# Vi-XFST

# A VISUAL INTERFACE FOR XEROX FINITE-STATE TOOLKIT

by

YASİN YILMAZ

Submitted to the Graduate School of Engineering and Natural Sciences

in partial fulfillment of the requirements for the degree of

Master of Science

Sabancı University

Spring 2003

Vi-XFST; A VISUAL INTERFACE FOR XEROX FINITE STATE TOOLKIT

APPROVED BY:

Kemal Oflazer ..... <br>
(Thesis Supervisor)

Berrin Yanıkoğlu .....

Uğur Sezerman .....

DATE OF APPROVAL:

Sevgili eşime ve oğlum Yusuf Bilge’ye...

**Acknowledgments**

I like to express special thanks to my supervisor Prof. Kemal Oflazer, who has supported me in several ways in this project. His motivation and encouragement have always guided me through out my whole academic career.

**Abstract**

Vi-XFST; A VISUAL INTERFACE FOR XEROX FINITE-STATE TOOLKIT
Yasin Yılmaz
MS in Computer Science
Supervisor: Prof. Kemal Oflazer
August,2003

This thesis presents a management model and integrated development environment software for finite-state network projects using Xerox Finite-State Toolkit (XFST). XFST is a popular command line tool to construct finite-states networks, used in natural language processing research. However, XFST lacks various sophisticated management features to help the development phase of large projects where there are hundreds of finite-state definitions.

In this thesis, we introduce a new approach to XFST finite-state development: The source files are handled in a visual workspace associated with a project, and the project is developed step by step interactively by the user just like contemporary software development projects. Vi-XFST, the software we have created for our development model, includes automatic dependency tracking, source file management, visual regular expression construction, definition management and network testing features.

With Vi-XFST, a textual file editing is replaced with a project-building concept similar to modern software development tools. The benefits of adopting an integrated development environment designed for finite-state development include productivity gains by substantial reduced time for debug and management. The visual features of Vi-XFST enable viewing complex networks at different levels of detail and make even large projects manageable and comprehensible.

**Keywords:** Natural Language Processing, Finite-State Toolkit, XFST

**Özet**

Vi-XFST; XEROX SONLU DURUM MAKİNA DERLEYİCİSİ İÇİN GÖRSEL ARAYÜZ

Yasin Yılmaz

Bilgisayar Bilimleri Yüksek Lisans Programı

Tez Danışmanı: Prof. Kemal Oflazer

Temmuz, 2003

Bu tez çalışması, Xerox Sonlu Durum Makina Derleyicisi (Xerox Finite-State Toolkit-XFST) programının kullanıldığı sonlu durum projeleri için bir yönetim modeli ve entegre geliştirme ortamı ortaya koymaktadır. XFST, doğal dil işleme araştırmalarında kullanılan sonlu durum tanıyıcı ve dönüştürücülerinin hazırlandığı popüler bir komut satırı programıdır. Ancak, XFST yüzlerce sonlu durum tanımlarının bulunabildiği bu büyük projelerde ihtiyaç duyulan yetenekli yardımcı yönetim özelliklerinden yoksundur.

Bu tezde, XFST sonlu durum ağlarının geliştirme aşamaları için yeni bir yaklaşım sunulmaktadır: Kaynak kodlar, bir proje oturumu içerisinde, görsel bir çalışma ortamında ele alınmakta ve proje etkileşimli olarak adım adım geliştirilmektedir. Geliştirmiş olduğumuz yazılım, Vi-XFST, otomatik düzgün deyimlerin bağımlılık takibi, proje kaynak kod yönetimi, görsel düzgün deyimlerin tanımlama araçları ve sonlu durum ağı test özellikleri sağlamaktadır.

Vi-XFST sayesinde, daha önce bir metin dosyası ile hazırlanan proje geliştirme adımları, modern yazılım geliştirme yöntemlerine benzer bir yaklaşım ile değiştirilmiştir. Vi-XFST'nin görsel özellikleri, kompleks sonlu durum ağlarının değişik detaylarda incelenebilmesine olanak sağlayarak büyük projeleri yönetilebilir ve anlaşılabilir kılmaktadır. Özellikle sonlu durum projeleri için tasarlanmış bu entegre geliştirme ortamı, hata ayıklama ve proje geliştirmede önemli avantajlar sağlamaktadır.

**Anahtar Kelimeler:** Doğal Dil İşleme, Sonlu Durum Makina Derleyicisi, XFST

# Contents

# List of Figures

# List of Tables

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

Current finite-state development toolkits provide sophisticated compilers for finite-state systems but they lack software engineering and visualization tools to aid in the development of large-scale networks. A finite-state project contains hundreds of regular expression definitions. These structures have to be constructed and handled manually by the developer. Most of the time, finite-state projects are edited in a text file and then processed with the compiler. Corrections, debugging and other maintenance operations have to be done afterwards on the same text file and the whole project has to be recompiled again and again during the development cycle. Obviously this development life cycle is painstaking. Developing large-scale finite-state systems for natural language processing requires many software facilities beside a powerful compiler.

Xerox Finite-State Tool (XFST) is one of the most popular tools in natural language applications. Researchers use this tool to build transducers for many purposes e.g.; for in spelling and grammar checking, morphological analysis and finite-state parsing [4]. Beside natural language applications, finite-state networks are also being used in DNA sequencing, intrusion detection systems and virus or content checking in computer security applications.

XFST fulfills the needs of finite-state calculus with its comprehensive command set and capabilities. It is a command line tool where the inputs are typed in by the user. In XFST, a finite-state network is built by defining new networks and combining them step by step. At the end, the top network is the actual finite-state machine. This hierarchy can be visualized as a tree of dependent objects. Unfortunately XFST does not have satisfactory tools to manage this hierarchical tree of networks. When the finite-state projects get larger, managing them with XFST command line, becomes too complex to do manually. For example, changing one network at a level in this tree requires a sequence of recompilations of the networks that depend on the modified ones. This dependency control must be done manually. In a large research project, this is every difficult task and subject to human errors. Therefore, usually the whole network list is recompiled, which is a time consuming method compared to the selective recompilation.

We have designed a a development environment designed for XFST finite-state project development. Vi-XFST[1] includes visual regular expression development components, definition and network management tools with a large set of supported XFST commands. Vi-XFST also provides project maintenance tasks automatically such as definition and network dependency recompilations. Vi-XFST wraps the functionalities of XFST and provides an extensible architecture with graphical editing, management and testing features for finite-state projects.

## 1.2 Layout of the Thesis

This thesis is structured as follows: Chapter 2 presents design considerations for an integrated development environment for finite-state projects and a sample project development using our model; Chapter 3 focuses on implementation issues for the software, and Chapter 4 is the conclusion and discussions on the thesis. The following two sections are appendixes; first one is a user guide for Vi-XFST. The second appendix is a short list of various statistics about the source code.

[1] Visual Interface for Xerox Finite State Toolkit

# Chapter 2

# Design Considerations for a Finite-State Integrated Development Environment

## 2.1 Introduction

### 2.1.1 Finite-State Networks and XFST

Finite-state automata play an important role in natural language processing. They are used in spelling and grammar checking, morphological analysis and finite-state parsing [4]. Beside natural language applications, finite-state networks are being used in DNA sequencing, intrusion detection systems and virus or content checking in computer security applications.

