundish life expectancy. The problem is that the attributes of a definite number of jobs must be aggregated, in this case by adding their durations, without knowing at the time when the static constraint is defined how many jobs will have to be eventually aggregated. Their number can only be determined dynamically at optimization time by checking certain criteria of the actual schedule instantiation. These aggregation operators are similar to those found in spreadsheet software that process a range of values.

Another difficulty arises because DynaFLIP++ contains absolutely no knowledge about the domain. Using DomFLIP++ (cf. Section 3, page 93), optimization structures are defined together with their associated constraints, using the knowledge-engineering features of DynaFLIP++. Additionally, relations between attributes of items in these optimization structures and the static constraints must be defined using a special rule-based language that is interpreted at evaluation time by DynaFLIP++. A rule specifying when and how a template constraint is used. Such a rule, slightly simplified but taken from the actual steelmaking domain, illustrates this mechanism as given in table 4.1 on page 99.

CONDITION:
$<\text{Job}_i, \text{Job}_{i+1}>$ *ELEMENT OF* CC3 *AND*
quality separation *NOT ELEMENT OF* $<\text{Job}_i,\text{Job}_{i+1}>$ *AND*
tundish change *NOT ELEMENT OF* $<\text{Job}_i,\text{Job}_{i+1}>$ *AND*
CC3 setup *NOT ELEMENT OF* $<\text{Job}_i,\text{Job}_{i+1}>$
CONSTRAINT:
Chemical_compatibility_CC3 $<\text{Job}_i,\text{Job}_{i+1}>$

Table 4.1: A rule specifying when and how a template constraint is used

Here, ¨Chemical_compatibility_CC3 $<\text{Job}_i,\text{Job}_{i+1}>$¨ stands as a macro that links several attribute values of Job_i and Job_{i+1} to linguistic variables defined in the pre-tuned static constraints. For the actual implementation, the whole definition of this macro must be specified. The actual compatibility encompasses 12 more chemical alloying elements, the degassing procedure in the secondary metallurgy aggregates, as well as the casting format between the jobs.

4.1 Evaluating dynamic constraints

To evaluate a given instantiation of a partial constraint satisfaction problem, a decision function aggregating all the constraints with their respective priorities, using an appropriate aggregation operator and a corresponding weighting scheme, must be established. DynaFLIP++ is responsible for efficiently establishing a new global constraint representation for a specific instantiation of the problem. This global constraint will result in a highly structured constraint tree for the whole schedule. The constraint evaluation function will return the weighted global satisfaction score based on the current schedule and the constraint evaluation tree. Figure 4.1 on page 100 outlines the general structure of this dynamically constructed tree, where the nodes are weighted aggregation operators (in the simplest case conjunctions) and the leafs are evaluations of individually fine-tuned constraints, again taken from the steelmaking domain. DynaFLIP++ is able to use most of the framework provided by FLIP++ to efficiently compute the evaluation scores for a new schedule.

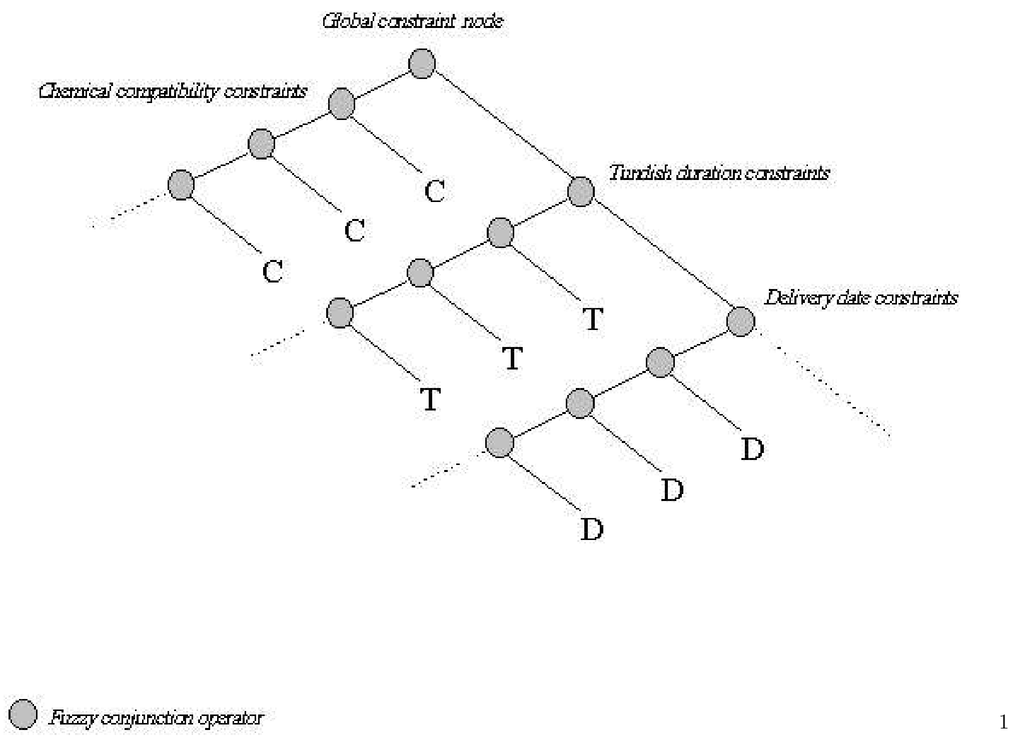


Figure 4.1: Outline of the constraint tree constructed by DynaFLIP++

When scheduling, it is often advisable to introduce an additional measure into the decision function dependent upon whether the current schedule (= instantiation of the constraint satisfaction problem) contains certain difficult jobs. If the scheduling of these jobs is not introduced as a bonus into the decision function, these jobs might never be considered for actual scheduling. There usually exists a non-empty pool of waiting jobs, and only a subset of jobs from the pool can be scheduled immediately. Therefore, the danger is that some difficult jobs will remain in the pool forever unless additional measures are taken. It is clear that this 'difficulty' or *importance* of a job must increase over the time for which it is still reasonable to schedule it, to favourize its eventual scheduling.

The easiest way to introduce this 'difficulty'-measure is to formulate a corresponding constraint with an associated priority that will represent these difficult jobs and which will therefore be represented by another branch of a certain constraint type as in figure 4.1 on page 100. Thus, the 'difficulty' of jobs will be one criteria considered when the partial constraint satisfaction problem is optimized. The same applies equally to other partial constraint satisfaction problems such as those encountered in design or planning.

4.2 Conceptual issues with DynaFLIP++

The DynaFLIP++ library is located between FLIP++ (cf. Section 1, page 1) and the domain knowledge representation library DomFLIP++, (cf. Section 3, page 93) which is a description tool for the environment that has to be optimized. DynaFLIP++ has no restrictions about the complexity of a domain structure, all domain data is imported by the use of CSI(C&F) variables. A description of all CSI(C&F) structures can be found here. In the steelmaking environment a domain holds a list of aggregates, which themselves hold a schedule of objects with their respective attributes and variables. For all variables used within a CSI(C&F)-script one or more constraints can be specified in order to describe relations and restrictions on the process variables. DomFLIP++ is also responsible for repair steps on a badly evaluated schedule, using a list of violations and badly evaluated variables, which are computed at runtime by DynaFLIP++, and the optimization algorithms supported by OptiFLIP++ (cf. Section 6, page 175).

To build up the evaluation tree, DynaFLIP++ has to consider the domain structure with its aggregates and scheduling objects, so that the evaluating tree is built up analogous to the hierarchical structure of the modeled application. The structure of the scheduling objects depends on the application they are designed for. Considering production scheduling in industrial environments as a special CSP, we meet different types of imprecision, stemming from constraints that are blurred in definition and include vagueness and uncertainty. We can imagine that an operation on the schedule may start a 'little' earlier and that 'small' deviations of optimal values may be acceptable. The scheduling object 'Order' then has attributes such as plant, plantmark, throughput, weight, format, thickness, speed, slabgroup, chemical_elements, delivery_date, ... and associated constraints like

domain constraint: out_date delivery_date

In this case, an aggregate could be the continuous caster CC-4, with constraints concerning tundish durations, average throughput, or setup-restrictions. Of course there would exist a variety of other constraints, such as compatibility constraints, capacity constraints, or temporal constraints. To evaluate a schedule for one aggregate, DynaFLIP++ has to first create and instanciate all constraints for the domain objects which are designed by the knowledge engineer. In a second step, all the variables that have restrictions in form of constraints have to be computed. This is done by evaluating the computation clauses designed parallel to the related constraints. The next step is the adaptation of template constraints to the actual situation on the schedule. We can imagine that a template compare value has a fuzzy distribution around zero, but the real compare value may be an aggregation of other processes variables and would have another value.

template constraint: fuzzy_var_foo1 0

specialized constraint: out_date - delivery_date 0

On the evaluation of each constraint, DynaFLIP++ invokes the evaluation mechanisms of FLIP++ (cf. Section 1, page 1). The evaluated constraints are put into the EVALTREE, where they can be aggregated as described in figure 4.1 on page 100. When all relevant constraints determined by the constraint script interpreter CSI(C&F) are evaluated, the overall evaluation score is returned to DomFLIP++ (cf. Section 3, page 93).

Further, we are interested in the biggest violations on the schedule, so we select bad evaluations at runtime and put them into an appropriate data structure, which is later used by DomFLIP++ to make changes on the schedule in order to avoid these violations. The repair steps depend on the optimization algorithm used by DomFLIP++. If a local change on the schedule has occurred, DynaFLIP++ has to check where the changes took place, in order not to build up a complete new evaluation tree, but only to recompute those parts of the schedule that have been changed. This reuse of already computed data structures will, similar to a caching mechanism, influence the runtime behavior of the evaluation process.

The most important object of DynaFLIP++ is the evaluation tree EVALTREE, which contains the specialized and evaluated constraints, and the aggregation of the latter. The variables with their actual values and the violations are stored in a separate structure.

4.3 Dynamic constraint satisfaction algorithms

To guide the search of the OptiFLIP++ (cf. Section 6, page 175) algorithms as discussed in , it is necessary to identify the constraint with the worst weighted evaluation, i.e., the severest conflict which can be attacked to minimize conflicts. This can be considered as a side product of evaluating the current instantiation. It corresponds to computing the evaluation using the minimum operator, and more importantly, to remember the constraint involved in the minimal weighted evaluation. This constraint represents the largest conflict for the current instantiation. Often the constraint corresponds to a general feature of the instantiation and cannot be attributed to a specific part of the instantiation. Depending on the repair operators available to the repair based constraint satisfaction algorithms, it can be helpful to find additionally the second largest and third largest conflict. Generally, the search should return the largest conflict being of a type that can be handled by an available repair operator. When DynaFLIP++ has to generate a new dynamic constraint representation for a given instantiation, it computes the individual 'leaf' constraints by calling FLIP++ (cf. Section 1, page 1) again and again with new variable instantiations on one of the stored reference constraints, and stores the results in an intermediate form that can be used by DynaFLIP++ for further aggregation. This ensures a relatively efficient processing of the constraints since the sometimes very large data structure of a static constraint can be reused for all dynamic constraints of its type. At the same time, DynaFLIP++ sorts all the computed intermediate evaluation scores, together with type information, for later selection of 'good' repair operators. All DynaFLIP++ functionality is implemented by CSI(C&F), an interpreter language for dynamic constraint adaptation and evaluation. In most cases, the values of variables are the result of mathematical or logical expressions, or aggregations of other variables. In this sense CSI(C&F) has to provide a bit more functionality than simple constraint stating and evaluating, thus CSI(C&F) comes quite close to traditional programming languages.

Chapter 5

Constraint Script Interpreter on Crisp & Fuzzy Sets

5.1 Introduction

CSI(C&F) is not a general-purpose programming language like C++ or JAVA. In a few words, the aim of CSI(C&F) is the evaluation of (fuzzy) constraints (cf. Section 5.5.5, page 143), and for this purpose it was developed at the Technical University of Vienna, Austria. Later we will see, what this constraints are and look like. But beside the wish to state constraints and to evaluate them with respect to the domain of their variables, we want to formulate when and how this constraints should be evaluated, or how single constraints and variables of constraints relate to other constraints and variables.

In most cases, the values of variables (cf. Section 5.3.1, page 125) are the result of mathematical or logical expressions, or aggregations of other variables. In this sense CSI(C&F) has to provide a bit more functionality than simple constraint stating and evaluating. In this sense, CSI(C&F) comes quite close to traditional programming languages.

CSI(C&F) is a procedural programming language that has some similarity to BASIC, but a reduced functionality of the latter and some restrictions on data types. On the other hand CSI(C&F) includes features like recursive (cf. Section 5.2.10, page 121) procedures as well as the ability to handle and operate on fuzzy data types. From this point of view CSI(C&F) can be seen as complete command-shell for the functionality of FLIP++.

In addition to this, CSI(C&F) provides an expression interpreter for mathematical and logical expressions, that is able to work on lists of arguments, even when the number of arguments is not known before runtime or may vary when used within a loop. Because of further functionality like the use of conditional operators (cf. Section 5.3.6.1, page 128) and other features, the expression interpreter can be seen as a stand-alone-module for simple ¨*functional*¨ programming (but not in the common sense as for example LISP). So CSI(C&F) is neither a subset of classical BASIC, nor is BASIC a subset of CSI(C&F).

Traditional programming languages like C++ or Pascal provide a variety of data types. In contrast, CSI(C&F) is a language based only on few data types that can be divided into two classes:

- *fuzzy data types* (cf. Section 5.5, page 135)
- *crisp data types* (cf. Section 5.3.2, page 125)

According to this distinction we have

- *operators/functions on fuzzy data types and*
- *operators/functions on crisp data types.*

The **crisp data type class** includes *floating point numbers* and *strings. Booleans* are floating point numbers where zero has the truth value *FALSE* and a value different from zero has the truth value *TRUE. Integers* are treated as a subset of floating point numbers. A detailed discussion on **fuzzy data types** will be given later.

One important fact is that CSI(C&F) knows only *global data*, that means that all variables are visible to the entire program. This has to be considered when procedures or function calls are invoked; CSI(C&F) has a global namespace and only one scope of domain. ¨Local¨ variables taking the same name as a global one may disturb the desired behavior of a program, even when this type of ¨error¨ is a semantically one and no error or warning message will be generated. The programmer has to take care not to change variables that are still used in an other part of the program. In this sense all variables are typically ¨static¨, as long as not removed explicitly with the DELETE statement (explicit *garbage collection*).

CSI(C&F) provides the fundamental control-flow constructions required for wellstructured programs: decision making (`IF - THEN - ELSE`), looping with the termination test at the top, the middle or the bottom (`FOR, LOOP`), early program exit (`EXIT`), procedure calls (`PROCEDURE - RETURN`), and labels.

As mentioned above, the main goal of CSI(C&F) consists in stating and **evaluating crisp** or fuzzy constraints. For this purpose CSI(C&F) provides *fuzzy variables* (`FUZZYVAR` (cf. Section 5.5.1, page 135)), objects for *parameter descriptions* (`PARAMETERSET` (cf. Section 5.5.2, page 137)), *operators* (`OPERATORSET` (cf. Section 5.5.3, page 139)), rules for the *fuzzy inference* (`RULESET` (cf. Section 5.5.4, page 140)), and a hierarchical *evaluation structure* (`EVALTREE` (cf. Section 5.5.8, page 147)) for the fuzzy conjunction or disjunction of evaluated constraints.

Finally, CSI(C&F) provides some basic input/output facilities (cf. Section 5.2.8, page 119) for the `IMPORT` and `EXPORT` of data. So far, a CSI(C&F) program may not exist in separate source files. Similarly, CSI(C&F) offers only straightforward, single-thread control flow.

CSI(C&F), like other languages, has its blemishes. Some parts of the syntax could be better, some extensions to the functionality of CSI(C&F) may be desirable, and some faults might not be eliminated while testing CSI(C&F). Future versions should eliminate most of the actual weak points.

The following chapters will be organized as follows. First of all a tutorial part with the purpose to get the reader started as quick as possible is given in section 5.2. Section 5.3 through 5.5 discuss various aspects of CSI(C&F) in more detail and rather more formally than does section 5.2. Section 5.3 deals with the basic data types, operators and expressions. Section 5.4 provides a formal language description by regular expressions. Section 5.5 covers fuzzy data types and operators as well as constraints and evaluation mechanisms. Section 5.6 gives a complete overview of the keywords (cf. Section 5.6, page 151) used in CSI(C&F).