XFST is a general-purpose utility for computing with finite-state networks. It enables the user to create simple automata and transducers from text and binary files, regular expressions and other networks by a variety of operations. The user can display, examine and modify the structure and the content of the networks. The result can be saved as text or binary files [1].

XFST is used to build networks from user defined regular expressions, which can be read from standard input or from a file. Networks can be combined with predefined operators to build new ones. The actual network can be saved to a binary file, which can be loaded and used without any compilation later. The user can apply strings to a top network to check if it is accepted. If the network is a transducer, the input may be transformed into another string.

XFST can read project files from a text file or the project can be typed at the XFST prompt. After XFST is loaded by the command shell, it prompts `xfst[0]:` and waits for user commands:

```
Copyright © Xerox Corporation 1997-2003 Xerox Finite-State
Tool, version 8.0.9
Type "help" to list all commands available or "help help" for
further help.
xfst[0]:
```

Figure 2.1: The XFST command prompt.

Commands are typed in after the prompt (`xfst [0]:`) as shown in Figure 2.1. When the return key is pressed, command execution starts. After the execution finishes, result messages are displayed and the prompt appears again for more commands. Error messages are also displayed in this same text screen.

## 2.2 Modular Structure of Finite-State Machines

Understanding the structure of finite-state networks and how they are built in XFST is the first step to figuring out the design requirements of their development environment.

XFST enables the user to develop finite-state transducers by defining regular expressions. Each regular expression can be used in other expressions with finite-state operators to form more complex definitions. Therefore, a finite-state development environment should facilitate this expression reuse and modular structure of expressions. This object hierarchy also intrudes two more concepts to the design considerations. First one is visualisation of the structure and sub-components of a complex regular expression. Second concept is tracking of expression dependencies based on this modular structure.

## 2.3 Viewing a Regular Expression

A transducer can be visualized as a black box that can take inputs on one side and produce outputs on the other side. For example, the following transducer maps input strings on the upper side, to the strings on the lower side, marking substrings that match a date format with parentheses. This simple date parser is implemented using following definitions with XFST [4]:

```
define 1to9 [ 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ];
define Day [
Monday|Tuesday|Wednesday|Thursday|Friday|Saturday|Sunday ];
define Month [
  January|February|March|April|May|June|
  July|August|September|October|November|December ];
define def2 [ 1|2 ];
define def4 [ 3 ];
define def5 [ ["0" | 1] ];
define EMPTY [ 0 ];
define def16 [ (Day ", ") ];
define SPACE [ " " ];
define 0To9 [ "0" | 1to9 ];
define Date [ 1to9 | [ def2 0To9 ] | [ def4 def5 ] ];
define Year [ 1to9 [ [ 0To9 [ [ [ 0To9 [ 0To9 | EMPTY ] ] |
EMPTY ] | EMPTY ] ] | EMPTY ] ];
define DateYear [ (", " Year) ];
define LeftPar [ "[" ];
define RightPar [ "]" ];
define Even [ "0" | 2 | 4 | 6 | 8 ];
define Odd [ 1 | 3 | 5 | 7 | 9 ];
define AllDates [ Day | [ def16 Month SPACE Date DateYear ] ];
define AllDatesParser [ AllDates @-> LeftPar ...  RightPar ];
read regex AllDatesParser;
```

Figure 2.2: The AllDatesParser transducer is defined using 19 regular expression definitions.

The top regular expression is the transducer that maps input strings to outputs. This transducer can be viewed as a box at the top-level:

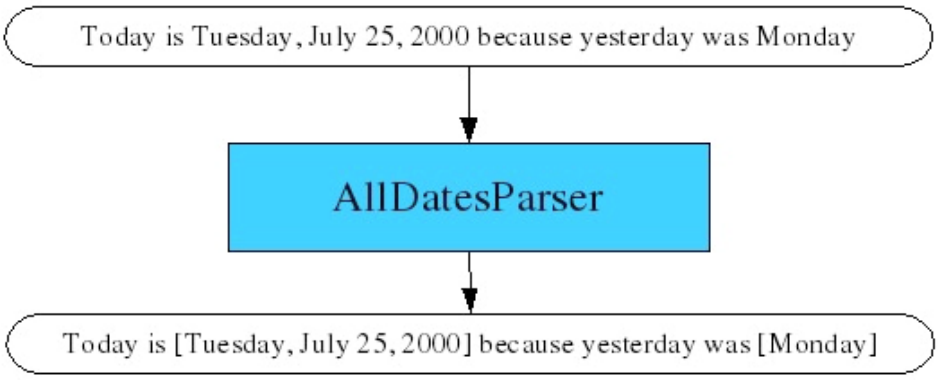


Figure 2.3: A transducer can be viewed as a closed box that maps inputs to some outputs.

The direction of this mapping can be reversed, which means that the input can be applied from the bottom of the box, where the output will be produced from the top of this virtual transducer. The top view of the parser gives us a little clue about the structure of the networks

it is composed of. In fact, in development phase, a transducer is built using previous smaller networks, each doing a sub-section of the task. So we would like to be able to view the internal structure of a regular expression.

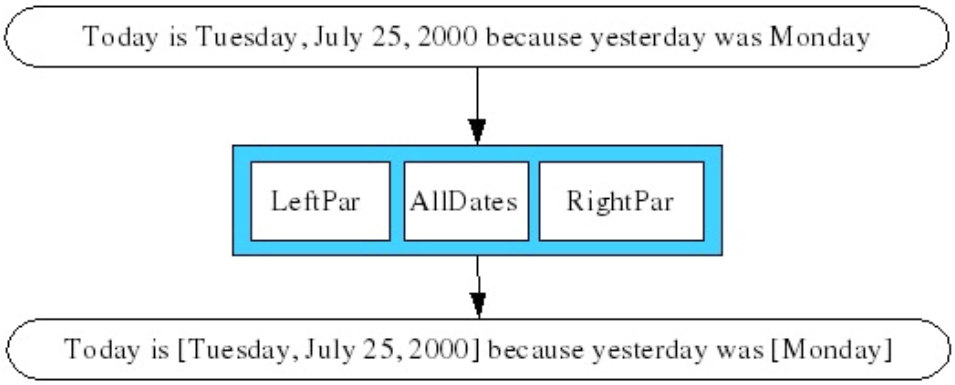


Figure 2.4: The first level of detail for the date parser.

In Figure 2.4, we can visualise a transucer with some of its subcomponents. These networks can also be enlarged into thier subcomponents. A finite-state development environment should be able to let the user view a transducer in different levels of details. This enhances visualization of regular expressions and improves the comprehension of complex transducers.

### 2.3.1 Dependency Tree of Networks

As described above, a network is constructed upon smaller ones. This dependency hierarchy at development time, can be viewed as an acyclic dependency graph as in Figure 2.5:

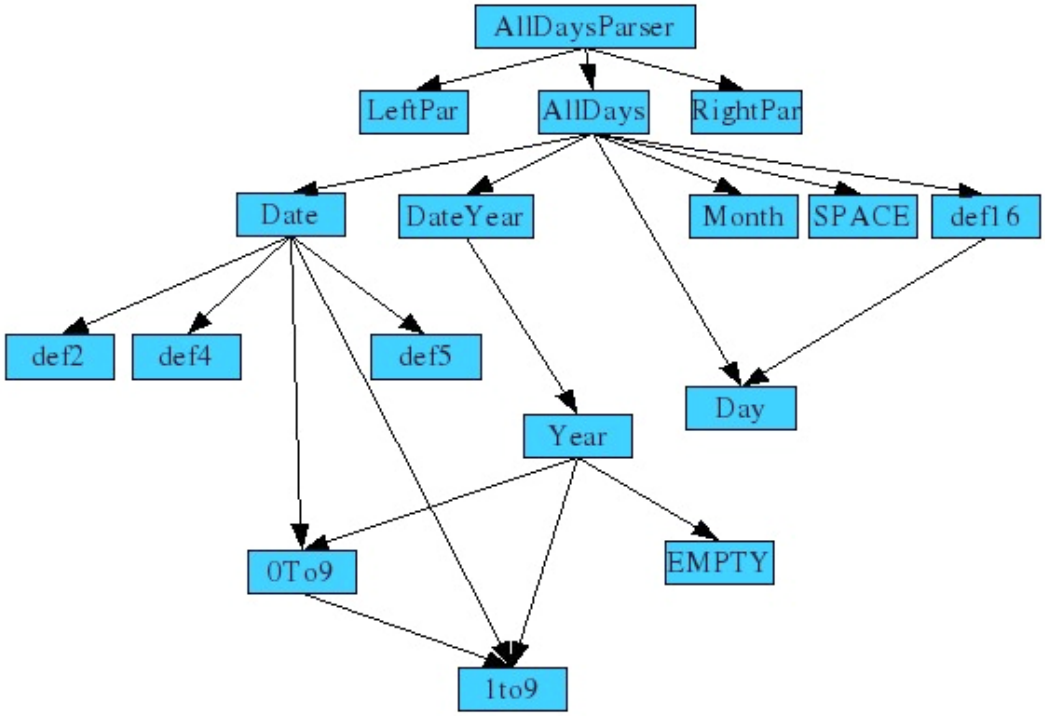


Figure 2.5: A dependency graph of a network.

A requirement of XFST is that, when there is a modification in a definition of regular expression, a minor correction for example, the whole dependency tree from the modified network up to the root network, must be recompiled by the user. It is obvious that even within such a tiny sample project, it is quite hard to predict the path from a node to the top root. With a quick heuristic, it is evident that compiling each node visited from the ancestors of the modified definition to top recursively is a solution. But it this leads to recompilations of same nodes more than once, which is quite time consuming, and not the optimal solution.

Suppose that the researcher updates definition of the network "`1to9`". One has to figure out which definitions have to be recompiled. Further, the order of this recompilation is very important. The ordering can be achieved by topological ordering of the sub-graph that spans the dependency relationship of the modified node. For example, the sub-graph of node `1to9` is shown in Figure 2.6:

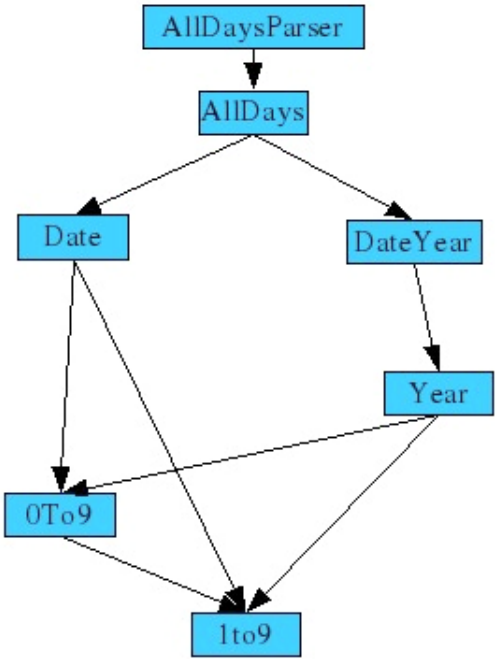


Figure 2.6: Dependency sub-graph of node `1to9`

Topological sort of the this dependency tree gives us the correct ordering of definitions that have to be recompiled:

```
1to9, 0To9, Date, Year, DateYear, AllDays, AllDaysParser
```

## 2.4 The Requirements of A Finite-State Project

As described in Section 2.2, regular expressions for finite-state transducers possess their own individual structure that must be taken into account while designing a development environment for them. XFST is a great tool to construct, test, and update networks, with a very comprehensive set of commands. Our intention is to fulfill the management needs of this powerful command line toolkit.

Based on the usage patterns of developers using XFST, and the properties of regular expressions, our project model has evolved with the following aims.

### 2.4.1 Access to Visual Model of The Expressions

As hinted earlier, visual development of complex regular expressions is a very desirable feature. Without a visual model, it is still possible to type in expressions in an edit box, but that does not contribute to the intention of an integrated development environment, which is to ease the burden on the developer. So wherever possible, the visual structure of an expression should be accessible in a finite-state project management solution.

### 2.4.2 Controllable Details

A finite-state development environment that facilitates model-based structures should also have features to control the details displayed. Detail hiding is a must when things on the screen get too crowded. It should be possible to focus on a section of a regular expression, debug that part, fix it and then move to other components. While doing these steps, the user should be able to control the level of detail with the help of his development environment.

### 2.4.3 Network Control and Reuse

In a finite-state project, hundreds of networks may be created by regular expression definitions. They are used in many different parts of the project to build new ones. This key concept of regualar expression and reuse, should be facilitated with easy to use functions in an integrated development environment. The list of available networks, in the order they are pushed into the stack, must be accessible to the user in a visual environment.

### 2.4.4 Managing Definition Dependencies

The dependency problem is one of the major issues that has to be solved. The definition recompilations should be accomplished in the most accurate and optimized way without user intervention. As stated in Section 2.3.1, the solution to dependency recompilation ordering is extracted from the topological sort of the dependency relationship sub-graph of a definition.

Dependency control is not only applicable to definition modifications. In our development environment model, a definition which has dependents, is prevented from being undefined, or renaming. For example, the following three lines defines three networks where the network AB depends on both A and B:

```
define R red;
define B blue;
define COLORS [ R | B ];
```

Our solution does not allow the user to `undefine` or `substitute` (rename) definition `R` or `B` since they have dependents that refer to them using these names in their regular expressions. If it was allowed to rename or undefine them, a recompilation for definition `COLORS` will not create the intended network. Suppose definition `R` is undefined and definition `COLORS` is recompiled again:

```
define R red;
define B blue;
define COLORS [ R | B ];
```

```
undefine R;
define COLORS [ R | B ];
```

XFST will not generate any error messages. The redefined definition `COLORS` now does not accept input strings *{"red","blue"}* but *{"R","blue"}.* This was probably not what the user wanted. This kind of problems during development phase can be prevented by dependency controls in our management model.