5.2 A Tutorial Introduction

We will start with a quick introduction to CSI(C&F). Our aim is to show the essential elements of the language without getting lost in details, rules, and exceptions. The reader should get started as quickly as possible to reach the point where he can write useful programs with CSI(C&F).

5.2.1 A first example

The first program is the same for all languages: say hello to the world! In CSI(C&F) the program to do this is

```
PRINT "Hello, World!"
PRINT CR
```

where ¨Hello, World¨ is the string that has to be printed and CR is a constant (cf. Section 5.3.4, page 127) representing the carriage return escape sequence.

To run this program on your system you must create the program in a file that has not to have a special extension or format. The standard extension used by the developer is *.cst, but you can use your own filenames and extensions. If you run

```
csi hello_1.cst
```

it will print

```
Hello, World!
```

A CSI(C&F) program consists of *procedures* (cf. Section 5.3.1, page 125) and *variables* (cf. Section 5.3.1, page 125), the latter being crisp or fuzzy data types. A procedure or function contains *statements* that specify the computing operations to be done, and variables store the *values* used during the computation.

In CSI(C&F) you have no main function like C++ has to have one. This does not mean that a CSI(C&F) program has no structural restrictions at all. The first part on the top of a CSI(C&F) program is always the **main program**, where other procedures, if given, can be invoked by a CALL statement. The part below the main program can be seen as **procedure declaration space**.

If you declare procedures, you have to mark the end of the main program with the `EXIT` (cf. Section 5.2.9, page 121) statement. Beside marking the end of the main program the `EXIT` statement can be used to leave the program at any point, even in a procedure.

Now we will give a second version with the same behavior as the program above:

```
/* This is the main program */
CALL Hello
EXIT
/* This is the procedure declaration space */
PROCEDURE Hello
LET Say_Hello$ = ¨Hello, World!¨
PRINT Say_Hello$
PRINT CR
RETURN
```

The two lines

```
/* This is the main program */
/* This is the procedure declaration space */
```

are comments, which can be used to explain briefly what the program does. Of course the keywords (cf. Section 5.6, page 151) `CALL`, `EXIT`, etc. have not be stated in uppercase characters. `CaLL`, `eXiT`, etc. are also valid notations of this keywords. In this sense CSI(C&F) is not case sensitive. Not so on names of variables, procedures, etc. In CSI(C&F) the following procedure declarations and variable assignments are different:

PROCEDURE `Hello`

PROCEDURE `heLLo`

LET `Say_Hello$ = ¨Hello, World!¨`

LET `SAY_hello$ = ¨Hello,World!¨`

Procedures are a convenient way to encapsulate some computation, which can then be used without worrying about its implementation. With properly designed procedures, it is possible to ignore how a job is done; knowing what is done is sufficient.

When calling a procedure with `CALL`, the statement block within the invoked procedure is executed and after reaching the `RETURN` statement the execution continues with the next statement behind the `CALL` statement.

At this point we want to remark that procedures in the actual CSI(C&F) version can not return a value and consequently procedure calls can not be used in assignment statements. This may be changed in future versions of CSI(C&F).

5.2.2 Loops, Variables and Arithmetic Expressions

The following, a bit more complex program will introduce some new ideas like *assignments*, *loops*, the concept of *variables* and *arithmetic expressions*. By variables at this point we mean crisp data types, fuzzy data types will be treated in Chapter 5.5. In CSI(C&F), crisp data types like strings and floating point numbers must not be declared before they are used. The next program uses the formula °`C = (5/9)*(F-32)` to print the following table of Fahrenheit temperatures and their centigrade or Celsius equivalents:

```
Fahrenheit :  0
Celsius :  -17.77777778
Fahrenheit :  20
Celsius :  -6.666666667
...
Fahrenheit :  300
Celsius :  148.88888888
```

The program itself consists only of the main program and for this we have not to state an `EXIT` statement behind `ENDLOOP`:

```
/* Print Fahrenheit - Celsius table using LOOP */
LET Fahreinheit$ = ¨Fahrenheit :  ¨
LET Celsius$ = ¨Celsius :  ¨
LET lower = 0
LET upper = 300
LET step = 20
LET fahr = lower
LOOP
EXITLOOP (fahr>upper)
LET celsius = (5/9)*(fahr-32)
PRINT Fahrenheit$ PRINT fahr PRINT CR
PRINT Celsius$ PRINT celsius PRINT CR
LET fahr = fahr+step
ENDLOOP
```

The computation in the temperature conversion program begins with the assignment statements

```
LET lower = 0
LET upper = 300
LET step = 20
LET fahr = lower
```

which set the variables to their initial values. Individual statements have not to be terminated by any termination characters like semicolons etc. As each line in the temperature table is computed in the same way we use a loop that repeats once per output line.

```
LOOP
EXITLOOP (fahr>upper)
...
ENDLOOP
```

The loop operates as follows: the condition in parenthesis is tested. If it is true, we leave the loop statement, otherwise the body of the loop is executed. Then the condition is re-tested, and if false, the body is executed again. The body of a `LOOP` can be one or more statements, where the statement block within the body of the loop is terminated by the `ENDLOOP` statement.

The computation and assignment of the Celsius temperature gets done in the body of the loop by the statement

```
LET celsius = (5/9)*(fahr-32)
```

String assignments work in much the same way as numerical assignments do. Although variables holding string values have the same name space and structure as variables that hold numerical values, we suggest to mark strings with a '$' at the end of the name. So confusion about the ¨type¨ of variables can be avoided:

```
LET Celsius$ = ¨Celsius :  ¨
LET Celsius = (5/9)*(fahr-32)
```

This does not mean that Celsius$ and Celsius are incompatible types of variables. As mentioned in the introduction (cf. Section 5.1, page 105), CSI(C&F) does only differentiate between fuzzy data types and crisp data types. So you may assign values to string variables and vice versa:

```
LET Celsius$ = 3.141592654
LET Celsius = ¨Pi¨
```

Although CSI(C&F) does not care about how a program looks, well-named variables, proper indentation and spacing are critical in making programs easy for people to read.

5.2.3 The FOR - NEXT Statement

Another way to write loop statements is the use of the `FOR` statement. This is a specialization of `LOOP`. The `FOR` is usually appropriate for loops in which the initialization and increment are single statements and logically related, since it is more compact than `LOOP` and it keeps the loop control statements together in one place.

Different to the `LOOP` statement, the control variable and control test can only handle integer or floating point variables.

Now we do the same program as before using the FOR statement:

```
/* Print Fahrenheit - Celsius table using FOR */
LET Fahreinheit$ = ¨Fahrenheit :  ¨
LET Celsius$ = ¨Celsius :  ¨
LET lower = 0
LET upper = 300
LET step = 20
LET fahr = lower
FOR fahr = lower TO upper STEP step
LET celsius = (5/9)*(fahr-32)
PRINT Fahrenheit$ PRINT fahr PRINT CR
PRINT Celsius$ PRINT celsius PRINT CR
NEXT fahr
```

As one can see, the `FOR` statement is well suited for loops in which the number of computations of the loop is exactly known, and the termination test is more of a ¨numerical¨ than a ¨logical¨ nature. Of course not only incremental, but also decrement `FOR` loops with a not necessarily integer step-size are possible.

The following program

```
/* Print Fahrenheit - Celsius table using FOR */
LET Fahreinheit$ = ¨Fahrenheit :  ¨
LET Celsius$ = ¨Celsius :  ¨
LET lower = 0
LET upper = 300
```

```
LET step = -20 /* step has a negative value */
LET fahr = lower
FOR fahr = upper TO lower STEP step
LET celsius = (5/9)*(fahr-32)
PRINT Fahrenheit$ PRINT fahr PRINT CR
PRINT Celsius$ PRINT celsius PRINT CR
NEXT fahr
```

prints as a list of Celsius - Fahrenheit correspondings in a decremental order:

```
Fahrenheit :  300
Celsius :  148.88888888
...
Fahrenheit :  20
Celsius :  -6.666666667
Fahrenheit :  0
Celsius :  -17.77777778
```

The FOR-loop operates as follows: the condition, whether there are still computation steps left is tested. If it is false, we leave the FOR statement, otherwise the body of the loop is executed. Then the condition is re-tested, and if true, the body is executed again. The body of a FOR can be one or more statements, where the statement block within the body of the loop is terminated by the NEXT statement.

Of course both, LOOP (cf. Section 5.2.2, page 109) and FOR are statements on their own and can be used to build a statement block for the loop body of other LOOP and FOR statements:

```
LOOP
...
| LOOP
| ...
| EXITLOOP (...)
| ...
```

| **ENDLOOP**

. . .

EXITLOOP `(...)`

. . .

ENDLOOP

FOR `a = ...` **TO** `...` **STEP** `...`

. . .

| **FOR** `b = ...` **TO** `...` **STEP** `...`

| . . .

| **NEXT** `b`

. . .

NEXT `a`

All control-flow constructions can encapsulate other ones without restrictions. So any statement can be several statements enclosed by keywords (cf. Section 5.6, page 151) denoting begin and end of a statement block. The general form of the control-flow constructions we met so far is:

a) **IF** <condition>**THEN**

<statement block>

ELSE

<statement block>

ENDIF

b) **LOOP**

<statement block>

EXITLOOP <condition>

<statement block>

ENDLOOP

c) **FOR** <start assignment>**TO** <end assignment>**STEP** <step assignment>

<statement block>

NEXT

d) **PROCEDUE** procedurename
<statement block>
RETURN

5.2.4 Arrays

As many other programming languages, CSI(C&F) offers the possibility to use vectors or multidimensional arrays (cf. Section 5.2.5, page 117) for comfortable data organization. In this introductional part we will explain the use of arrays by modifying the Fahrenheit-Celsius program as follows:

```
/* Print Fahrenheit - Celsius table using arrays */
LET Fahreinheit$ = ¨Fahrenheit :  ¨
LET Celsius$ = ¨Celsius :  ¨
LET lower = 0
LET upper = 300
LET step = 20
LET fahr = lower
LET index = 0
LOOP
EXITLOOP (fahr>upper)
LET index = index+1
/* the results are stored in an array */
LET result[index].celsius = (5/9)*(fahr-32)
LET result[index].fahr = fahr
LET fahr = fahr+step
ENDLOOP
FOR i = 1 TO index STEP 1
PRINT Fahrenheit$ PRINT result[i].fahr PRINT CR
PRINT Celsius PRINT result[i].celsius PRINT CR
NEXT i
```

In contrast to other languages like C or C++, arrays in CSI(C&F) do not have to be declared before use; neither has the size of an array to be. Whenever an element of an array is used in an assignment statement like

```
LET result[index].celsius = (5/9)*(fahr-32)
```

a new element of this type is created in CSI(C&F)'s data space. When you try to use an arrays element that previously was not created by any assignment statement, CSI(C&F) will generate a warning or an error, depending on the context in which the ¨unknown¨ element was incorrectly used. **??**

By the use of '[' and ']' in names of variables and procedures (cf. Section 5.2.4, page 115) we can easily establish arrays without restrictions on their dimension. In 5.2.5 we will show the use of rectangular multidimensional arrays (cf. Section 5.2.5, page 117).

In CSI(C&F), indexing is not only a way of organizing data, but also a concept of names used in assignments, procedures, etc. This includes, that CSI(C&F) provides the feature of constructing *pseudo arrays of functions*, a concept well known from other programming languages. A brief example will illustrate this concept:

```
/* pseudo-arrays of procedures */
FOR i = 4 TO 8 STEP 1
LET output = (i>5)
CALL function[output]
NEXT i
EXIT
PROCEDURE function[0]
PRINT i PRINT ¨ is lower than or equal 5¨
PRINT CR
RETURN
PROCEDURE function[1]
PRINT i PRINT ¨ is greater than 5¨
PRINT CR
RETURN
```

Depending on the BOOLEAN result of the logical test condition `(i>5)`, the statement

```
CALL function[output]
```

alternates in calling `function[0]` and `function[1]`. The output of this program would be:

```
4 is lower than or equal 5
5 is lower than or equal 5
6 is greater than 5
7 is greater than 5
8 is greater than 5
```

5.2.5 Multi-dimensional Arrays

The use of array-indexing is not restricted to one dimension. CSI(C&F) provides rectangular multi-dimensional arrays. The following program stores the settings on a chess-board:

```
FOR row = 1 TO 8 STEP 1
FOR column = 1 TO 8 STEP 1
PRINT "Figure on position ["
PRINT row PRINT "," PRINT column
PRINT "] is :  "
INPUT Board[row][column]
PRINT "You entered "
PRINT Board[row][column] PRINT CR
NEXT column
NEXT row
```

Beside the use of the multi-dimensional array `Board[8][8]` we encountered the `INPUT` statement. This statement can be used to get variables values from the console. Numerical values are typed in their conventional form like

```
Figure on position [1,1] is :  10
You entered 10
```

The input of strings has to be enclosed by double-quotes:

```
Figure on position [1,2] is :  "King"
You entered King
```

5.2.6 Flow Control with IF - THEN - ELSE

In 5.2.4 we have seen the assignment statement

```
LET output = (i>5)
```

An assignment is an expression with a value and assignments associate from right to left. The condition test (i>5) returns the value 1 or 0, depending whether the test results true or false. But CSI(C&F) offers the possibility, to formulate more transparent and complex decision structures (cf. Section 5.3.3, page 127). Now we modify the referenced example to bring it to a more readable form:

```
/* the use of IF - THEN - ELSE */
FOR i = 1 TO 10 STEP 1
IF (i>5) THEN
PRINT i
PRINT ¨ is greater than 5¨
PRINT CR
ELSE
PRINT i
PRINT ¨ is lower than or equal 5¨
PRINT CR
ENDIF
NEXT i
```

This example shows a `THEN` and an `ELSE` statement block, which specifies the alternative actions if the condition part of an IF statement is true or false. The general form is

IF <condition>**THEN**

<statement block 1>

ELSE

<statement block 2>

ENDIF

One and only one of the two statement blocks associated with an `IF-THEN-ELSE` (cf. Section 5.2.6, page 118) statement is performed. If the condition is true, <statement block 1> is executed; if not, <statement block 2> is executed. Like anywhere, a statement block can be a single statement or several statements.

5.2.7 BGOTO and LABEL

CSI(C&F) provides the abusable `GOTO` statement, and labels to branch to. Formally, the `GOTO` is never necessary, and in practice it is almost always easy to write code without it. Nevertheless, there might be a few situations where `GOTO` may find place. As there exists no early loop exiting in CSI(C&F), as, e.g., ¨break¨ in C and C++, we can use `GOTO` to leave inner loops without exiting the whole program with `EXIT`.

```
/* Do some computation */
FOR ...
FOR ...
...
IF (exception) THEN /* early loop exiting */
GOTO Error
ELSE
...
ENDIF
NEXT ...
NEXT ...
EXIT
LABEL Error
/* Clean up and exit */
...
EXIT
```

The use of `GOTO` can always be avoided, though perhaps at the price of additional code.

5.2.8 I/O Facilities

Input and output of data are accessed through the use of `IMPORT` and `EXPORT`. As CSI(C&F) is not a general purpose programming language, it does not support any general I/O - functionality. At the moment we will only show the `IMPORT` and `EXPORT` of simple variables. Later we will treat the import and export of more complex objects like `RULESET`, `PARAMETERSET` (cf. Section 5.5.2, page 137), `PSLIST`, `OPERATORSET` (cf. Section 5.5.3, page 139), etc.

For this reason we have the `VARIABLES` qualifier in our `IMPORT` statement, as we are importing variables in our data space.

```
(1) IMPORT VARIABLES My_Vars.dat
LET number_of_factors = number_of_factors+1
LET factor[number_of_factors] = number_of_factors
LET product = 1
LET MyName$ = ¨Ulrich¨
FOR i = 1 TO number_of_factors STEP 1
LET product = product*factor[i]
NEXT i
DELETE factor[5]
(2) EXPORT VARIABLES My_Vars.dat
DELETE ALL
LET Message = ¨Only me¨
(3) EXPORT VARIABLES My_Vars.dat
EXIT
```

where My_Vars.dat is a file with the following contents at the moment `(1)`, `(2)`, and `(3)` in our program:

```
Situation (1): Situation (2): Situation (3):
/* #:  6 */ /* #:  8 */ /* #:  1 */
number_of_factors 5 5 number_of_factors 6 6 Message
¨Only me¨ 0
factor[1] 1 1 factor[1] 1 1
factor[2] 2 2 factor[2] 2 2
factor[3] 3 3 factor[3] 3 3
factor[4] 4 4 factor[4] 4 4
factor[5] 5 5 factor[6] 6 6
product product*factor[i] 720
MyName$ ¨Ulrich¨ 0
```

The structure of `My_Vars.dat` implies something about the structure of variables in CSI(C&F). Each variable has a *name*, a *formula* (!), and a *value*. This can be useful, when we want to import formulas those evaluation can only be done at runtime, after that other variables referenced in these formulas were computed. More detailed information about this is given later.