### 2.4.5 Definition Name Controls

Another issue that has to be controlled by a finite-state development environment to keep the project "strongly typed", is the uniqueness of regular expressions names defined. For example, redefining a definition in XFST is valid as in the following piece of code:

```
define R light_red;
define B light_blue;
define LIGHT_COLORS [ R | B ]; //R is used in this expression
define R dark_red; //R is redefined
```

The finite-state definitions above are not a good development practice. A network is redefined although it has dependents and maybe used in other definitions. This piece of code is valid and does not produce any error or warning messages in XFST. But if the user recompiles definition LIGHT_COLORS during debugging his code, the LIGHT_COLORS will reject the intended input string set *{ "light_red","light_blue" }* and accept set *{"dark_red","light_blue"}*.

These kinds of errors will become a common issue when there are thousands of definition names to remember. Definition name conflicts are even more common when the development has more than one developers working on it. Therefore, to prevent such errors and ambiguities, our management model will not allow definition name overriding.

## 2.5 Features of Vi-XFST

Vi-XFST provides a simple and easy, yet powerful way to develop finite-state networks without involving developers in the complexities of a command line tool. With its set of innovative features, less experienced developers can quickly start testing with finite-state concepts seeing the actual picture on their workspace, while the advanced developers are freed from many manual tasks and controls that they had to cope with before. This means that they can focus on what to build, not on how to.

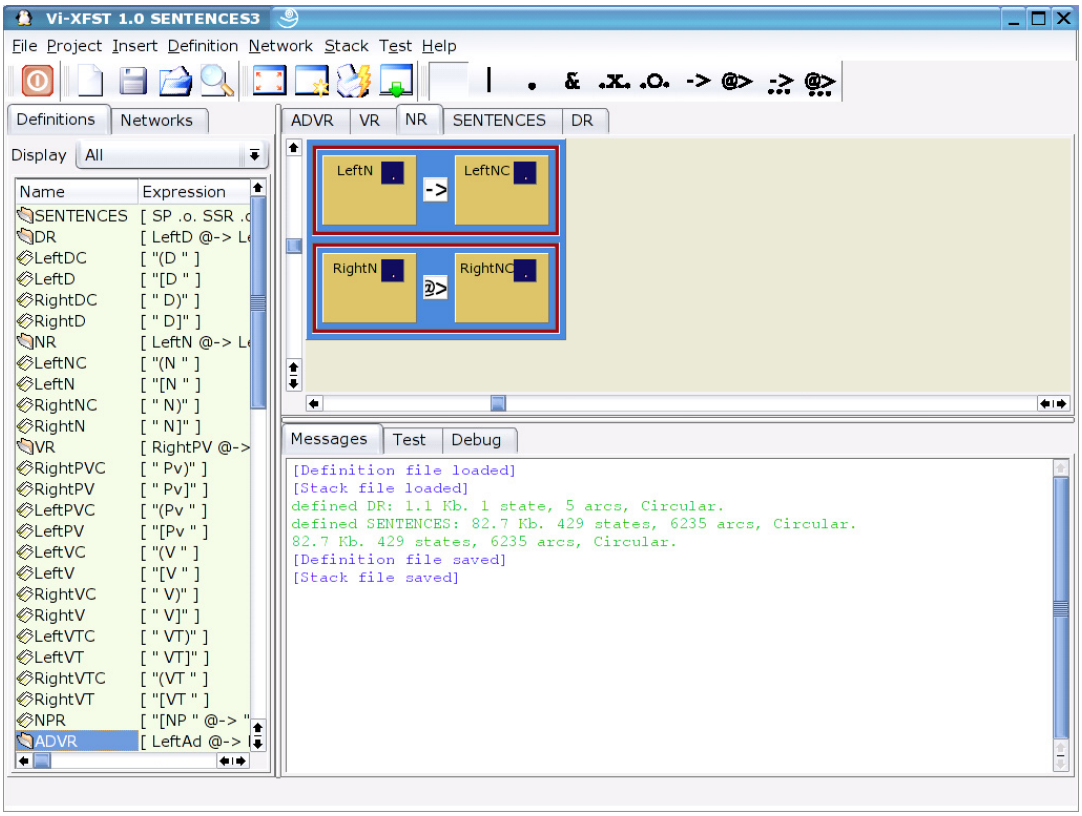


Figure 2.7: A sample screen-shot from the Vi-XFST IDE.

The following are important features of Vi-XFST:

- *A development environment designed for XFST:* Vi-XFST treats an XFST file as a development project that has to be managed on behalf of the user as he builds the regular expressions. The developer can move from the traditional way of editing an XFST source file, mostly done with a text editor like ”*vi*” or ”*emacs*”, to a real integrated development environment, like Vi-XFST. He can see results of his work at design time, modify the code and retest any component.

- *Visual Regular Expression Development:* Vi-XFST’s graphical regular expression construction tools allow developers to quickly build a visual model of their finite-state regular expressions. The developer quickly creates a topological model of the expression showing the relationship between expressions, as they are combined on the canvas of the visual editor. It is also easier to break down into the visual structure of large regular expressions. There is a hierarchical view of networks available, and that hierarchy is visible with in Vi-XFST. The user can zoom in a definition to see what is inside it, and go deeper, do tests at any level on a component, modify it, and go back to the top picture. This makes it possible to view networks at different levels of detail and make even large structures manageable and comprehensive.

- *Automatic definition and regular expression dependency checks and recompilation:* Vi-XFST watches modifications to a regular expression and recompiles any other definition that depends on it. Even the networks on the stack created with previous "*regex*" commands are recompiled if they depend on a modified regular expression. This is, in general, a difficult process for the developers to do manually. But with Vi-XFST, it is just transparent to the user at the background and automatically handled. Vi-XFST determines which definitions have to be recompiled. This selective recompilation of modified definitions is much efficient than recompiling the whole project.

- *A large set of supported XFST commands:* Beside expression operators, Vi-XFST supports many of the XFST's comprehensive command set and their options. They are hidden behind many easy to use dialog boxes, menu buttons and other graphical components of Vi-XFST. The developer will even use some of them without noticing, as he changes a project setting, clicks a button or updates an expression. Vi-XFST will send the appropriate commands to XFST on behalf of him to accomplish the requests.

- *Definition and Network Browsers:* These two browsers introduced in Vi-XFST display the list of defined regular expression definitions and available networks on the stack. The developer can access any one of the definitions or networks with just a mouse click. He can check or modify their properties, or use them in other parts of the project. For example, it is very easy to view which regular expression depends on a particular one. Without Vi-XFST, it is a quite difficult task. Also, properties of a network can be accessed with only a mouse click.

- *Drag and Drop:* Once a regular expression definition is defined, it is available in the Definition Browser. Then the user can drag and drop it with his mouse onto canvas to construct new expressions. Once basic definitions are defined, user can build new expressions without typing anything at all; create a definition base, drop previous definitions into it, and click *Push definition* and new definition is ready. Even a unique definition name is automatically created on behalf of the user. Vi-XFST provides a strongly typed development environment that reduces type errors while writing definition regular expressions.

- *History of input and output test strings:* Vi-XFST keeps track of strings applied to a network on the stack. User can always go back and test with his previous inputs with just one mouse click. He does not have to try to remember the inputs of last tests. They are saved inside the project file, and can be exported to any text file.

- *Message handling:* Vi-XFST handles every message from XFST program. They are never lost between user commands as before. Error messages, test outputs, normal XFST messages are all differentiated by Vi-XFST, parsed and indicated to the user.