Now we will discuss the last example. Before (1) the data space is empty. After the `IMPORT` statement our data space holds 6 variables as in (1). Then `number_of_factors` is incremented by the program and 2 new variables `factor[6]` and `MyName$` are added to the data space. Finally the variable `product` is added with its formula. Before saving the data space in `My_Vars.dat`, `factor[5]` is deleted with the `DELETE` statement. Now our data space has the contents as in (2). `DELETE ALL` cleans up the whole data space. A new variable named `Message` is inserted and finally the data space is written to `My_Vars.dat` again (3).

5.2.9 The EXIT Statement

The `EXIT` statements can be used at any point in the code to leave the program. Before exiting all objects allocated are cleaned up.

5.2.10 Recursion

CSI(C&F) procedures may be used recursively; that is, a procedure may call itself either directly or indirectly. In contrast to recursion in most other modern programming languages, recursion in CSI(C&F) has to consider the fact, that all variables are global and no local data is created on a recursive call of a procedure. A brief example should illustrate this fact:

```
/* recursive procedure calls */
IMPORT VARIABLES My_Vars.dat
LET count = number_of_factors
LET product = 1
CALL prod
PRINT product
EXIT
/* prod is called again and again, */
/* as long as count is greater than 0 */
PROCEDURE prod
```

```
IF (count>0) THEN
LET product = product*factor[count]
LET count = count-1
CALL prod
ENDIF
RETURN
```

This program has the same functionality as the one shown in 5.2.8. The `FOR-NEXT` loop is replaced by the recursive call of the procedure prod. Recursive code is often more compact and easier to read than the non-recursive equivalent.

5.2.11 Memory Management and Debugging Facilities

On each assignment or implicit creation of variable as, e.g., in `FOR - NEXT` (cf. Section 5.2.3, page 112) loops, a new variable is added to the data space as we have seen in 5.2.8. The same happens when we create fuzzy variables and more complex objects like constraints. To remove variables, that are no longer of interest or that have only temporary character, we can use the `DELETE` statement to cancel this variable (or object). An example was already given in 5.2.8. To delete the whole data space we can use the qualifier `ALL`, in order to reduce the work that had to be done if we deleted each variable by name.

```
LET temp = ...
...
/* temp is no longer of interest */
DELETE temp
...
/* the variables in the memory */
/* are no more of interest */
DELETE ALL
```

Sometimes, especially while debugging and coding, it is useful to take a look at the data space and the variables values, their formula, computation flags etc. In much the same way as deleting objects, we can display variables on the console by the use of `MEMDUMP`. Again an additional qualifier can determine whether all objects or just a single one should be displayed:

MEMDUMP `varname`

MEMDUMP ALL

In the following sections we will treat CSI(C&F)'s language features for fuzzy computation (cf. Section 5.5, page 135). `MEMDUMP` and `DELETE` can be applied to most of those objects too, as, e.g.,

DELETE RULESET

DELETE EVALTREE

MEMDUMP RULESET

5.2.12 Constructions for Fuzzy Computation

A detailed description for fuzzy computation and fuzzy constraints is given in Chapter 5.5. In this introductional part we will just give a short example for fuzzy computation in order to get the reader familiar with fuzzy computation objects.

```
/* Import RuleSet with rules and ParameterSets */
IMPORT RULESET MyRules.RuleSet
/* table of correspondings for rule generation */
TABLE Fuzzy.Table
/* declare fuzzy variables with the associated */
/* ParameterSet, its lower and upper support */
/* end, median, distribution, dilatation, etc.  */
FUZZYVAR f_in1 fuzzy_in
FUZZYVAR f_in2 fuzzy_in
FUZZYVAR f_out fuzzy_out
FUZZYVAR f_comp fuzzy_out
/* set f_in1 to crisp value 10 */
SET f_in1 = 10
/* set f_in2 to crisp value 80 */
SET f_in2 = 80
/* f_comp is a fuzzy comparison of f_in1 < 20 */
FUZZYCOMPARE f_comp IS f_in1 < 20
```

```
/* f_out is fuzzy result of f_comp AND f_in2, */
/* where AND is a fuzzy conjunctor */
FUZZYCONCAT f_out IS f_comp AND f_in2
/* evaluate f_out according to the RuleSet */
EVAL f_out
/* display defuzzified value for f_out */
PRINT f_out
/* export modified RuleSet */
EXPORT RULESET MyRules.RuleSet
```

In contrast to crisp variables, all fuzzy variables have to be declared before use by the assignment of a *ParameterSet* (cf. Section 5.5.2, page 137) that describes the *linguistic variables*, *distribution*, *dilatation*, *lower* and *upper support end* of the variable, *median* and *importance*.

The evaluation `EVAL f_out` happens according to the rules in the RuleSet. In our example such rules could be of the form:

```
IF f_comp IS negative AND f_in2 IS negative THEN f_out
IS bad WITH CERTAINTY 1.
...
IF f_comp IS zero AND f_in2 IS zero THEN f_out IS
neutral WITH CERTAINTY 1.
...
IF f_comp IS positive AND f_in2 IS positive THEN f_out
IS good WITH CERTAINTY 1.
```

The single statements and keywords (cf. Section 5.6, page 151) are treated in detail in section 5.5, but anyone familiar with fuzzy set theory should get an idea of what is going on in this brief example.

The more general notation of these rules is of the format:

IF <termlist>THEN <term>WITH CERTAINTY c.

where

termlist = term | termlist ("AND" | "OR") term

term = Ident

Ident = (letter | digit) (letter | digit)*

Number = digit+ | digit+ ¨.¨ digit+

digit = 0..9

letter = printable ascii characters

c = Number

5.3 Types, Operators, and Expressions

¡!latex ¿

5.3.1 Variable and Procedure Names

Variable names and procedure names are not allowed to include spaces. Instead of a blank character you should use the underscore ¨_¨ for improving the readability of long variable names. Upper case and lower case letters are distinct, so `x` and `X` are two different names. Although keywords like `IF`, `THEN`, `LOOP`, etc., are not reserved words and would be correct names for variables and procedures, it is not very useful to use such keywords (cf. Section 5.6, page 151) as names. It would be convenient to choose names that are related to the purpose of the variable or procedure. The actual implementation allows names to have a maximum length of 80 characters.

5.3.2 Data Types and Sizes

There exist only two basic data types in CSI(C&F):

- *numerical data type (floating point numbers, integers, booleans)* and
- *alphanumerical data type (strings)*

There is no special *integer* type in CSI(C&F), they are treated as a subset of floating point numbers. The range of floating point numbers depends on the system used. Strings are restricted not to exceed a maximum length of 128 characters. This seems to be a silly restriction, but as mentioned in the introduction (cf. Section 5.1, page 105), CSI(C&F) was not designed to be a complete, general-purpose programming language. So this restriction should be irrelevant for most tasks of constraint satisfaction problems.

The two basic data types own different semantics, but are implemented by one and the same concept. Both have the following internal representation:

```
name    name of Variable

formula stores a formula for anumerical variable
or a string for an alphanumerical variable

value   holds the value of a numerical variable or the numerical
        equivalent of an alphanumerical variable
```

CSI(C&F)'s data space consists of a linear sequence of variables with the format we have just seen. A brief example at the end of this section will figure out this concept.

Whenever a fuzzy variable is declared or evaluated, a numerical variable is created which holds the name of the fuzzy variable, the name of the associated ParameterSet (cf. Section 5.5.2, page 137), and the defuzzified value of the fuzzy variable.

```
name    name of FUYYZVAR

formula name of PARAMETERSET object

value   defuzzified value of FUYYZVAR
```

This guarantees, that fuzzy variables, once defuzzified, can be used in other regular expressions and computations. At this time we should remember, that fuzzy variables have two representations in CSI(C&F): on the one hand they are stored in the ¨normal¨ data space so that they can be accessed like any other variable, on the other hand it's obvious, that a fuzzy variable has a second, much more complex structure for fuzzy computation (cf. Section 5.5, page 135) too. This structure will be discussed in detail in section 5.5.

To illustrate the memory management (cf. Section 5.2.11, page 122) for variables we give a short example. The following assignments:

LET `Name$ = ¨A¨`

LET `fahr = 0`

LET `celsius = (5/9)*(fahr-32)`

FUZZYVAR `fuzzyvar default_parameterset`

SET `fuzzyvar = 3.4`

EVAL `fuzzyvar`

added to an empty data space would result in the following memory structure

```
name      Name$
formula   "A"
value     64
name      fahr
formula   "0"
value     0
name      celsius
formula   "(5/9)*(fahr-32)"
value     -17.77777778
name      fuzzyvar
formula   default_parameterset
value     48.33333333
```

5.3.3 Structures

Although CSI(C&F) does not provide any special constructions for structures or records, they can easily be realized by smart naming of the variables:

LET `Person[i].name[1]$ = ¨Donald¨`

LET `Person[i].name[2]$ = ¨Duck¨`

LET `Person[i].age = 26`

LET `Person[i].score = test[i].result`

For this reason, ¨fields¨ or elements of structures can only be accessed via their full name. The data structure `Person[]` in the last example can be seen as an array of structures (or records) with the attributes `name[1]$`, `name[2]$`, `age` and `score`.

5.3.4 Constants

In CSI(C&F) we can not find a special concept for constants. It remains in the responsibility of the programmer not to change variables that should be constant.

LET `PI = 3.141592654`

To support mnemonically the right use of constants and variables, we suggest to use lower case for variables, and all upper case for constants.

5.3.5 Declarations

In CSI(C&F) only fuzzy variables must be declared before use. This seems to be convenient, as it combines the declaration of variable with the binding of a fuzzy variable to a certain ParameterSet-object. All other declarations are made implicitly by context. One exception is the `EVALTREE` (cf. Section 5.5.8, page 147) object; in a first step you can create such an object explicitly by the contemporaneous definition of its structure. In a second step various evaluated `FUYYZVAR` objects can be inserted in this tree, but this is not the only way to create an `EVALTREE`. If you try to insert a `FUZZYVAR` (cf. Section 5.5.1, page 135) object in an empty tree or in a not existent branch, a new tree or branch will be generated automatically. Nevertheless, CST(C&F) gives you a warning message in such a case.

5.3.6 Arithmetic, Relational, and Logical Operators

To guarantee the correct precedence of the operators, any triple

(<expression> `operator` <expression>)

should be put into brackets, independent whether the is an arithmetical, relational, or logical one.

5.3.6.1 Binary Arithmetic Operators

$+$	addition
$-$	subtraction
$*$	multiplication
$/$	division

Table 5.1: Binary Arithmetic Operators

5.3.6.2 Relational Operators

>	greater than
<	smaller than
=	equal
!	not equal

Table 5.2: Relational Operators

The relational operators are the only one that work on both, numerical and alphanumerically data (cf. Section 5.3.2, page 125) types, as illustrated by the following example:

```
LET value = 5
LET my_name = ¨Author¨
LET numcmp = (3.14<Value)
LET strcmp = (¨Author¨=my_name)
```

5.3.6.3 Logical Operators

&	logical AND
|	logical OR

Table 5.3: Logical Operators

with the following semantics:

& (AND)	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

| (OR)	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

Table 5.4: Semantics of Logical Operators

5.3.7 The built-in-functions of CSI(C&F)

5.3.7.1 unary Functions

SIN'('<expression>')'

COS'('<expression>')'

TAN'('<expression>')'

ABS'('<expression>')'

LOG'('<expression>')'

SQR'('<expression>')'

SRT'('<expression>')'

5.3.7.2 n-ary Functions

SUM['(' index '=' start [':' end] ')'] '{' <expression> [',' <expression>]'}'

MUL['(' index '=' start [':' end] ')'] '{' <expression> [',' <expression>]'}'

MIN['(' index '=' start [':' end] ')'] '{' <expression> [',' <expression>]'}'

MAX['(' index '=' start [':' end] ')'] '{' <expression> [',' <expression>]'}'

Note: when you use an index variable in n-ary functions, you can use the count-operator # to count the appearances of substituted indices. The count-operator can only be used within one expression, that means that #i has no value when used in a following statement.

LET sum : #i***SUM**(i=1:5)**{**Vector[i]**}**

would be equivalent to

LET sum = ***SUM{**Vector[1],Vector[2],Vector[3], Vector[4],Vector[5]**}**

would be equivalent to

To make such a substitution explicitly, we use the modified assignment operator ':' instead of the '=' assignment operator in our LET or SET statements. If such a statement is encountered, CSI(C&F) scans the data space for all instanciated Vector[i] objects, and inserts them in the expression list of this n-ary function. In the same step #i is incremented. The start and end condition (i=1:5) are optional and can be used in much the same way as in a FOR - NEXT loop.

Note that you can state a search condition without upper bound, as e.g.,

```
LET m :  MIN(i=4){Vector[i]}.
```

This would be equivalent to the statement

```
LET m = MIN{Vector[4],Vector[5]}
```

if we assume that no variable `Vector[i]` exists with i>5.

5.3.8 Conditional Expressions

The statements

```
IF (a>b) THEN
LET m = a
ELSE
LET m = b
ENDIF
```

compute in m the maximum of a and b. The conditional expression, written with the ternary operator '?', provides an alternative way to write this and similar constructions. In the expression

?{<expression_if>**,**<expression_then>**,**<expression_else>**}**

the expression <expression_if> is evaluated first. If it is true, then <expression_then> is evaluated, and that is the value of the conditional expression. Otherwise <expression_else> is evaluated, and that is the value. Only one of <expression_then> and <expression_else> is evaluated. According to the IF-THEN-ELSE statement above we can state the equivalent conditional statement

```
LET m = ?{(a>b),a,b}.
```

The conditional expression can be used to reduce code and to make programs more compact. A brief example should illustrate the use of CST(C&F)'s expression interpreters functionality.

```
LET c1 = MIN(i=1:10){Vector[i]}.
LET c2 = MAX{Array[j]}*#j.
LET result = ?{((c1>c2)|(c2=0)),(c1-c2),SQR(c1/c2)}
PRINT result
```

As we can see, this program works without any use of `LOOP` (cf. Section 5.2.2, page 109), `FOR-NEXT` (cf. Section 5.2.3, page 112) or `IF-THEN-ELSE`-statements.

5.4 Formal Description of CSI(C&F)

The syntax of CSI(C&F) programs is given by the following formal description using regular expressions (RE):

Program = StatSeq?