- *XFST compatibility:* Vi-XFST project file can be used directly inside XFST as a script file. There is no Vi-XFST specific code inside the source file of the project that may be rejected by XFST.

- *Multi-platform IDE:* Vi-XFST runs on many Unix systems (Sun Solaris and all Linux distributions) and even on Microsoft Windows platforms with the same functionality. It is a fast pure C++ application, not a slow interpreted code like Java or TCL.

# Chapter 3

# An Example Project Development

In this chapter we present our management solutions for finite-state project development issues with examples. Only the key concepts of our model will be presented here. For more details of Vi-XFST features and usage, please refer to manual documentation of Vi-XFST in the Appendix sections.

The project that is built below is a regular expression that describes the operation of a vending machine that dispenses drinks for 65 cents a can. It accepts any sequence of coins: 5 cents (represented by input *'n'*), 10 cents (represented by input *'d'*) or 25 cents (represented by input *'q'*). If one puts in the right amount of money in any combination of these coins, a can of soft drink drops into a bin (represented by output *'PLONK'*); otherwise nothing happens. To focus on the features of the development environment, we will demonstrate a simpler version of vending machine that does not return any changes [3].

## 3.1 Starting a Project

Vi-XFST handles each development session with XFST in a *Project Workspace*. A finite-state project is no longer created by text file editing. In a workspace of Vi-XFST, there are various settings, controls and options associated with the project. For each project three files are used; one for the regular expression definitions and two more binary files for network and definitions on the stack. These files are the ones created by *"save defined <filename>"* and *"save network <filename>"* commands of XFST. These files are also a part of the project, and they are automatically synchronized (saved or loaded) with the project content transparent to the user.

To start a new project workspace, one clicks **Project|New**[1] menu item, or the associated button on the toolbar. The *Project Options* dialog will be invoked. A descriptive name for the project and a directory path for the workspace files are expected to be entered in this dialog.

[1]This syntax printed in bold, defines a command (*New*), under a menu item (*Project*) on the top of the main window.

Default value displayed for the directory points to the current directory, but it is probable that the developer will want his project files saved in a more reasonable location.

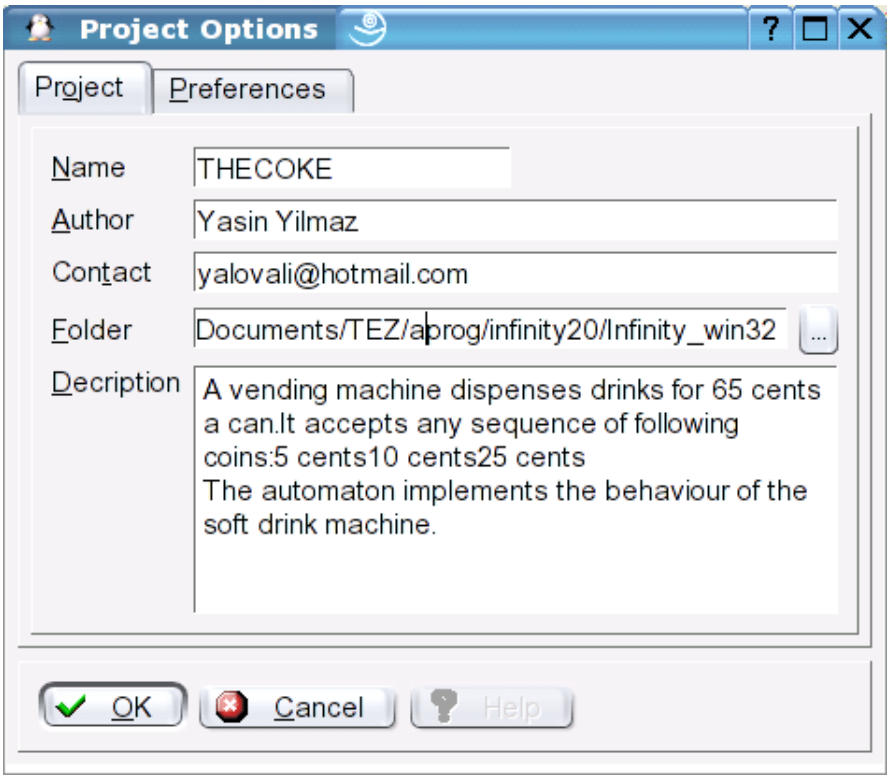


Figure 3.1: The Project Options dialog

When the OK button is clicked, the *Project Options* dialog will be closed and a new project workspace will be initiated. An XFST process will be loaded while menu items, browsers and workspace canvas are initialized. As this initialization procedure is carried out, the XFST Progress dialog will appear for a short period of time. This dialog indicates that Vi-XFST is busy with executing some commands and it will disappear automatically with the end of the active task. When the initialization finishes, one can start adding definitions to his workspace.

## 3.2 Building Expressions

To build our vending machine, we start by defining a mapping for the coins to their corresponding cents values. This can be accomplished by constructing a transducer that maps each of coins to a string of ”c”s of length that represents its value. For example, `c^5` denotes the language consisting of the string "`ccccc`". Thus the expression `[n .x. c^5]` expresses the fact that a nickel is worth 5 cents and defines a mapping that *transduces* `n` into "`ccccc`". A relation that will be of the following form will map one coin to the given cent value:

```
[ [ N .x.  c^5 ] | [ D .x.  c^5 ] | [ Q .x.  c^5 ] ];
```

The base operand of this expression is the Union operator ( `|` ). So, to build this expression on the canvas, we select the union operator icon from the tool bar and click on the empty expression canvas. An operator base with two open slots will be opened.

We will insert three crossproduct expressions (`[ N .x. c^5 ] , [ D .x. c^5 ] , [ Q .x. c^5 ]`) to map each coin type into their cent values. Before inserting these crossproduct operators as operants, we see that our union operator has only two empty slots, whereas we need three. So we just right click the union operator base and select "*New slot*" menu item. This will add an extra slot to the base. Now our union operator has three empty slots:

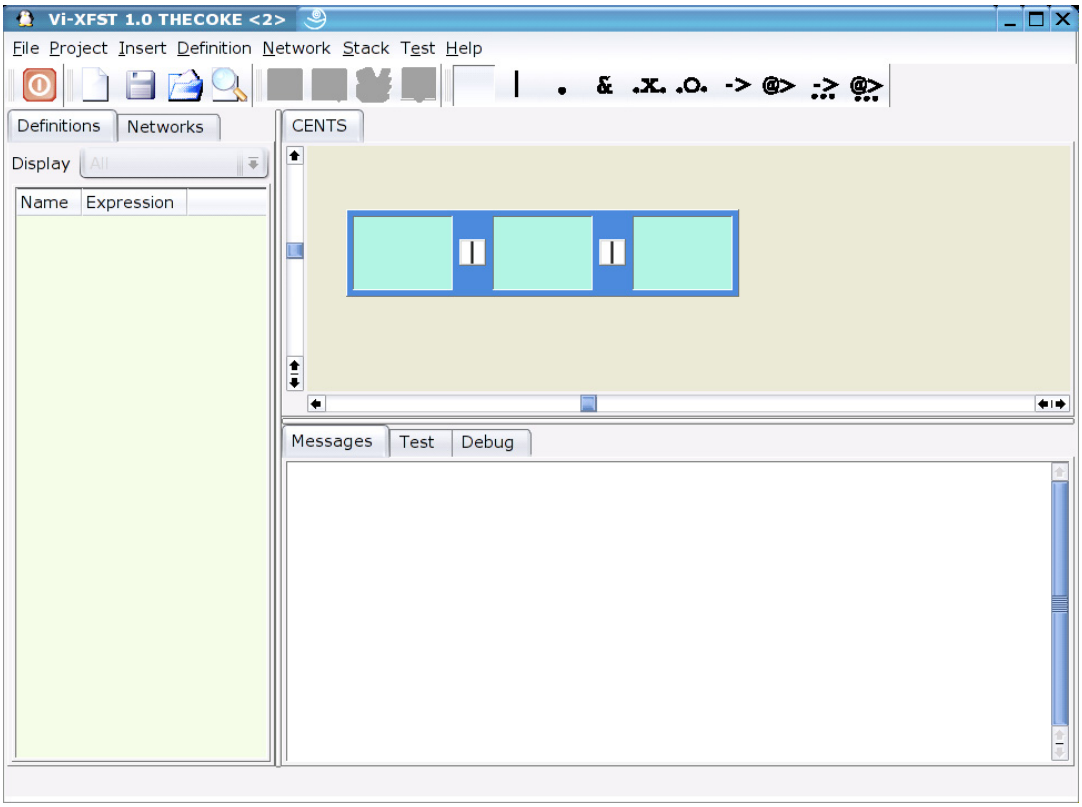


Figure 3.2: Union operator base with three empty slots.

This operator regular expression will be named as "`CENTS`". To change its name; we press `F9` to invoke the *Definition Options* dialog to edit properties of the active definition on the canvas, then change the auto-generated definition name to "`CENTS`". Also a comment seems reasonable here:

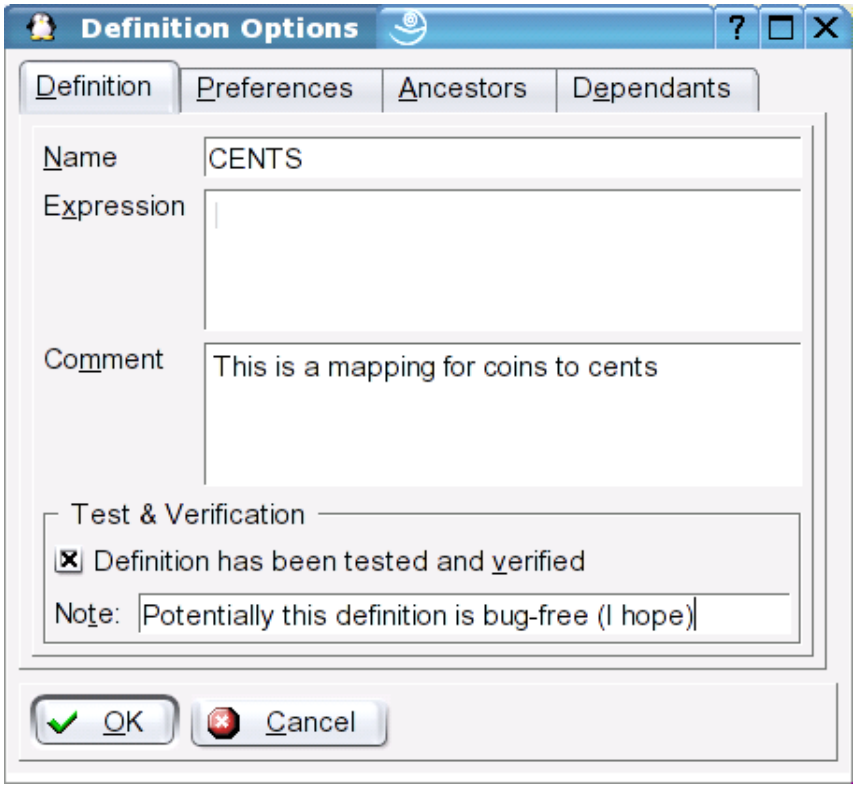


Figure 3.3: *Definition Options* dialog is used to access definition properties.

Then, we click OK to accept the changes.

Next, we select *Crossproduct* (`.x.`) operator from the tool bar and click inside one of the empty slots of the union base on the canvas. As one can see, it is possible to nest operators within each other to construct more complex regular expressions. Now we repeat adding crossproduct operators two more times for each empty slots of the union operator:

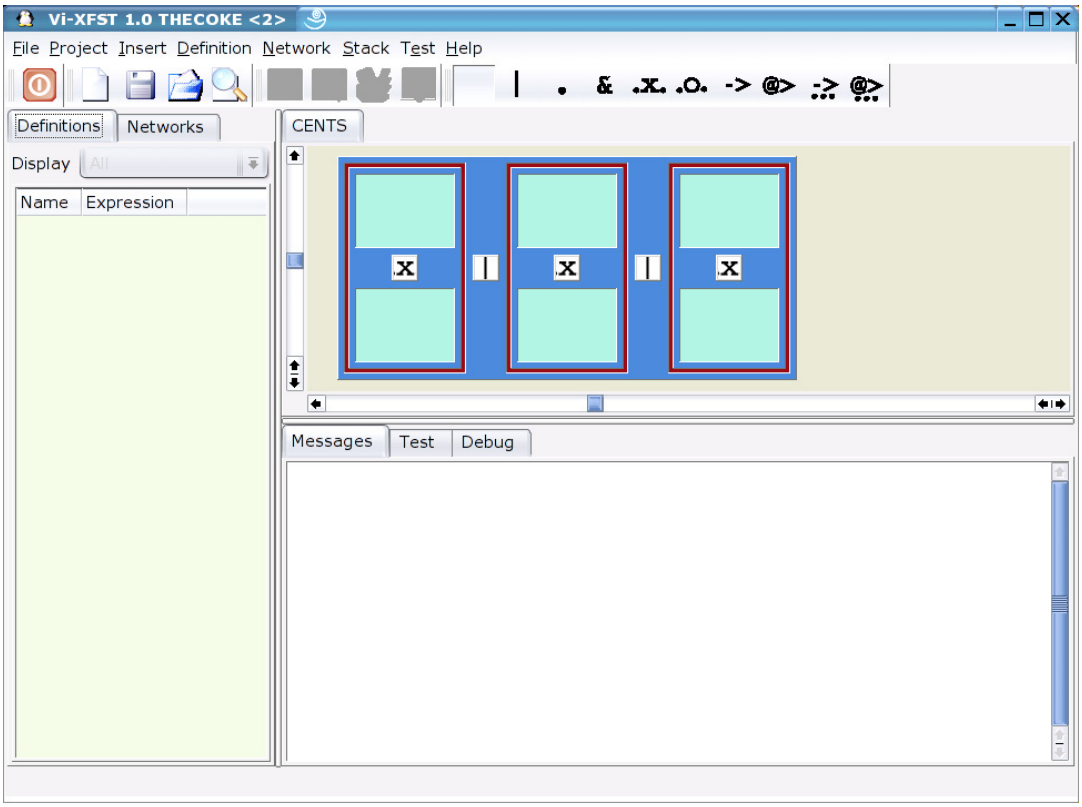


Figure 3.4: It is possible to nest operator bases inside each other.

So far no definition has been entered into XFST. We are still working on the skeleton of our regular expression. At any point, we can define our definitions and insert them in the slots of an operator base. Vi-XFST makes it much easier to focus on the ”design” of the models before the actual code is implemented, similar to object-oriented design principles. Vi-XFST’s model-driven approach to regular expression definitions allows developers to quickly build a visual model of their expression before they type in any definition.

Once we are satisfied with the operator base, we start adding actual definitions inside the slots. To start with, we double click in the upper slot of the first *Crossproduct* base. A new definition will be created and inserted into this slot. The *Definition Options* dialog is invoked, presenting a new definition for us. A definition name is already generated. We change the name to ”`N`” and regular expression to ”`n`”. This definition only denotes a single symbol. Similarly, it would have been defined in XFST command prompt as:

```
define N n;
```

We click OK. The XFST Progress dialog will appear and define the regular expression to XFST and name it as ”`N`”. If there is no error, the definition will appear in the *Definition Browser* and it is automatically inserted into the empty slot that we have double clicked in.

The *Message Tab* will be popped up if it is not visible. This tab contains a text box where any XFST or Vi-XFST messages are displayed. One should check these messages for his definitions. If there has been an error, Vi-XFST would have noticed that. But it is always wise to

check for any inconsistencies; such as unexpected definition sizes may be hint for debugging of the code in the future.

We double click the lower side of the first *Crossproduct* base, and add a new definition with name ”C5” and expression ”c^5”. The first operand of the Union operator base is finished.

Then we add other definitions to the two crossproduct bases, with definitions:

| Name | Expression |
|---|---|
| D | d |
| C10 | c^10 |
| Q | q |
| C25 | c^25 |

Table 3.1: List of definitions to add into the base.

The completed operator base looks like:

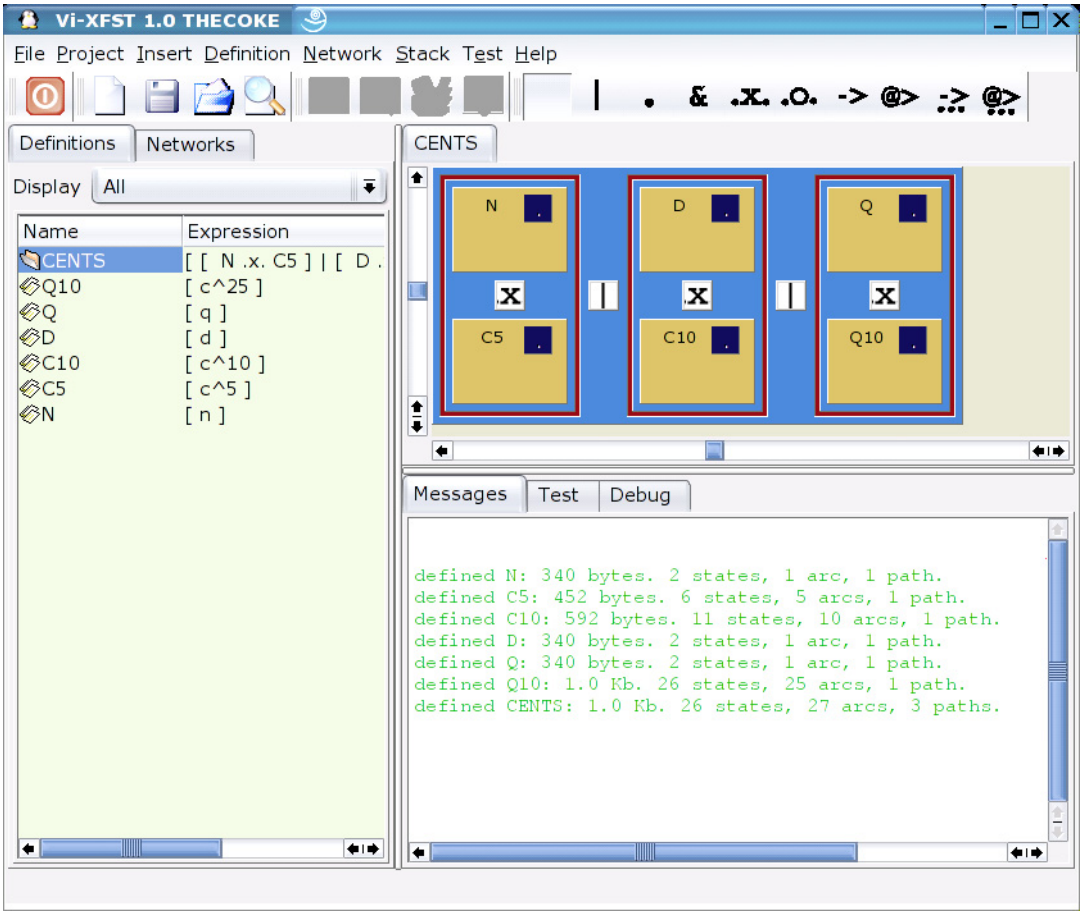


Figure 3.5: A mapping from coins to cent values: `[ [ N .x. c^5 ] | [ D .x. c^5 ] | [ Q .x. c^5 ] ]`

Next, we right click on the *Union* operator base and select *Push definition* option from the pull-down menu. The definition will be defined in XFST and added to the *Definition Browser*.

Now we want to define a Kleene star for the "`COINS`" definition. Not all operators of XFST are available in Vi-XFST expression canvas yet. Therefore we have to type this definition just like we have done for symbols above.

We select **Definition|New definition** menu, or just press F4 to open the *Definition Options* dialog. Then we change the default name to "`SixtyFiveCents`" and type "`COINS*`" for the expression and close the dialog with the OK button. Although it is not edited on the visual expression canvas, the same dependency tracking and recompilations are applicable to this expression, like other definitions in Vi-XFST.

Another mapping should be for 65 cents to the output can, "`PLONK`". The actual regular expression that we will construct on the canvas will be "`[ C65 .x. DefPlong ]`". We just select the Crossproduct icon from the toolbar and click anywhere on the *expression canvas*. A new workspace tab will be opened to place the operator base. We rename this base as "`PRICE`" as described above from the *Definition Options* dialog, then we insert two new definition into empty slots of this crossproduct base which are:

| Name | Expression |
|---|---|
| C65 | c^65 |
| Def_PLONK | PLONK |

Table 3.2: Definitions to be inserted into the slots of PRICE Crossproduct base.