StatSeq = Stat | StatSeq Stat

Stat = e | InputStat | PrintStat | ImportStat | ExportStat | DelStat | MemDumpStat | ExitStat |

GotoStat | LabelStat | CallStat | ProcStat | AssStat | IfStat1 | IfStat2 | LoopStat |

ForStat | ComStat | OperStat | ParaStat | TableStat | FuzzyVarStat | EvalTreeStat |

FuzzyCompStat | FuzzyConcStat | AggregateStat | EvalStat

ComStat = ¨/*¨ StatSeq? ¨*/¨

OperStat = ¨OPERATORSET¨

¨IMPORTANCE¨ FuzzyOp

¨AND¨ FuzzyOp

¨OR¨ FuzzyOp

¨CERTAINTY¨ FuzzyOp

¨INFERENCE¨ FuzzyOp

¨ACCUMULATE¨ FuzzyOp

¨DEFUZZIFICATION¨ DefuzzyOp

¨GAMMA¨ Number

¨ACCURACY¨ Number

¨END¨

ParaStat = ¨PARAMETERSET¨

¨NAME¨ Ident

¨LOWER¨ Number

¨UPPER¨ Number

¨MEDIAN¨ Number

¨DISTIRBUTION¨ Number

¨DILATATION¨ Number

¨IMPORTANCE¨ Number

¨TERMS¨ Ident+

¨END¨

TableStat = ¨TABLE¨ Ident

FuzzyVarStat = ¨FUZZYVAR¨ Ident Ident

EvalTreeStat = ¨EVALTREE¨ Ident Ident (Ident (¨OR¨ | ¨AND¨))* ¨END¨

FuzzyCompStat = ¨FUZZYCOMPARE¨ Ident ¨IS¨ Ident (¨<¨ | ¨>¨ | ¨=¨ | ¨#¨) Expr

FuzzyConcStat = ¨FUZZYCONCAT¨ Ident IS Ident (¨AND¨ | ¨OR¨) Ident

AggregateStat = ¨AGGREGATE¨ Ident (¨AND¨ | ¨OR¨) Ident Ident

EvalStat = ¨EVAL¨ Ident

DefuzzyOp = ¨COG¨ | ¨NO_DFOP¨

FuzzyOp = (¨MAXIUMUM¨ | ¨MINIMUM¨ | ¨FUZZY_AND¨ | ¨FUZZY_OR¨ |

¨ASS_COMP_AND¨ | ¨ASS_COMP_OR¨ | ¨NO_FOP¨)

InputStat = ¨INPUT¨ Ident

PrintStat = ¨PRINT¨ (Ident | ¨CR¨)

ImportStat = ¨IMPORT¨ (¨RULESET¨ | ¨VARIABLES¨ | ¨PARAMETERSET¨ Ident | ¨PSLIST¨) Ident

ExportStat = ¨EXPORT¨ (¨RULESET¨ | ¨VARIABLES¨ | ¨PARAMETERSET¨ Ident | ¨PSLIST¨) Ident

DelStat = ¨DELETE¨ (¨ALL¨ | (¨EVALTREE¨ | ¨RULESET¨ | e) Ident)

MemDumpStat = ¨MEMDUMP¨ (¨ALL¨ | Ident)

ExitStat = ¨EXIT¨

GotoStat = ¨GOTO¨ Ident

LabelStat = ¨LABEL¨ Ident

CallStat = ¨CALL¨ Ident

ProcStat = ¨PROCEDURE¨ Ident StatSeq? ¨RETURN¨

AssStat = (¨LET¨ | ¨SET¨) Ident (¨=¨ | ¨:¨) Expr

IfStat1 = ¨IF¨ Cond ¨THEN¨ StatSeq? ¨ENDIF¨

IfStat2 = ¨IF¨ Cond ¨THEN¨ StatSeq? ¨ELSE¨ StatSeq? ¨ENDIF¨

LoopStat = ¨LOOP¨ StatSeq? ¨EXITLOOP¨ Cond StatSeq? ¨ENDLOOP¨

ForStat = ¨FOR¨ Ident ¨=¨ Expr ¨TO¨ Expr ¨STEP¨ Expr StatSeq ¨NEXT¨ Ident

Cond = ¨(¨ Expr (¨=¨ | ¨!¨ | ¨<¨ | ¨>¨ | ¨&¨ | ¨|¨) Expr¨)¨

Expr = signedExpr | unsignedExpr

unsignedExpr = Expr (¨+¨ | ¨-¨ | ¨*¨ | ¨/¨) Expr | Number | Ident | ¨(¨ Expr ¨)¨ | Func | CondOp

signedExpr = (¨+¨ | ¨-¨) ¨(¨ unsignedExpr ¨)¨

Func = unary_Func | (CountOp n_ary_Func CountOp)

unary_Func = (¨SIN¨ | ¨COS¨ | ¨TAN¨ | ¨LOG¨ | ¨ABS¨ | ¨SQR¨ | ¨SRT¨) Arglist

n_ary_Func = (¨SUM¨ | ¨MUL¨ | ¨MIN¨ | ¨MAX¨) (IndexStat | e) Arglist

CondOp = ¨?¨ ¨{¨ Expr, Expr, Expr¨}¨

Arglist = ¨{¨ Expr (¨,¨ Expr | e)* ¨}¨

IndexStat = ¨(¨ letter ¨=¨ Number (¨:¨ Number | e) ¨)¨

Count = ¨#¨ letter

CountOp = (Count (¨+¨ | ¨-¨ | ¨*¨ | ¨/¨) | e) | ((¨+¨ | ¨-¨ | ¨*¨ | ¨/¨) Count | e)

Number = digit+ | digit+ ¨.¨ digit+

Ident = (letter | digit) (letter | digit)*

digit = 0..9

letter = printable ascii characters

e = empty

This formal description should give the reader a quite complete structure analysis of a CSI(C&F) program.

5.5 Fuzzy Computation

In this section we will discuss the fuzzy computation constructions of CSI(C&F). All fuzzy functionality of CSI(C&F) is achieved by the mapping of most CSI(C&F) fuzzy constructions to FLIP++ or DynaFLIP++ objects and methods. We will briefly line out the relations of these two worlds by giving a list of corresponding objects of CSI(C&F) and FLIP++:
CSI(C&F) FLIP++ FILES

`FUZZYVAR class CombinedVariable (flipcvar.h and flipcvar.cpp) (cf. Section 1.2.3)`

`PARAMETERSET class ParameterSet (flippara.h and flippara.cpp (cf. Section 1.2.2))`

`OPERATORSET class OperatorSet (flipoper.h and flipoper.cpp (cf. Section 1.2.4))`

`RULESET class RuleSet (fliprule.h and fliprule.cpp (cf. Section 1.2.5))`

`FUZZYCOMPARE - (dynaflip.h and dynaflip.cc)`

`FUZZYCONCAT - (dynaflip.h and dynaflip.cc)`

`TABLE - (dynaflip.h and dynaflip.cc)`

`EVALTREE class TEvalTree (evaltree.h and evaltree.cpp)`

`AGGREGATE - (dynaflip.h and dynaflip.cc)`

In the following sections all these objects will be treated in more detail.

5.5.1 FUZZYVAR

A `FUZZYVAR` object is a representation of a fuzzy variable consisting of various linguistic terms with their own parameters, membership functions and instantiated membership functions.

The *parameters* for the fuzzy variable and its linguistic terms are taken from the `PARAMETERSET` (cf. Section 5.5.2, page 137) object that was associated to a `FUZZYVAR` on declaration (cf. Section 5.1, page 105) of the latter by a statement with the following syntax:

FUZZYVAR `fuzzyvar_name parameterset_name`

In contrast to crisp variables, fuzzy variables have to be declared before use. This makes sense, because we are not able to work on a `FUZZYVAR` object that is not described by a `PARAMETERSET`.

A `FUZZYVAR` is implemented by the use of the FLIP++ class Combined-Variable (flipcvar.h). To illustrate the concept of a `FUZZYVAR` object we give a complete internal representation of `My_fuzzyvar` from the example in 5.5.2:

`cv_name:` **My_fuzzyvar**

`crisp_value:` **20**

`range_lower_end: 0`

`range_upper_end: 50`

`median: 20`

`distribution: 0.9`

`dilatation: 0.9`

`importance: 1.0`

`linguistic_term:` **low**

`lower_support_end: 0`

`upper_support_end: 19.1`

`fs_color: blue`

`function: point_list`

`match_value:` **0**

`membership_function: 0/0 0/1 11/1 19.1/0`

`instanciated_membership_function: 0/0 19.1/0`

`linguistic_term:` **medium**

`lower_support_end: 2.9`

`upper_support_end: 37.1`

`fs_color: green`

`function: point_list`

`match_value:` **0.167**

`membership_function: 2.9/0 11/1 29/1 37.1/0`

`instanciated_membership_function: 2.9/0 4.25/0.167 35.8/0.167 37.1/0`

`linguistic_term:` **high**

```
lower_support_end: 20.9
upper_support_end: 50
fs_color: red
function: point_list
match_value: 0
membership_function: 20.9/0 29/1 50/1 50/0
instanciated_membership_function: 20.9/0 50/0
```

The membership and instanciated membership functions are represented by a linear list of point (x and y-coordinate) objects. This is indicated by the type of function that is used: `point_list`. The match_value represents the value of intersection between the crisp_value of a `FUZZYVAR` (CombinedVariable) and the membership function of a specific linguistic term, or, in other words, match_value indicates the satisfaction of a linguistic term achieved by a `crisp_value`.

A FUYYZVAR is assigned a value by the SET statement:

SET `My_fuzzyvar ( = | : ) <expression>`

The evaluation of a `FUZZYVAR` objects can be forced by the following statement:

EVAL `My_fuzzyvar`

This statement starts the process of converting a fuzzy value to a crisp value. The most common method of defuzzification is the centroid, or center of gravity, defuzzification method. This evaluation happens according to the rules stated in the `RULESET` for `My_fuzzyvar` and the selected operators used in the `OPERATORSET` (cf. Section 5.5.3, page 139). Once evaluated, a `FUZZYVAR` object can be accessed like any other (crisp) variable in CSI(C&F).

5.5.2 PARAMETERSET

The `PARAMETERSET` object (FLIP++ class ParameterSet in flippara.h) is a composition of all parameters that are needed to describe a `FUZZYVAR` (CombinedVariable). Among these is the range with its *lower* and *upper end*, the *median*, the *dilatation*, the *distribution*, and the *importance*. Additionally, it describes the *linguistic terms*, e.g. names and colors (for graphical representation), lower and upper support end as well as the shapes of the (instanciated) membership function of each linguistic term.

The declaration and instantiation of a `PARAMETERSET` object occur at the same time by the definition in the following way:

PARAMETERSET
NAME `Ident`
LOWER `Number`
UPPER `Number`
MEDIAN `Number`
DISTRIBUTION `Number`
DILATATION `Number`
IMPORTANCE `Number`
TERMS `Ident+`
END

where

Number = (¨+¨ | ¨-¨ | e) digit+ | digit+ ¨.¨ digit+
Ident = (letter | digit) (letter | digit)*
digit = 0..9
letter = printable ascii characters
e = empty

To concretize the ideas of 5.5.1 and 5.5.2 we give a brief example of how `FUZZYVAR` (cf. Section 5.5.1, page 135) and `PARAMETERSET` cooperate:

`/* Declare ParameterSet */`
PARAMETERSET
NAME `My_parameterset`
LOWER `0`
UPPER `50`
MEDIAN `20`
DISTRIBUTION `0.9`
DILATATION `0.9`
IMPORTANCE `1.0`
TERMS `low medium high`
END

```
/* Create FUZZYVAR with associated ParameterSet */
```

FUZZYVAR `My_fuzzyvar My_parameterset`

`(1)` **SET** `My_fuzzyvar = 20`

After (1) we can observe My_fuzzyvar to have exactly the internal state as shown in the example in 5.5.1. The I/O-facilities (cf. Section 5.2.8, page 119) for `PARAMETERSET` are the same as for most of the other CSI(C&F) objects:

IMPORT PARAMETERSET `File_Name`

EXPORT PARAMETERSET `ParameterSet_Name File_Name`

IMPORT PSLIST `File_Name`

EXPORT PSLIST `File_Name`

The `PSLIST` qualifier is responsible for the import and export of lists of `PARAMETERSET` objects. This PSLIST is the list of `PARAMETERSET` objects held by the `RULESET` (cf. Section 5.5.4, page 140) object.

5.5.3 OPERATORSET

The `OPERATORSET` class aggregates the operators and methods necessary to do calculations with a `FUZZYVAR` (cf. Section 5.5.1, page 135). There are four important steps to achieve results from a rulebase: these are *aggregation*, *certification*, *inference* and *accumulation*. In the aggregation step the match values of all instantiated membership functions are aggregated to an overall match value by applying the selected `AND/OR` operators. In the next step the overall certified match value is calculated by applying the selected `CERTAINTY` operator to the overall match value and the certainty value. The inference step calculates a new match value by applying the `INFERENCE` operator to the overall certified match value and the old match value. Finally the crisp value of a `FUZZYVAR` is calculated through the defuzzification process (center of gravity: `COG`).

OPERATRORSET

IMPORTANCE `FuzzyOp`

AND `FuzzyOp`

OR `FuzzyOp`

CERTAINTY `FuzzyOp`

INFERENCE `FuzzyOp`

ACCUMULATE `FuzzyOp`

DEFUZZIFICATION `DefuzzyOp`

GAMMA [`0..1`]

ACCURACY [`0..10`]

END

At the moment FLIP++ and CSI(C&F) support the following fuzzyfication and defuzzification operators:

DefuzzyOp = `COG`

`NO_DFOP`

FuzzyOp = `MAXIUMUM`

`MINIMUM`

`FUZZY_AND`

`FUZZY_OR`

`ASS_COMP_AND`

`ASS_COMP_OR`

`NO_FOP`

where COG stands for ¨center of gravity¨-defuzzification method, NO_DFOP stand for ¨no defuzzification operator¨, MAXIMUM and MINIMUM stand for that what we suggest them to do, FUZZY_AND is realized with a T-Norm calculation, FUZZY_OR is realized with a T-CoNorm calculation, ASS_COMP_AND stands for ¨associative compensatory and¨, ASS_COMP_OR stands for ¨associative compensatory or¨, and finally NO_FOP stands for ¨no fuzzy operator¨. Further mathematical and set theoretical discussion on these operators can be found in Chapter 4 or in the bibliography indicated there.

5.5.4 RULESET

Each CSI(C&F) program has a `RULESET` object, that has not to be declared explicitly. It even can't be declared by the use of any `RULESET` statement. Whenever a `FUZZYVAR`, `OPERATORSET`, or `PARAMETERSET` object is declared in a CSI(C&F) program, it is automatically inserted in the `RULESET` object of CSI(C&F). Now we will discuss the functionality of `RULESET` in detail.

The CSI(C&F) object `RULESET` (FLIP++ class RuleSet in fliprule.h) aggregates all the different rules for a list of `FUZZYVAR` objects (FLIP++ class CombinedVariable in flipcvar.h) to one linear list, with additional parameters. Among them is a list of `FUZZYVAR` (cf. Section 5.5.1, page 135) objects, a list of PARAMETERSET objects and one OPERATORSET object, so this class combines all of the functionality contained in FLIP++. All rules concerning one specific `FUZZYVAR` object (one CombinedVariable) are aggregated to one RuleBlock. Such rules consist of terms on the left side and a term on the right side. The left side terms are aggregated by a concatenation operator out of {AND, OR}, those functionality is specified in`OPERATORSET`. Additionally, every rule has a certainty attached to it. The format of a rule is

`IF` <termlist>`THEN` <term>`WITH CERTAINTY c.`

where

termlist = term | termlist (¨AND¨ | ¨OR¨) term

term = Ident

Ident = (letter | digit) (letter | digit)*

digit = 0..9

letter = printable ascii characters

c [0,1]

The internal structure of a `RULESET` can be written to a file by the following statement:

EXPORT RULESET `File_Name`

Analogous to this we can read a whole `RULESET` from a file by the use of

IMPORT RULESET `File_Name`

To delete a whole `RULESET` we have to write

DELETE RULESET

To display the actual contents or a `RULESET` we can use the debugging (cf. Section 5.2.11, page 122) statement

MEMDUMP RULESET

Now we complete the example started in 5.5.1 by giving the internal representation of an instantiation of a `RULESET` object:

`/* #: 1 */`

My_parameterset (1)

`0 50 20 0.9 0.9 1.0`

`/* #: 3 */ blue green red`

`/* #: 3 */ low medium high`

`/* #: 3 */` **(2)**

My_fuzzyvar `0 50 20 0.9 0.9 1.0`

`...linguistic terms, membership functions, etc., as in 5.5.1`

in1 `0 50 20 0.9 0.9 1.0`

`...linguistic terms, membership functions, etc., as in 5.5.1`

in2 `0 50 20 0.9 0.9 1.0`

`...linguistic terms, membership functions, etc., as in 5.1`

`minimum minimum maximum minimum maximum minimum cog 1 10` **(3)**

RuleSet (4)

`/* #: 1 */`

`/* #: 9 */`

`IF in1 IS low AND in2 is low THEN My_fuzzyvar IS low WITH CERTAINTY 1.`

`IF in1 IS low AND in2 is medium THEN My_fuzzyvar IS low WITH CERTAINTY 1.`

`IF in1 IS low AND in2 is high THEN My_fuzzyvar IS medium WITH CERTAINTY 1.`

`IF in1 IS medium AND in2 is low THEN My_fuzzyvar IS low WITH CERTAINTY 1.`

`IF in1 IS medium AND in2 is medium THEN My_fuzzyvar IS medium WITH CERTAINTY 1.`

```
IF in1 IS medium AND in2 is high THEN My_fuzzyvar IS
medium WITH CERTAINTY 1.

IF in1 IS high AND in2 is low THEN My_fuzzyvar IS
medium WITH CERTAINTY 1.

IF in1 IS high AND in2 is medium THEN My_fuzzyvar IS
medium WITH CERTAINTY 1.

IF in1 IS high AND in2 is high THEN My_fuzzyvar IS
high WITH CERTAINTY 1.
```

In this example only one `PARAMETERSET` object My_parameterset (1) was inserted after declaration in RULSET. Then we can find 3 `FUZZYVAR` objects, namely `in1`, `in2`, and My_fuzzyvar (2). They were all created with the same `object My_parameterset. At (3) we can see the operators selected for the` OPERATORSET object. Finally we have one ruleblock for `FUZZYVAR` (CombinedVariable) `My_fuzzyvar` holding 9 rules. When evaluating the right side `FUZZYVAR My_fuzzyvar`, all `FUZZYVAR` objects on the left side term-list (`in1` and `in2`) are evaluated recursively (cf. Section 5.2.10, page 121). This provides the possibility to build up hierarchical rule-structures for more complex evaluating.

5.5.5 FUZZYCOMPARE

The `FUZZYCOMPARE` object can be used to state fuzzy constraints of the following format:

> **FUZZYCOMPARE** FuzzyVar1 `IS` FuzzyVar2 CompareOp Expr

where FuzzyVar1 and FuzzyVar2 are `FUZZYVAR` (cf. Section 5.5.1, page 135) objects. CompareOp is a compare operator out of a user-definable set of operators. In most cases these would be {=, #, <, >}. Expris a regular expression as defined in section 5.4. A brief example shows the use of `FUZZYCOMPARE`:

```
/* declaration of FUZZYVAR objects and their
PARAMETERSET objects */

FUZZYVAR fuzzy_variable1 parameterset1

FUZZYVAR fuzzy_variable2 parameterset1

FUZZYVAR fuzzy_constraint parameterset2
```

```
/* give the variables values */
LET crisp_variable = 5
SET fuzzy_variable1 = 20
SET fuzzy_variable2 = 0
/* define and evaluate some constraints */
FUZZYCOMPARE fuzzy_constraint IS
fuzzy_variable1 = 10
EVAL fuzzy_constraint
FUZZYCOMPARE fuzzy_constraint IS
fuzzy_variable1 # fuzzy_variable2
EVAL fuzzy_constraint
FUZZYCOMPARE fuzzy_constraint IS
fuzzy_variable1 < SQR(crisp_variable)
EVAL fuzzy_constraint
FUZZYCOMPARE fuzzy_constraint IS
fuzzy_variable1 > crisp_variable
EVAL fuzzy_constraint
```

A `FUZZYCOMPARE` constraint compares a fuzzy variable `FUZZYVAR fuzzy_variable` to a crisp value. What the compare operators are and how single linguistic terms correspond to other when such comparison is done, has to be described in a `TABLE` (cf. Section 5.5.7, page 145) file explicitly. This makes sure that the system is open to extensions by the user. The format of such `TABLE` files can be found in section 5.5.7. Whenever such a constraint is encountered, CSI(C&F) generates automatically the appropriate rules using the table of correspondings as defined in `TABLE`. So the programmer has to take care about defining the right `TABLE` before the use of `FUZZYCOMPARE`.

5.5.6 FUZZYCONCAT

A `FUZZYCONCAT` constraint concatenates two `FUZZYVAR` objects with a concatenation operator out of {AND; OR}. In much the same way as mentioned in 5.5.5, the correspondings of linguistic terms have to be defined in a `TABLE` (cf. Section 5.5.7, page 145) file. So for all concatenation operators and all combinations of linguistic terms there must be a resulting term in the table of correspondings. For detailed information see section 5.5.7. Whenever a

FUZZYCONCAT is encountered, CSI(C&F) generates automatically the appropriate rules using this table, as long as no such rules were already generated previously.

The format of a FUZZYCONCAT statement has the format

> **FUZZYCONCAT** FuzzyVar1 IS FuzzyVar2 ConcatOp FuzzyVar3

where FuzzyVar1, FuzzyVar2, and FuzzyVar3 are FUZZYVAR objects. ConcatOp is a concatenation operator out {AND, OR}. A brief example shows the use of FUZZYCONCAT:

```
FUZZYVAR fuzzy_variable parameterset1
FUZZYVAR fuzzy_constraint1 parameterset2
FUZZYVAR fuzzy_constraint2 parameterset2
FUZZYVAR fuzzy_constraint3 parameterset2
SET fuzzy_variable = 15
FUZZYCOMPARE fuzzy_constraint1 IS
fuzzy_variable > 10
FUZZYCOMPARE fuzzy_constraint2 IS
fuzzy_variable < 20
FUZZYCONCAT fuzzy_constraint3 IS
fuzzy_constraint1 AND fuzzy_constraint2
EVAL fuzzy_constraint3
```

5.5.7 TABLE

Whenever a FUZZYCOMPARE or FUZZYCONCAT (cf. Section 5.5.6, page 144) statement is used, we have to include a table where the correspondings for all linguistic terms for comparison and concatenation is defined. The statement to do that is

> TABLE File_Name

Such tables have to meet the following structure:

CONCAT **ConcatOperator**	*parameterset1*	*parameterset2*	*parameterset3*
	<list_of_lingterms>	<list_of_lingterms>	<list_of_lingterms>

Table 5.5: Fuzzycompare

COMPARE	**CompareOperator**	*parameterset_in* <list_of_lingterms>	*parameterset_out* <list_of_lingterms>

Table 5.6: Fuzzyconcat

To concretize this concept, we will give a sample for such a table, according to the example we started in 5.5.1:

CONCAT AND fuzzy1 fuzzy1 fuzzy1

```
low low low
low medium low
low high medium
medium low low
medium medium medium
medium high medium
high low medium
high medium medium
high high high
```

OR fuzzy1 fuzzy1 fuzzy1

```
low low low
low medium medium
low high high
medium low medium
medium medium medium
medium high high
high low high
high medium high
high high high
```

COMPARE = fuzzy1 fuzzy2

```
low bad
medium good
high bad
```

```
# fuzzy1 fuzzy2
low good
medium bad
high good
< fuzzy1 fuzzy2
low good
medium neutral
high bad
> fuzzy1 fuzzy2
low bad
medium neutral
high good
```

Whenever we want to use new operators or new linguistic terms, the appropriate entries in such a file have to be performed. In this example `fuzzy1` and `fuzzy2` are names of `PARAMETERSET` (cf. Section 5.5.2, page 137) objects, so for each `PARAMETERSET` there should exist the corresponding entries in the table if we want to state constraints on `FUZZYVAR` objects declared with such a `PARAMETERSET`.

5.5.8 EVALTREE

The `EVALTREE` structure can be seen as a hierarchical structure for aggregating several evaluated constraints in a second evaluation step. One way to get an ¨overall¨ evaluation for a set of evaluated constraints would be the explicit conjunction or disjunction of all constraints by a set of ¨cascaded¨ `FUZZYCONCAT` (cf. Section 5.5.6, page 144) statements.

Real world applications and the resulting, usually very huge numbers of constraints make clear, that this is not a very smart way to achieve this result. To make such aggregations easier, CSI(C&F) offers the possibility to build up an `EVALTREE` structure, where single evaluations (`FUZZYVAR` objects) can be inserted and aggregated to an *overall aggregation* for *each branch* in this structure, and for the *whole tree* too. The declaration of such a tree has the following format:

```
EVALTREE
ParameterSet_Name
```

```
Overall_Name
(Branch_Name ConcatOperator)*
```
END

where `Overall_Name` is the name the evaluation tree should take. Obviously all evaluated `FUZZYVAR` (cf. Section 5.5.1, page 135) objects have to be of the same type, i.e. they have all to be assigned to the same `PARAMETERSET` object. Otherwise they could not be aggregated in a second step. For this the second entry must be the name of the desired `PARAMETERSET`. The following part describes the branches of the tree; the ConcatOperator (AND, OR) makes explicit the way this branch is aggregated to the overall evaluation.

The single branches of such a `EVALTREE` object are containers for groups of evaluated constraints that are of the same type. Although there are no restrictions where and how a `FUZZYVAR` object must be inserted in such a tree, it is a matter of semantically correctness where and how we insert an evaluated `FUZZYVAR`.

In section 5.5.9 we will see, how such insertion can be done by the use of the `AGGREGATE` (cf. Section 5.5.9, page 149) statement.

To make transparent the use of the `EVALTREE` structure we give a concrete example stemming from scheduling applications:

EVALTREE

`My_parameterset`

`Quality_of_plan`

`All_Capacity` **OR**

`All_DueDate` **AND**

`All_Compatibility` **AND**

END

CSI(C&F) provides the well known `EXPORT` structure for `EVALTREE` too. The data stored in such a tree can be written to a file by the use of

EXPORT EVALTREE `File_Name`

The resulting file holds the aggregated node results as well as the single evaluations of `FUZZYVAR` objects in the leaves. In addition to this we can specify a comment for each `FUZZYVAR` we want to insert. This makes it possible to link additional information to a single evaluation score in the leaves.

5.5.9 AGGREGATE

Inserting an evaluated `FUZZYVAR` in a `EVALTREE` structure can be achieved by the statement

> **AGGREGATE** Branch_Name ConcatOperator FuzzyVar Comment

where Branch_Name is the name of the branch of `EVALTREE` (cf. Section 5.5.8, page 147) to which the `FUZZYVAR` (cf. Section 5.5.1, page 135) FuzzyVar should be aggregated using a ConcatOperator out of {AND, OR}. Comment is a userdefinable string that is not allowed to include spaces. According to the example in section 5.5.8 such a statement could be:

FUZZYVAR `Cap_Constraint1 My_parameterset`

FUZZYVAR `DueDate_Constraint1 My_parameterset`

FUZZYVAR `Comp_Constraint1 My_parameterset`

FUZZYCOMPARE `Cap_Constraint1` **IS** ...

FUZZYCOMPARE `DueDate_Constraint1` **IS** ...

FUZZYCOMPARE `Comp_Constraint1` **IS** ...

EVAL `Cap_Constraint1`

EVAL `DueDate_Constraint1`

EVAL `Comp_Constraint1`

`/* Do the same for all other constraints ... */`

`/* Add the evaluated nodes to EVALTREE */`

AGGREGATE `All_Capacity` **OR** `Cap_Constraint1 no_comment`

AGGREGATE `All_DueDate` **AND** `DueDate_Constraint1 no_comment`

AGGREGATE `All_Capacity` **AND** `Cap_Constraint1 no_comment`

`/* Do the same for all other constraints ... */`

In this example,`Cap_Constraint1` is inserted by the conjunction `OR` in the branch `All_Capacity`. In the same step the node `All_Capacity` with a respective new evaluation is aggregated to the root node `Quality_of_plan` by disjunction. After insertion of some farther `FUZZYVAR` objects, the resulting `EVALTREE` structure for the example started in 5.5.9 could be the following one:

/* #: 3 */

Branch: **Quality_of_plan** Crisp value: **24.4836**

--------------------<End of branch: Quality_of_plan>

Branch: **All_Capacity** Crisp value: **50**

/* #: 6 */

Cap_Constraint1 no_comment 51.9002

Cap_Constraint2 no_comment 23.7656

Cap_Constraint3 no_comment 33.2249

Cap_Constraint4 no_comment 39.1101

Cap_Constraint5 no_comment 46.2728

Cap_Constraint6 no_comment 52.6279

--------------------<End of branch: All_Capacity >

Branch: **All_DueDate** Crisp value: **22.067**

/* #: 7 */

DueDate_Constraint1 no_comment 11.5385

DueDate_Constraint2 no_comment 50

DueDate_Constraint3 no_comment 42.8542

DueDate_Constraint4 no_comment 18.3544

DueDate_Constraint5 no_comment 22.067

DueDate_Constraint6 no_comment 88.4615

DueDate_Constraint7 no_comment 42.147

--------------------<End of branch: All_DueDate >

Branch: **All_Compatibility** Crisp value: **24.0933**

/* #: 3 */

Comp_Constraint1 no_comment 50

Comp_Constraint2 no_comment 79.8842

Comp_Constraint3 no_comment 11.5385

--------------------<End of branch:
All_Compatibility >

The results stored in this structure could further be used , e.g. by intelligent schedulers, to make configuration changes on the plan in order to eliminate bad evaluations and to achieve a higher overall evaluation score. A detailed example can be found in 5.7. To visualize the concept of `EVALTREE` and `AGGREGATE`, we give a more graphical representation of the example we have introduced in this section:

5.6 Keywords

These are the keywords of CSI(C&F) in alphabetical order:

ABS

ACCURACY

AGGREGATE

ALL

AND

ASS_COMP_AND

ASS_COMP_OR

CALL

CERTAINTY

COG

COS

DEFUZZY-FICATION

DELETE

DILATATION

DISTRIBUTION

ELSE

END

ENDIF

ENDLOOP

EVALTREE

EXIT

EXITLOOP

EXPORT
FOR
FUZZY_AND
FUZZY_OR
FUZZYCOMPARE
FUZZYCONCAT
FUZZYVAR
GAMMA
GOTO
IF
IMPORT
IMPORTANCE
INFERENCE
INPUT
IS
LABEL
LET
LOG
LOOP
LOWER
MAX
MAXIMUM
MEDIAN
MEMDUMP
MIN
MINIMUM
MUL
NEXT
NO_DFOP
NO_FOP
OPERATORSET

OR
PARAMETERSET
PRINT
PROCEDURE
PSLIST
RULESET
SET
SIN
SQR
SRT
STEP
SUM
TERMS
THEN
TO
UPPER
VARIABLES

5.7 An Application Example

In this section we will give an example for fuzzy constraint evaluation, stemming from the steelmaking environment as described in Chapter 5. In order not to get lost by the quite complex application properties, we assume a reduced process complexity. In a first step we will state the CSI(C&F) script. Beside the basic structures (cf. Section 5.3.3, page 127) we have seen in 5.5 the structure of the domain data will be also described.

```
/* Author : Ulrich Santa
File : LD3.SCR
Project : DynaFLIP++: CSI(C&F)
Start : 13.06.1996
Last Update : 24.07.1996
Comment : Example for fuzzy constraint evaluation in
a steelmaking plant
```

```
*/
/* import domain data */
IMPORT VARIABLES ld3.dat
/* define table for automatic rule generation */
TABLE ld3.table
/* define OperatorSet for fuzzy operators */
OPERATORSET
IMPORTANCE MINIMUM
AND MINIMUM
OR MAXIMUM
CERTAINTY MINIMUM
INFERENCE MINIMUM
ACCUMULATE MINIMUM
DEFUZZIFICATION COG
GAMMA 1
ACCURACY 10
END
/* define ParameterSets for fuzzy variables */
PARAMETERSET
NAME fuzzy_time
LOWER -1000
UPPER 1000
MEDIAN 0
DISTRIBUTION 0.1
DILATATION 0.5
IMPORTANCE 1.0
TERMS NB N Z P PB
END
PARAMETERSET
NAME fuzzy_chem
LOWER -3
```

UPPER `3`
MEDIAN `0`
DISTRIBUTION `0.7`
DILATATION `0.7`
IMPORTANCE `1`
TERMS `NB N Z P PB`
END
PARAMETERSET
NAME `fuzzy_weight`
LOWER `-100`
UPPER `100`
MEDIAN `0`
DISTRIBUTION `1.0`
DILATATION `0.5`
IMPORTANCE `1`
TERMS `NB N Z P PB`
END
PARAMETERSET
NAME `fuzzy_format`
LOWER `-10`
UPPER `10`
MEDIAN `0`
DISTRIBUTION `1.0`
DILATATION `0.5`
IMPORTANCE `1`
TERMS `NB N Z P PB`
END
PARAMETERSET
NAME `fuzzy_vacuum`
LOWER `-5`
UPPER `5`

MEDIAN `0`
DISTRIBUTION `0.5`
DILATATION `0.5`
IMPORTANCE `1`
TERMS `NB N Z P PB`
END
PARAMETERSET
NAME `fuzzy_satisfaction`
LOWER `0`
UPPER `100`
MEDIAN `50`
DISTRIBUTION `0.7`
DILATATION `1`
IMPORTANCE `1`
TERMS `B Z G`
END
`/* define fuzzy variables */`
FUZZYVAR `time fuzzy_time`
FUZZYVAR `chem_si fuzzy_chem`
FUZZYVAR `chem_al fuzzy_chem`
FUZZYVAR `chem_mg fuzzy_chem`
FUZZYVAR `weight fuzzy_weight`
FUZZYVAR `format fuzzy_format`
FUZZYVAR `vacuum fuzzy_vacuum`
FUZZYVAR `c_vacuum fuzzy_satisfaction`
FUZZYVAR `c_format fuzzy_satisfaction`
FUZZYVAR `c_weight fuzzy_satisfaction`
FUZZYVAR `c_whole_duration fuzzy_satisfaction`
FUZZYVAR `c_tundish fuzzy_satisfaction`
FUZZYVAR `c_due_date fuzzy_satisfaction`
FUZZYVAR `c_chem_mg fuzzy_satisfaction`

```
FUZZYVAR c_chem_si fuzzy_satisfaction
FUZZYVAR c_chem_al fuzzy_satisfaction
FUZZYVAR c_chem_si_al fuzzy_satisfaction
FUZZYVAR c_chem_all fuzzy_satisfaction
/* define structure of evaluation tree */
EVALTREE
LD3_ALL
fuzzy_satisfaction
CHEM OR
DUE_DATE AND
TUNDISH OR
WEIGHT AND
FORMAT AND
DURATION AND
VACUUM AND
END
FOR i = 1 TO domain.aggregates STEP 1
/* initialize casting duration on aggregate[i] */
LET aggregate[i].duration = 800
LET tundish_duration = 0
/* process on all items of the actual aggregate */
FOR j = 1 TO aggregate[i].items STEP 1
/* add duration of item to duration of aggregate */
LET aggregate[i].duration = aggregate[i].duration+item[j].duration
LET tundish_duration = tundish_duration
+item[j].duration
/* evaluate due_date-constraint */
IF (item[j].name="JOB") THEN
LET difference = aggregate[i].duration-item[j].due_date
SET time = difference
FUZZYCOMPARE c_due_date IS time < 0
```

```
EVAL c_due_date
AGGREGATE DUE_DATE AND c_due_date
aggregate[i].job[j]
/* check degassing compatibility of subsequent jobs
*/
IF (j=1) THEN
LET degassing = item[j].degassing
ENDIF
IF (degassing=item[j].veredelung) THEN
SET vacuum = 1
ELSE
SET vacuum = -(1)
ENDIF
FUZZYCOMPARE c_vacuum IS vacuum > 0
EVAL c_vacuum
AGGREGATE VACUUM AND c_vacuum
aggregate[i].job[j]
LET degassing = item[j].degassing
ENDIF
LET succ = j+1
IF (succ>aggregate[i].items) THEN
/* no items left in this aggregate*/
ELSE
IF ((item[j].name=¨JOB¨)&(item[succ].name=¨JOB¨))
THEN
LET si.min = MAX{item[j].si.min,item[succ].si.min}
LET si.max = MIN{item[j].si.max,item[succ].si.max}
LET si.overlap = si.max-si.min
SET chem_si = si.overlap
FUZZYCOMPARE c_chem_si IS chem_si > 1.4
LET al.min = MAX{item[j].al.min,item[succ].al.min}
LET al.max = MIN{item[j].al.max,item[succ].al.max}
```

```
LET al.overlap = al.max-al.min
SET chem_al = al.overlap
FUZZYCOMPARE c_chem_al IS chem_al > 0.6
LET mg.min = MAX{item[j].mg.min,item[succ].mg.min}
LET mg.max = MIN{item[j].mg.max,item[succ].mg.max}
LET mg.overlap = mg.max-mg.min
SET chem_mg = mg.overlap
FUZZYCOMPARE c_chem_mg IS chem_mg > 0.5
FUZZYCONCAT c_chem_si_al IS c_chem_al AND
c_chem_si
FUZZYCONCAT c_chem_all IS c_chem_si_al OR
c_chem_mg
EVAL c_chem_all
AGGREGATE CHEM AND c_chem_all
aggregate[i].job[j][j+1]
/* subsequent jobs should have the same weight */
SET weight = item[j].weight-item[succ].weight
FUZZYCOMPARE c_gewicht IS weight = 0
EVAL c_weight
AGGREGATE WEIGHT AND c_weight
aggregate[i].job[j][j+1]
/* only negative format changes should be made */
LET formatchange = item[j].format-item[succ].format
SET format = formatchange
FUZZYCOMPARE c_format IS format > 0
EVAL c_format
AGGREGATE FORMAT AND c_format
aggregate[i].job[j][j+1]
ENDIF
ENDIF
IF (item[j].name="TUNDISH") THEN
LET tundish_diff = tundish_duration-240
```

```
LET tundish_duration = 0
SET time = tundish_diff
FUZZYCOMPARE c_tundish IS time = 0
EVAL c_tundish
AGGREGATE TUNDISH AND c_tundish
aggregat[i].tundish[j]
ENDIF
NEXT j
/* overall casting time should be around 2050 */
SET time = aggregate[i].duration-2050
FUZZYCOMPARE c_whole_duration IS time = 0
EVAL c_whole_duration
AGGREGATE DURATION AND c_whole_duration
whole_duration
NEXT i
PRINT "OVERALL SCORE: " PRINT LD3_ALL
EXPORT RULESET ld3.ruleset
EXPORT VARIABLES ld3.varout
EXPORT EVALTREE ld3.eval
EXIT
```

This program evaluates different groups of constraints for a list of jobs on a continuous caster in a steelmaking plant. The constraints are

- *temporal constraints*
- *chemical constraints*
- *format constraints*
- *weight constraints*
- *capacity constraints*

On the next pages you can find the tables, which this program needs for automatic rule generation, and the hierarchical`EVALTREE` structure build up by the program and returned to the scheduler. The generated rules and the structure of the data imported from `ld3.scr` are also given on the next pages.

COMPARE fuzzy_vacuum fuzzy_satisfaction

```
< PB B
P B
Z Z
N G
NB G
> PB G
P G
Z Z
N B
NB B
= PB B
P Z
Z G
N Z
NB B
# PB G
P Z
Z B
N Z
NB B
```

COMPARE fuzzy_format fuzzy_satisfaction

```
< PB B
P B
Z Z
N G
NB G
> PB G
P G
Z Z
N B
```

```
NB B
= PB B
P Z
Z G
N Z
NB B
# PB G
P Z
Z B
N Z
NB B
```

COMPARE fuzzy_weight fuzzy_satisfaction

```
< PB B
P B
Z Z
N G
NB G
> PB G
P G
Z Z
N B
NB B
= PB B
P Z
Z G
N Z
NB B
# PB G
P Z
Z B
N Z
```

```
NB B
COMPARE fuzzy_time fuzzy_satisfaction
< PB B
P B
Z Z
N G
NB G
> PB G
P G
Z Z
N B
NB B
= PB B
P Z
Z G
N Z
NB B
# PB G
P Z
Z B
N Z
NB B
COMPARE fuzzy_chem fuzzy_satisfaction
< PB B
P B
Z Z
N G
NB G
> PB G
P G
Z Z
```

```
N B
NB B
= PB B
P Z
Z G
N Z
NB B
# PB G
P Z
Z B
N Z
NB B
```

CONCAT fuzzy_satisfaction fuzzy_satisfaction fuzzy_satisfaction

```
AND G G G
G Z Z
G B B
Z G Z
Z Z Z
Z B B
B G B
B Z B
B B B
OR G G G
G Z G
G B G
Z G G
Z Z Z
Z B Z
B G G
B Z Z
```

```
B B B
/* #: 8 */
Branch: LD3_ALL Crisp value: 24.4836
----------------------<End of branch: LD3_ALL>
Branch: CHEM Crisp value: 50
/* #: 6 */
c_chem_all aggregate[1].job[1][2] 51.9002
c_chem_all aggregate[1].job[2][3] 23.7656
c_chem_all aggregate[1].job[3][4] 33.2249
c_chem_all aggregate[1].job[6][7] 39.11
c_chem_all aggregate[1].job[9][10] 46.2728
c_chem_all aggregate[1].job[10][11] 52.6279
----------------------<End of branch: CHEM>
Branch: DUE_DATE Crisp value: 22.067
/* #: 9 */
c_due_date aggregate[1].job[1] 11.5385
c_due_date aggregate[1].job[2] 50
c_due_date aggregate[1].job[3] 50
c_due_date aggregate[1].job[4] 42.8542
c_due_date aggregate[1].job[6] 18.3544
c_due_date aggregate[1].job[7] 22.067
c_due_date aggregate[1].job[9] 88.4615
c_due_date aggregate[1].job[10] 11.5385
c_due_date aggregate[1].job[11] 42.147
----------------------<End of branch: DUE_DATE>
Branch: TUNDISH Crisp value: 24.0933
/* #: 3 */
c_tundish aggregate[1].tundish[5] 50
c_tundish aggregate[1].tundish[8] 79.8842
c_tundish aggregate[1].tundish[12] 11.5385
----------------------<End of branch: TUNDISH>
```

Branch: **WEIGHT** Crisp value: **39.7022**

/* #: 6 */

c_weight aggregate[1].job[1][2] 42.147

c_weight aggregate[1].job[2][3] 39.7561

c_weight aggregate[1].job[3][4] 88.4615

c_weight aggregate[1].job[6][7] 68.9262

c_weight aggregate[1].job[9][10] 50

c_weight aggregate[1].job[10][11] 50

----------------------<End of branch: WEIGHT>

Branch: **FORMAT** Crisp value: **22.067**

/* #: 6 */

c_format aggregate[1].job[1][2] 42.147

c_format aggregate[1].job[2][3] 50

c_format aggregate[1].job[3][4] 68.9262

c_format aggregate[1].job[6][7] 68.8199

c_format aggregate[1].job[9][10] 88.4615

c_format aggregate[1].job[10][11] 22.067

----------------------<End of branch: FORMAT>

Branch: **DURATION** Crisp value: **68.8199**

/* #: 1 */

c_whole_duration whole_duration 68.8199

----------------------<End of branch: DURATION>

Branch: **VACUUM** Crisp value: **11.5385**

/* #: 9 */

c_vacuum aggregat[1].job[1] 88.4615

c_vacuum aggregat[1].job[2] 88.4615

c_vacuum aggregat[1].job[3] 11.5385

c_vacuum aggregat[1].job[4] 11.5385

c_vacuum aggregat[1].job[6] 88.4615

c_vacuum aggregat[1].job[7] 88.4615

c_vacuum aggregat[1].job[9] 11.5385

```
c_vacuum aggregat[1].job[10] 88.4615
c_vacuum aggregat[1].job[11] 11.5385
----------------------<End of branch:  VACUUM>
/* #:  118 */
domain.aggregates 1 1
domain.name ¨LD3¨ 0
aggregate[1].items 12 12
aggregate[1].name ¨CC_3¨ 0
item[1].name ¨JOB¨ 0
item[1].format 10 10
item[1].due_date 700 700
item[1].weight 20 20
item[1].duration 60 60
item[1].degassing ¨A¨ 0
item[1].al.min 0.1 0.1
item[1].al.max 0.8 0.8
item[1].mg.min 1 1
item[1].mg.max 1.2 1.2
item[1].si.min 0.1 0.1
item[1].si.max 1.5 1.5
item[2].name ¨JOB¨ 0
item[2].format 12 12
item[2].due_date 880 880
item[2].weight 90 90
item[2].duration 30 30
item[2].degassing ¨A¨ 0
item[2].al.min 0.2 0.2
item[2].al.max 0.7 0.7
item[2].mg.min 0.8 0.8
item[2].mg.max 1.2 1.2
item[2].si.min 0.1 0.1
```

```
item[2].si.max 1.5 1.5
item[3].name ¨JOB¨ 0
item[3].format 11 11
item[3].due_date 945 945
item[3].weight 30 30
item[3].duration 50 50
item[3].degassing ¨C¨ 0
item[3].al.min 0.3 0.3
item[3].al.max 0.8 0.8
item[3].mg.min 1.5 1.5
item[3].mg.max 1.8 1.8
item[3].si.min 0.2 0.2
item[3].si.max 1.5 1.5
item[4].name ¨JOB¨ 0
item[4].format 9 9
item[4].due_date 955 955
item[4].weight 30 30
item[4].duration 45 45
item[4].degassing ¨B¨ 0
item[4].al.min 0.3 0.3
item[4].al.max 1.6 1.6
item[4].mg.min 1.1 1.1
item[4].mg.max 2.2 2.2
item[4].si.min 0.8 0.8
item[4].si.max 1.2 1.2
item[5].name ¨TUNDISH¨ 0
item[5].duration 100 100
item[6].name ¨JOB¨ 0
item[6].format 10 10
item[6].due_date 1100 1100
item[6].weight 60 60
```

```
item[6].duration 80 80
item[6].degassing ¨B¨ 0
item[6].al.min 0.3 0.3
item[6].al.max 0.8 0.8
item[6].mg.min 0.9 0.9
item[6].mg.max 1.1 1.1
item[6].si.min 0.2 0.2
item[6].si.max 1.5 1.5
item[7].name ¨JOB¨ 0
item[7].format 7 7
item[7].due_date 1130 1130
item[7].weight 30 30
item[7].duration 45 45
item[7].degassing ¨B¨ 0
item[7].al.min 0.1 0.1
item[7].al.max 0.4 0.4
item[7].mg.min 0.9 0.9
item[7].mg.max 1.2 1.2
item[7].si.min 1.2 1.2
item[7].si.max 1.5 1.5
item[8].name ¨TUNDISH¨ 0
item[8].duration 100 100
item[9].name ¨JOB¨ 0
item[9].format 8 8
item[9].due_date 1610 1610
item[9].weight 60 60
item[9].duration 260 260
item[9].degassing ¨E¨ 0
item[9].al.min 0.3 0.3
item[9].al.max 0.8 0.8
item[9].mg.min 0.9 0.9
```

```
item[9].mg.max 1.1 1.1
item[9].si.min 0.2 0.2
item[9].si.max 1.5 1.5
item[10].name ¨JOB¨ 0
item[10].format 3 3
item[10].due_date 1510 1510
item[10].weight 100 100
item[10].duration 100 100
item[10].degassing ¨E¨ 0
item[10].al.min 0.9 0.9
item[10].al.max 2.8 2.8
item[10].mg.min 0.9 0.9
item[10].mg.max 2.1 2.1
item[10].si.min 0.2 0.2
item[10].si.max 0.5 0.5
item[11].name ¨JOB¨ 0
item[11].format 10 10
item[11].due_date 1910 1910
item[11].weight 60 60
item[11].duration 260 260
item[11].degassing ¨B¨ 0
item[11].al.min 0.3 0.3
item[11].al.max 0.4 0.4
item[11].mg.min 0.9 0.9
item[11].mg.max 1.0 1.0
item[11].si.min 1.2 1.2
item[11].si.max 1.5 1.5
item[12].name ¨TUNDISH¨ 0
item[12].duration 100 100
```

RULESET

```
/* #:  13 */
```

```
/* #:  5 */

IF /* #:  1 */ c_due_date:time< IS PB THEN c_due_date
IS B WITH CERTAINTY 1.

IF /* #:  1 */ c_due_date:time< IS P THEN c_due_date
IS B WITH CERTAINTY 1.

IF /* #:  1 */ c_due_date:time< IS Z THEN c_due_date
IS Z WITH CERTAINTY 1.

IF /* #:  1 */ c_due_date:time< IS N THEN c_due_date
IS G WITH CERTAINTY 1.

IF /* #:  1 */ c_due_date:time< IS NB THEN c_due_date
IS G WITH CERTAINTY 1.

/* #:  5 */

IF /* #:  1 */ c_vacuum:vacuum> IS PB THEN c_vacuum
IS G WITH CERTAINTY 1.

IF /* #:  1 */ c_vacuum:vacuum> IS P THEN c_vacuum IS
G WITH CERTAINTY 1.

IF /* #:  1 */ c_vacuum:vacuum> IS Z THEN c_vacuum IS
Z WITH CERTAINTY 1.

IF /* #:  1 */ c_vacuum:vacuum> IS N THEN c_vacuum IS
B WITH CERTAINTY 1.

IF /* #:  1 */ c_vacuum:vacuum> IS NB THEN c_vacuum
IS B WITH CERTAINTY 1.

/* #:  5 */

IF /* #:  1 */ c_chem_si:chem_si> IS PB THEN c_chem_si
IS G WITH CERTAINTY 1.

IF /* #:  1 */ c_chem_si:chem_si> IS P THEN c_chem_si
IS G WITH CERTAINTY 1.

IF /* #:  1 */ c_chem_si:chem_si> IS Z THEN c_chem_si
IS Z WITH CERTAINTY 1.

IF /* #:  1 */ c_chem_si:chem_si> IS N THEN c_chem_si
IS B WITH CERTAINTY 1.

IF /* #:  1 */ c_chem_si:chem_si> IS NB THEN c_chem_si
IS B WITH CERTAINTY 1.

/* #:  5 */
```

```
IF /* #:  1 */ c_chem_al:chem_al> IS PB THEN c_chem_al
IS G WITH CERTAINTY 1.

IF /* #:  1 */ c_chem_al:chem_al> IS P THEN c_chem_al
IS G WITH CERTAINTY 1.

IF /* #:  1 */ c_chem_al:chem_al> IS Z THEN c_chem_al
IS Z WITH CERTAINTY 1.

IF /* #:  1 */ c_chem_al:chem_al> IS N THEN c_chem_al
IS B WITH CERTAINTY 1.

IF /* #:  1 */ c_chem_al:chem_al> IS NB THEN c_chem_al
IS B WITH CERTAINTY 1.

/* #:  5 */

IF /* #:  1 */ c_chem_mg:chem_mg> IS PB THEN c_chem_mg
IS G WITH CERTAINTY 1.

IF /* #:  1 */ c_chem_mg:chem_mg> IS P THEN c_chem_mg
IS G WITH CERTAINTY 1.

IF /* #:  1 */ c_chem_mg:chem_mg> IS Z THEN c_chem_mg
IS Z WITH CERTAINTY 1.

IF /* #:  1 */ c_chem_mg:chem_mg> IS N THEN c_chem_mg
IS B WITH CERTAINTY 1.

IF /* #:  1 */ c_chem_mg:chem_mg> IS NB THEN c_chem_mg
IS B WITH CERTAINTY 1.

/* #:  9 */

IF /* #:  2 */ c_chem_al IS G AND c_chem_si IS G THEN
c_chem_si_al IS G WITH CERTAINTY 1.

IF /* #:  2 */ c_chem_al IS G AND c_chem_si IS Z THEN
c_chem_si_al IS Z WITH CERTAINTY 1.

IF /* #:  2 */ c_chem_al IS G AND c_chem_si IS B THEN
c_chem_si_al IS B WITH CERTAINTY 1.

IF /* #:  2 */ c_chem_al IS Z AND c_chem_si IS G THEN
c_chem_si_al IS Z WITH CERTAINTY 1.

IF /* #:  2 */ c_chem_al IS Z AND c_chem_si IS Z THEN
c_chem_si_al IS Z WITH CERTAINTY 1.

IF /* #:  2 */ c_chem_al IS Z AND c_chem_si IS B THEN
c_chem_si_al IS B WITH CERTAINTY 1.
```

```
IF /* #: 2 */ c_chem_al IS B AND c_chem_si IS G THEN
c_chem_si_al IS B WITH CERTAINTY 1.

IF /* #: 2 */ c_chem_al IS B AND c_chem_si IS Z THEN
c_chem_si_al IS B WITH CERTAINTY 1.

IF /* #: 2 */ c_chem_al IS B AND c_chem_si IS B THEN
c_chem_si_al IS B WITH CERTAINTY 1.

/* #: 9 */

IF /* #: 2 */ c_chem_si_al IS G OR c_chem_mg IS G THEN
c_chem_all IS G WITH CERTAINTY 1.

IF /* #: 2 */ c_chem_si_al IS G OR c_chem_mg IS Z THEN
c_chem_all IS G WITH CERTAINTY 1.

IF /* #: 2 */ c_chem_si_al IS G OR c_chem_mg IS B THEN
c_chem_all IS G WITH CERTAINTY 1.

IF /* #: 2 */ c_chem_si_al IS Z OR c_chem_mg IS G THEN
c_chem_all IS G WITH CERTAINTY 1.

IF /* #: 2 */ c_chem_si_al IS Z OR c_chem_mg IS Z THEN
c_chem_all IS Z WITH CERTAINTY 1.

IF /* #: 2 */ c_chem_si_al IS Z OR c_chem_mg IS B THEN
c_chem_all IS Z WITH CERTAINTY 1.

IF /* #: 2 */ c_chem_si_al IS B OR c_chem_mg IS G THEN
c_chem_all IS G WITH CERTAINTY 1.

IF /* #: 2 */ c_chem_si_al IS B OR c_chem_mg IS Z THEN
c_chem_all IS Z WITH CERTAINTY 1.

IF /* #: 2 */ c_chem_si_al IS B OR c_chem_mg IS B THEN
c_chem_all IS B WITH CERTAINTY 1.

/* #: 5 */

IF /* #: 1 */ c_weight:weight= IS PB THEN c_weight IS
B WITH CERTAINTY 1.

IF /* #: 1 */ c_weight:weight= IS P THEN c_weight IS
Z WITH CERTAINTY 1.

IF /* #: 1 */ c_weight:weight= IS Z THEN c_weight IS
G WITH CERTAINTY 1.

IF /* #: 1 */ c_weight:weight= IS N THEN c_weight IS
Z WITH CERTAINTY 1.
```

```
IF /* #:  1 */ c_weight:weight= IS NB THEN c_weight IS
B WITH CERTAINTY 1.

/* #:  5 */

IF /* #:  1 */ c_format:format> IS PB THEN c_format
IS G WITH CERTAINTY 1.

IF /* #:  1 */ c_format:format> IS P THEN c_format IS
G WITH CERTAINTY 1.

IF /* #:  1 */ c_format:format> IS Z THEN c_format IS
Z WITH CERTAINTY 1.

IF /* #:  1 */ c_format:format> IS N THEN c_format IS
B WITH CERTAINTY 1.

IF /* #:  1 */ c_format:format> IS NB THEN c_format
IS B WITH CERTAINTY 1.
```

... etc.

Chapter 6

OptiFLIP++

OptiFLIP++ is a heuristic optimizing library. Several heuristics have so far been implemented and tested, namely:

- a tabu list min-conflicts repair based hill climbing heuristic,
- a min-conflicts repair based iterative deepening heuristic,
- a min-conflicts repair based random search hill climbing heuristic,
- a min-conflicts repair based genetic algorithm heuristic.

Results using the repair based algorithms implemented in StarFLIP++ have so far yielded intermediate results, almost equal in performance. Simple random search repair is easier trapped in suboptimal solutions. Starting from a better initial instantiation (for example from the solution found with the constructive approach) does not really influence the results. More statistics and a detailed analysis of the different algorithms can be found in Dorn et al. . All repair based heuristics are much faster and yield better results than the constructive approach that was evaluated using the same configuration parameters.

Chapter 7

GenFLIP++ library description

7.1 Contents

   (b) Mutation (cf. Section 7.6.2)
      i. Position-based mutation (cf. Section 7.6.2.1)
      ii. Order-based mutation (cf. Section 7.6.2.2)
   (c) Crossover (cf. Section 7.6.3)
      i. Position-based crossover (cf. Section 7.6.3.1)
      ii. Order-based crossover (cf. Section 7.6.3.2)

6. GENFLIP++ (cf. Section 7.7)
   (a) Interface (cf. Section 7.7.1)
   (b) Constants (cf. Section 7.7.2)
   (c) Methods (cf. Section 7.7.3)

7. GENFLIP++-The User Interface (cf. Section 7.8)
   (a) Strategy (cf. Section 7.8.1)
   (b) Crossover (cf. Section 7.8.2)
   (c) Mutation (cf. Section 7.8.3)
   (d) Repair-Steps (cf. Section 7.8.4)

8. References (cf. Section 7.9)

7.2 Overview

This document describes a genetic algorithm that has been implemented to optimize a given schedule within the FLIP++-Environment. The FLIP++ project intends to solve scheduling-problems by means of Fuzzy-Logic. Fuzzy-Logic is used to describe the constraints that must be taken into account to create a valid schedule and to evaluate it.

7.3 Task

The CD-Labor for Experten Systeme has a library under development which is able to solve scheduling-problems by means of Fuzzy-Logic.

One part of this library is the OPTIFLIP-modul which has the task of making chances in the scheduling in such a way that the result is an optimized schedule that does not violate given constraints.

Several methods will be implemented by the CD-Labor. The implementation of a genetic algorithm becomes interesting when we try to find out whether this method or specialized search-methods are more sophisticated in solving scheduling problems.

A Part of DOMFLIP contains the optimization algorithms whereas the genetic algorithm is implemented as a seperate object that is then linked to the domflip-object.

7.4 Embedding

The genetic algorithm must be included (#include "genalg.h") in the domflip.h-File.

In domlib.h you can already find the enumeration-type optmethode that has a "GeneticAlg" value. To set the Genetic Algorithm as the default optimization method you have to call the set_opt_meth(GeneticAlg) of the Domain-object.

7.5 Genetic Algorithm

7.5.1 History

The field of genetic algorithms has been founded by John Holland. His publication of Adaptation in Natural and Artificial Systems (Holland 1975 (cf. Section 7.9)) describes the ability of simple representations (bit strings) to encode complicated structures and simple transformations which have enough power to improve such structures. Holland showed that with the proper control structure, rapid improvements of bit strings could occur under certain transformations, so that a population of bit strings could be made to evolve as a population of animals would.

An important formal result was that even in large and complicated search spaces, genetic algorithms would tend to converge on solutions that were globally optimal or nearly so.

7.5.2 Methods

A genetic algorithm to solve a problem must have 5 components:

- a chromosomal representation of solutions to the problem
- a way to create an initial population of solutions

- an evaluation function that plays the role of the environment, rating solutions in term of their "fitness"
- genetic operators that alter the composition of children during reproduction
- values of the parameters that the genetic algorithm uses (population size, probabilities of applying genetic operators, etc.)

7.5.2.1 Representation:

In Holland's work chromosomes are bit strings. Some researchers have explored the uses of other representations, often in connection with industrial applications of genetic algorithms. Examples of other representations include embedded lists (for scheduling problems) and variable-element lists (for semiconductor layout).

7.5.2.2 Initialization:

Initialization routines vary. Moving from a randomly created population to a well-adapted population is a good test of the algorithm, since the critical features of the final solution will have been produced by the search and recombination mechanism of the algorithm, rather than the initialization procedure.

7.5.2.3 Evaluation Function:

There are a great number of evaluation functions that enhance or hinder a genetic algorithm's performance. An important question to be considered in designing an evaluation function is the implementation of constraints on solutions. Constraints that cannot be violated can be implemented by imposing great penalties on individuals that violate them.

7.5.2.4 Genetic Operators:

Genetic operators for bit string representations have been extensively studied, while operators for other representation types have not. A good part of the engineering task at present in applying genetic algorithms to industrial problems lies in choosing a chromosomal representation of solutions and a set of genetic operators.

7.5.3 Strength and Weaknesses:

Genetic algorithms promise to be a reusable, easy to implement and fast search method for vast search-spaces. No doubt that this is true, the performance and results of genetic algorithms depend very strong on the applied genetic operators and the evaluation function.

As there is no theory describing the right operators for a specific problem, designed a good genetic algorithm is some sort of black magic.

7.6 Genetic Algorithm and Scheduling

7.6.1 Breeding Strategies

7.6.1.1 Best n percent survive:

This breeding strategy assumes that the best n percent of all strings survive. This means that parents and offspring are seen as one set.

7.6.1.2 Best n percent descendants survive:

This strategy is slightly different from "best n percent survive" but this difference has a great impact on the overall population. Each step the GA does all parents die and just the best n percent of the offspring survive. This strategy comes close to the strategy nature uses but has the disadvantage that good solutions can be forgotten.

7.6.1.3 Proportional fitness survival:

This strategy assumes that strong (respectively good) individuals produce more offspring than less good individuals. Therefore the number of children is proportional to the fitness of the parents and the possibility to die is the same for the whole population.

7.6.1.4 Elitism:

This strategy is quite non-biological but produces good results in the case of computers. It assumes that the best individual survives no matter whether it is from the parent population or a child. Further, the best n percent of descendants survive. Should it be the case that the fittest individual of the new generation is worse than the unfittest individual of the parents generation, therefore the worst individual of the new generation will be replaced by the best individual of the parent generation.

7.6.2 Mutation

7.6.2.1 Position-based mutation:

The idea of the position-based mutation is to swap two genes in the string by random.

```
predecessor:            A B C D E F     swap A and F
                        *               *
successor:              F B C D E A
```

The position-based mutation can be written in pseudo-code as: pos-based()

```
pos1 is rand()
pos2 is rand()
(...,xpos2,.....,xpos1,....) is (...,xpos1,....,xpos2,....)
```

7.6.2.2 Order-based mutation:

The idea of the order-based mutation is to push one randomly selected gene in front of another randomly selected gen.

```
predecessor:            A B C D E F     push F in front of A
                        *1              *2
successor:              F A B C D E
```

The order-based mutation can be written in pseudo-code like this:

```
ord-based()
pos1 is rand()
pos2 is rand()
(...,xpos1,.....,xpos2,...) is (...,xpos2,xpos1,...)
```

7.6.3 Crossover

The idea of the crossover is that several characteristics of a number of elements are combined into a new string. This can be described for biological systems with the cutting of parent strings into several pieces and rearranging them into a new string by making a selection over the pieces.

This model can not be easily transformed for scheduling-problems as a gene in a schedule has to describe a specific job. Just cutting the parent strings and rearranging them could cause a duplication of certain jobs.

7.6.3.1 Position-based crossover:

The idea of the position-based crossover is to shift the gene to the position corresponding to that of the other parent.

```
parents:         offspring:
A B C D E F G    A G C D E F B     shift B in front of G
*                *
G F E D C B A    G A E D C B F     shift F in front of A
```

7.6.3.2 Order-based crossover

The idea behind the order-based crossover is to swap the genes in the order found at the other parent.

```
parents         offspring:
A B C D E F G   A G C D E F B      swap B with G
*               *
G F E D C B A   G A E D C B F      swap F with A
```

7.7 GENFLIP++

7.7.1 Interface

The interface of the genalg-object is implemented within the genalg.h-file. The following public methods are accessible:

set_crossover to define the crossover strategy the genetic algorithm will use. The default value is the "position-based crossover".

set_mutation to define the mutation strategy the genetic algorithm will use. The default value is the "position-based mutation".

set_strategie to define the survival-strategy the genetic algorithm will use. The default value is the "best n percent survive strategy".

set_percent_survivers to define how many individuals will survive. The meaning of the value is connected with the survival-strategy.

set_max_strings_number to define how many adults are allowed at one time.

set_max_childs_number to define how many children are allowed at one time.

set_max_init_mixture to define the maximum number of mutations that will be applied to the init-strings.

set_repair(true/false) to define whether the repair-steps given by DOM-FLIP will be applied or not.

set_string to copy a schedule into the genetic algorithm directly e(pointer-reference).

set_string_copy to copy a schedule into the genetic algorithm (a new instance within the genetic algorithm will be created).

init_start_strings to initialize the genetic algorithm. Always call this routine before one of the following routines.

one_step evaluates one genetic-step.

several_steps evaluates the given number of genetic-steps.

go_to_fitness evaluates a maximum number of genetic-steps (zero means infinity) and at the specified fitness.

get_string_with_highest_fitness returns the schedule with the highest fitness directly (pointer reference).

get_string_with_highest_fitness_copy returns the schedule with the highest fitness by creating a new instance.

7.7.2 Constants

The following constants are defined within the genalg.h-File.

- Kinds of mutations:
  - POSITION_BASED_MUTATION
  - ORDER_BASED_MUTATION
- Kinds of crossover:
  - POSITION_BASED_CROSSOVER
  - ORDER_BASED_CROSSOVER
- Kinds of survival strategies:
  - BEST_N_PERCENT_SURVIVE
  - BEST_N_PERCENT_DESCENDANTS_SURVIVE

7.7.3 Methods

The implementation of the genetic algorithm uses two double linked lists. One (called actual_generation) holds the parent-schedules and the schedules that survived a genetic algorithm step. The other (called children) holds the descendants of the parent-schedules after crossover and mutation.

An entry will be inserted in both listed sorted by the fitness to allow an easy implementation of the survival strategies.

7.8 GENFLIP++-The User Interface

7.8.1 Strategy

To select a strategy ,all the known strategies to the GA are listed in an "Listbox". You can only select one Strategy at one time.

One part of the strategy is to determine the maximum number of individuals and children for one step of the GA. Be careful in setting these values, as you can produce an infinite amount of individuals which will lead the program to stop when no more memory is available.

7.8.2 Crossover

To select the crossover all crossover-methods supported by the GA are listed in an "Listbox". You can only select one crossover-method at a time.

7.8.3 Mutation

To select the mutations that will by applied to our children all supported crossover-methods are listed in a "Listbox". Here you are allowed to make a multiple-selection.

7.8.4 Repair-Steps

To define that the repair-steps will be applied to a new element, a toggle-button is provided to switch this feature on and off.

7.9 References

- Wolfgang Slany, "Fuzzy Scheduling"; CD-Technical Report 94/66

- Mario Girsch, "Optimierung von Schedules mit Genetischen Algorithmen und Iterativer Vertiefung"
- John Holland, "Adaptation in Natural and Artificial Systems"; (Holland 1975)
- Lawrence Davis and Martha Steenstrup, "Genetic Algorithms and Simulated Annealing"
- David E. Glover, "Solving a Comples Keyboard Configuration Problem Through Generalized Adaptive Search"
- Grefenstette, "Incorporating Problems Specific Knowledge into Genetic Algorithms"

Appendix A

GNU Library General Public License

A.1 GNU LIBRARY GENERAL PUBLIC LICENSE

Version 2, June 1991

A.2 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users.

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor's protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect transforming the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License, which was designed for utility programs. This license, the GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary one; be sure to read it in full, and don't assume that anything in it is the same as in the ordinary license.

The reason we have a separate public license for some libraries is that they blur the distinction we usually make between modifying or adding to a program and simply using it. Linking a program with a library, without changing the library, is in some sense simply using the library, and is analogous to running a utility program or application program. However, in a textual and legal sense, the linked executable is a combined work, a derivative of the original library, and the ordinary General Public License treats it as such.

Because of this blurred distinction, using the ordinary General Public License for libraries did not effectively promote software sharing, because most developers did not use the libraries. We concluded that weaker conditions might promote sharing better.

However, unrestricted linking of non-free programs would deprive the users of those programs of all benefit from the free status of the libraries themselves. This Library General Public License is intended to permit developers of non-free programs to use free libraries, while preserving your freedom as a user of such programs to change the free libraries that are incorporated in them. (We have not seen how to achieve this as regards changes in header files, but we have achieved it as regards changes in the actual functions of the Library.) The hope is that this will lead to faster development of free libraries.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, while the latter only works together with the library.

Note that it is possible for a library to be covered by the ordinary General Public License rather than by this special one.

A.3 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Library General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- **a)** The modified work must itself be a software library.

- **b)** You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

- **c)** You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

- **d)** If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful. (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

- **a)** Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

- **b)** Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

- **c)** If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

- **d)** Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

- **a)** Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

- **b)** Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Library General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

A.4 How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found. *one line to give the library's name and an idea of what it does.*

```
Copyright (C)
```

year name of author

```
This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Library General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Library General Public License for more details.
```

```
You should have received a copy of the GNU Library General Public
License along with this library; if not, write to the
Free Software Foundation, Inc., 59 Temple Place - Suite 330,
Boston, MA  02111-1307, USA.
```

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in
the library ‘Frob’ (a library for tweaking knobs) written
by James Random Hacker.
```

signature of Ty Coon

```
 , 1 April 1990
Ty Coon, President of Vice
```

That’s all there is to it!

Appendix B

Publications on the StarFLIP++ project

B.1 Publications related to StarFLIP++ as a whole

- StarFLIP-Team: StarFLIP++ Version 1.0 A reusable iterative optimization library for combinatorial problems with fuzzy constraints. Technical Report DBAI-TR-97-11, Institute of Information Systems (E184-2), Vienna University of Technology, 1997.

- Georg Gottlob and Wolfgang Slany: Report on the ASHCOMP 96 school and workshop on approximate solution of hard combinatorial problems. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 61, February 1997.

- Wolfgang Slany: Scheduling as a fuzzy multiple criteria optimization problem. *Fuzzy Sets and Systems*, 78:197-222, March 1996.

- Juergen Dorn, Mario Girsch, Guenther Skele, and Wolfgang Slany: Comparison of iterative improvement techniques for schedule optimization. *European Journal of Operational Research*, 94, 1996.

- Erich Peter Klement and Wolfgang Slany: Fuzzy logic in artificial intelligence. In Allen Kent and James G. Williams, editors, *Encyclopedia of Computer Science and Technology*, Volume 34. Marcel Dekker, Inc. New York, 1996.

- Fritz Leisch, Kurt Hornik, Markus Bonner, and Wolfgang Slany: A proposal of hard- and software for desulphurization control with neural

networks and fuzzy logic. CD-Studie 96/21, Christian Doppler Laboratory for Expert Systems, Technical University of Vienna, 1996.

- Markus Bonner: A fuzzy tools comparison. CD-Studie 96/23, Christian Doppler Laboratory for Expert Systems, Technical University of Vienna, 1996.

- Markus Bonner: Using genetic algorithms to learn a fuzzy rule base for the desulphurization control at VA Stahl Linz. CD-Studie 96/27, Christian Doppler Laboratory for Expert Systems, Technical University of Vienna, 1996.

- Wolfgang Slany: *Fuzzy Scheduling.* PhD thesis, Technical University of Vienna, 1994.

- Wolfgang Slany: Fuzzy expert system for maintenance interval prediction. *Computers Chemical Engineering*, 18:S155-S159, 1994.

- Wolfgang Slany: Scheduling with fuzzy decision making methods (invited talk). In Mario Fedrizzi, Erich P. Klement, Aldo Ventre, and Alessandro Zorat, editors, *Proceedings of CIFT'94: Current Issues in Fuzzy Technologies: Decision Models and Systems*, pages 167-171, Trento, Italy, June 1994.

- Roger M. Kerr and Wolfgang Slany: Research issues and challenges in fuzzy scheduling. CD-Technical Report 94/68, Christian Doppler Laboratory for Expert Systems, Technical University of Vienna, 1994.

- Wolfgang Slany: Fuzzy expert system to predict maintenance intervals in a continuous caster. In Duk-Hyon Baik, editor, *Preprints of the international conference CPC-93, Computerized Production Control in Steel Plants*, pages 291-296, Seoul, Korea, November 1993. The Korean Institute of Metals and Materials, The Institute of Materials, UK.

- Wolfgang Slany: Fuzzy expert system for maintenance interval prediction. In Franz Moser, Hans Schnitzer, and Hansjoerg Bart, editors, *European Symposium on Computer Aided Process Engineering-3. ESCAPE-3, 25th European Symposium of the Working Party on Computer Aided Process Engineering, 494th Event of the European Federation of Chemical Engineering (EFChE)*, Supplement to Computers & Chemical Engineering, pages S155-S159, Graz, Austria, July 1993. Pergamon Press Ltd.

- Wolfgang Slany: Fuzzy constraint relaxation techniques for knowledge-based scheduling. In Hans-Juergen Zimmermann, editor, *EUFIT'93, First European Congress on Fuzzy and Intelligent Technologies*, pages 1124-1127, Aachen, Germany, September 1993. Augustinus Buchhandlung.

- Wolfgang Slany: Fuzzy constraint relaxation techniques for knowledge-based scheduling. In Roger Kerr, editor, *Workshop Notes of the Workshop on Fuzzy Scheduling Systems*, Linz, Austria, June 1993. University of Linz, Department of Mathematics.

- Juergen Dorn, Wolfgang Slany, Christian Stary, Wolfgang Steindl, Wolfgang Snopek, and Klaus Stohl: Aufgabenbasierte Spezifikation des Bildschirmarbeitsplatzes fuer Dispatcher im Stahlwerk LD3. CD-Studie 93/12, Christian Doppler Laboratory for Expert Systems, Technical University of Vienna, 1993. In German.

- Juergen Dorn, Wolfgang Slany, Wolfgang Snopek, Christian Stary, Wolfgang Steindl, and Klaus Stohl: Aufgabenanalyse der Dispatcher im Stahlwerk LD3. CD-Studie 93/11, Christian Doppler Laboratory for Expert Systems, Technical University of Vienna, 1993. In German.

- Juergen Dorn and Wolfgang Slany, editors: Scheduling von Produktionsprozessen - Von linearen Integermodellen zu symbolischen AI-Modellen. CD-Studie 93/9, Christian Doppler Laboratory for Expert Systems, Technical University of Vienna, July 1993. In German.

- Wolfgang Slany, editor: Knowledge based scheduling. CD-Studie 92/6, Christian Doppler Laboratory for Expert Systems, Technical University of Vienna, January 1992.

- Wolfgang Slany, Christian Stary, and Juergen Dorn: Vague data management in production process scheduling applied to high-grade steelmaking. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 214-221, University of Maryland, June 1992. Morgan Kaufmann Publishers, Inc.

B.2 Publications particular to FLIP++

- Markus Bonner, Stefan Mayer, Andreas Raggl, and Wolfgang Slany: FLIP++: A fuzzy logic inference processor library. In Trevor P. Martin and Anca L. Ralescu, editors, *Fuzzy Logic in Artificial Intelligence:*

Towards Intelligent Systems, volume 1188 of Lecture Notes in Artificial Intelligence, pages 44-56. Springer, 1997.

- Markus Bonner: Learning fuzzy rules with genetic algorithms and neural nets. Diplomarbeit (master's thesis), Technical University of Vienna, 1996.

- Markus Bonner: Description of fuzzy logic and FLIP++: a fuzzy logic inference library intended for the desulphurization control at VA Stahl Linz. CD-Studie 96/26, Christian Doppler Laboratory for Expert Systems, Technical University of Vienna, 1996.

- Markus Bonner, Stefan Mayer, Andreas Raggl, and Wolfgang Slany: Modeling constraints with the fuzzy logic inference processor FLIP++. In Hans-Juergen Zimmermann, editor, *EUFIT'96, Fourth European Congress on Intelligent Technologies and Soft Computing*, volume 2, pages 1312-1316, Aachen, Germany, September 1996. Verlag Mainz.

- Markus Bonner and Wolfgang Slany: Man machine cooperation for learning a complex control task with FLIP++. In Rudolf Felix, editor, *European Workshop on Fuzzy Decision Analyis for Management, Planning and Optimization (EFDAN'96)*, pages 115-120, Dortmund, Germany, May 1996.

- Markus Bonner, Stefan Mayer, Andreas Raggl, and Wolfgang Slany: FLIP++: A fuzzy logic inference processor library. In *Workshop Notes of the IJCAI'95 Workshop on Fuzzy Logic in Artificial Intelligence*, pages 83-92, Montreal, Canada, August 1995.

B.3 Publications particular to InterFLIP++

- Stefan Mayer: InterFLIP++: A user interface for StarFLIP++. Diplomarbeit (master's thesis), Technical University of Vienna, 1997.

- Stefan Mayer: InterFLIP++, a fuzzy logic inference library to interpolate a control variable for desulphurization control at VA Stahl Linz. CD-Studie 97/29, Christian Doppler Laboratory for Expert Systems, Technical University of Vienna, 1997.

B.4 Publications particular to ConFLIP++ and DynaFLIP++

- Ulrich Santa: DynaFLIP++: Dynamically adapted constraints for fuzzy multiple criteria optimization. Diplomarbeit (master's thesis), Technical University of Vienna, 1996.

- Wolfgang Slany: DynaFLIP++: a knowledge interpreter that matches dynamic domain knowledge with static fuzzy constraints. In *Proceedings of the 4th International Conference on Soft Computing-IIZUKA'96*, volume 2, pages 955-958, Iizuka, Japan, October 1996.

- Ulrich Santa, Wolfgang Slany, and Mazen Younes: *FLIP++: a partial constraint satisfaction system for steelmaking scheduling. In Marc Wallace, editor, *Proceedings of the Second International Conference on the Practical Applications of Constraint Technology (PACT'96): Scheduling, Resource Allocation, Planning, Configuration*, pages 211-220, London, UK, April 1996.

- Wolfgang Slany: Dynamic constraints in the *FLIP++ optimization libraries. In *Proceedings of IFSA'95*, volume 2, pages 13-16, Sao Paulo, Brazil, July 1995.

B.5 Publications particular to DomFLIP++

- Andreas Raggl: Implementation of a shift scheduling application using the *FLIP++ environment. Diplomarbeit (master's thesis), Technical University of Vienna, 1997.

- Witold Hendrysiak, Andreas Raggl, and Wolfgang Slany: DomFLIP++. In Da Ruan, Pierre D'hondt, Paul Govaerts, and Etienne E. Kerre, editors, *Intelligent Systems and Soft Computing for Nuclear Science and Industry, Proceedings of the 2nd International FLINS Workshop*, pages 123-129, Mol, Belgium, September 1996. World Scientific.

- Witold Hendrysiak: Ein objektorientierter Entwurf von wiederverwendbarer Scheduling Software. Diplomarbeit (master's thesis), Technical University of Vienna, 1995. In German.

- Ali Gharakani and Wolfgang Slany: Hospital scheduling from a fuzzy logic programming perspective - a case study. In *Workshop Notes of* *the IJCAI'95 Workshop on Fuzzy Logic in Artificial Intelligence*, pages 66-76, Montreal, Canada, August 1995.

- Aliakbar Gharakani: Hospital scheduling from a fuzzy logic programming perspective. Diplomarbeit (master's thesis), Technical University of Vienna, 1995.

- Thomas Laenger: Fuzzy and possibilistic approaches towards scheduling applications. Diplomarbeit (master's thesis), Technical University of Vienna, 1994.

B.6 Publications particular to OptiFLIP++

- Guenther Skele: Repair Based Scheduling mit Tabu Search. Diplomarbeit (master's thesis), Technical University of Vienna, 1994. In German.

- Mario Girsch: Optimierung von Schedules mit Genetischen Algorithmen und Iterativer Vertiefung. Diplomarbeit (master's thesis), Technical University of Vienna, 1994. In German.

- Juergen Dorn and Wolfgang Slany: A flow shop with compatibility constraints in a steelmaking plant. In Mark Fox and Monte Zweben, editors, *Intelligent Scheduling*, Chapter 22, pages 629-654. Morgan Kaufmann, 1994.

- Juergen Dorn, Mario Girsch, and Wolfgang Slany: Reparatur von Plaenen durch fallbasiertes Schliessen. In A. Bockmayr and F. J. Radermacher, editors, *Forschungsbericht des Max-Planck-Instituts fuer Informatik zum Workshop Kuenstliche Intelligenz und Operations Research*, September 1993. In German.

- Juergen Dorn, Wolfgang Slany, and Christian Stary: Uncertainty management by relaxation of conflicting constraints in production scheduling. In Marc Drummond, Mark Fox, Austin Tate, and Monte Zweben, editors, *Practical Approaches to Scheduling and Planning* , Working Notes from the 1992 AAAI Spring Symposium Series, republished as NASA Technical Report FIA-92-17, pages 62-66, Stanford, CA, May 1992. NASA Ames Research Center, Artificial Intelligence Branch.

- Wolfgang Slany, Christian Stary, and Juergen Dorn: Uncertainty management in production scheduling applied to high-grade steelmaking. In *6. Workshop Planen und Konfigurieren* FORWISS-Report, pages 51-60, Erlangen Muenchen Passau, March 1992. Bayrisches Forschungszentrum fuer Wissensbasierte Systeme.

B.7 Publications particular to CheckFLIP++

- Jan Vascak and Wolfgang Slany: The automatic check of knowledge base consistency. In *Proc. of the Conference on Intelligent Technologies - The Symposium on New Trends in Control of Large Scale Systems*, pages 113-122, Herlany, Slovakia, November 1996.

- Wolfgang Slany: A knowledge-base revision tool for the fuzzy constraints-based *FLIP++ scheduling library. In *Proceedings of the 5th International Conference on Fuzzy Systems-FUZZ-IEEE'96*, volume 1, pages 306-312, New Orleans, September 1996.

- Wolfgang Slany and Jan Vascak: A consistency checker for a fuzzy diagnosis system applied to warm rolling-mills in steelmaking plants. In *Proceedings of the 5th International Conference on Fuzzy Systems-FUZZ-IEEE'96*, volume 1, pages 206-212, New Orleans, September 1996.

- Wolfgang Slany: CheckFLIP++: A knowledge acquisition tool for the fuzzy constraints-based *FLIP++ scheduling library. In *Proceedings of the 6th Int. Conf. on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU'96)*, volume 2, pages 611-616, Granada, Spain, July 1996.

- Wolfgang Slany: Comparing partial constraint satisfaction models[1]. In *Workshop Notes of the CP'95 Workshop on Over-Constrained Systems*, pages 151-159, Cassis, France, September 1995.

[1]See URL ftp://ftp.cs.city.ac.uk/ocs-workshop.tar.gz