After we enter the actual definition in to XFST:

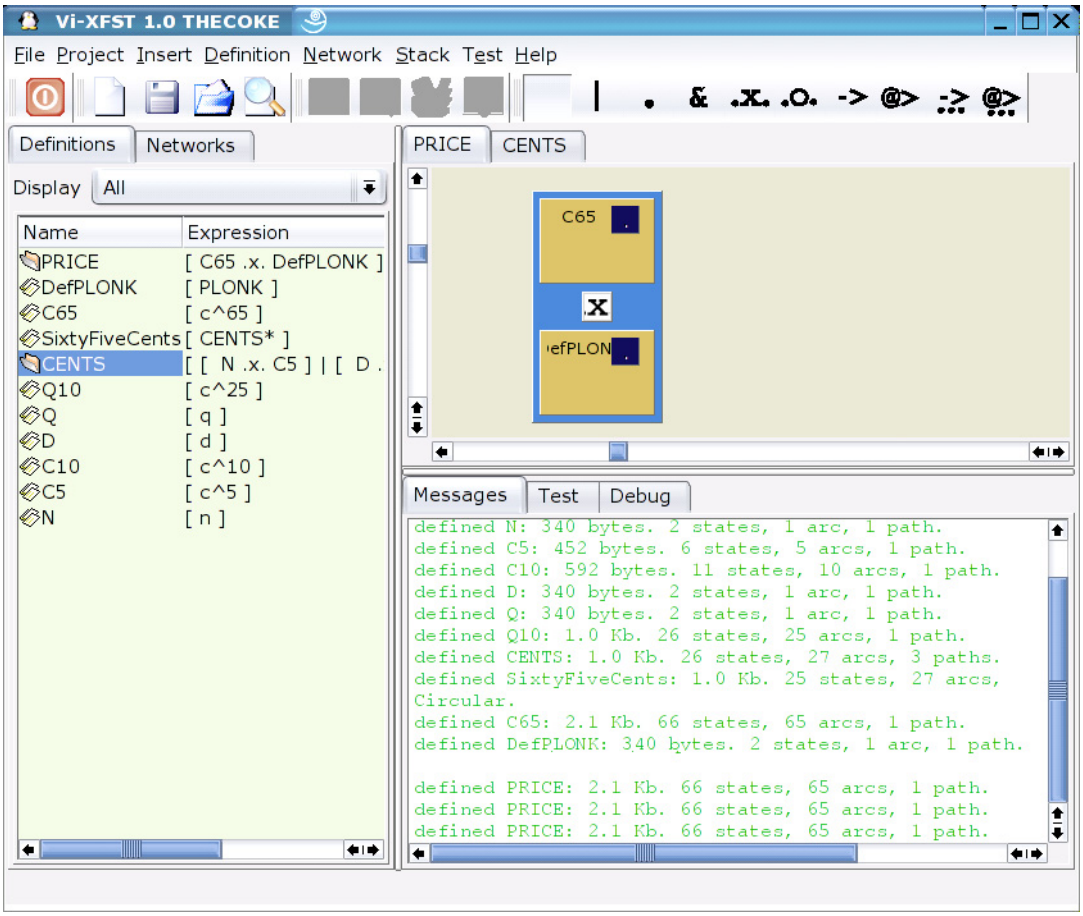


Figure 3.6: The `PRICE` definition on the canvas

Now the final step is a mapping from cents to the price of the drink can. This is simply "`[SixtyFiveCents .o. PRICE]`". We place a Composition operator (`.o.`) on the canvas and rename it to "`BuyCoke`". Next we select definition `SixtyFiveCents` from the *Definition Browser*, then drag and drop it into the upper slot of the composition base. In the same manner we add the `PRICE` definition also into the lower slot of the base:

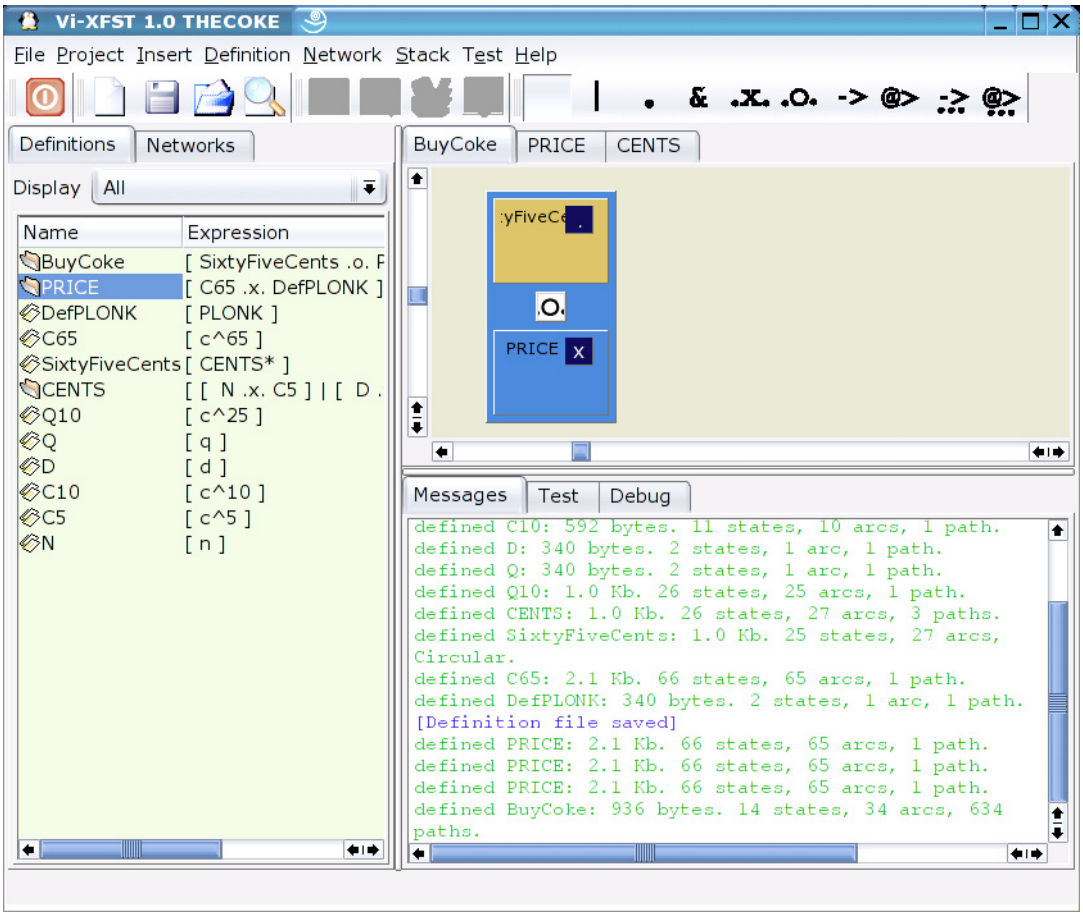


Figure 3.7: SixtyFiveCents .o. PRICE

Then we right click and send the definition to XFST. Now we have completed building our regular expressions and ready to push our network onto the XFST stack.

## 3.3 Compiling a Regular Expression

Just like most of the commonly used commands of Vi-XFST, there are various ways to compile a regular expression. A simple way is to right click the definition name on the *Definition Browser* and select ”*Read Regex*” menu item.

When the network is successfully created on the stack, it is displayed in the *Network Browser*. After each compilation, Vi-XFST will switch the workspace tabs to test phase automatically.

We can compile regular expressions at any time while building a project. One can just right click a base on the canvas and select ”*Read Regex*”. The compilation will start. Then the graphical user interface switches to test tabs, ready to debug the expression with inputs. When we are done, we can switch back to building our expressions again. We can always see which network is on the top of stack. The stack can be rotated, reversed, cleared or modified by pop-up command. Without Vi-XFST, users had to print content of the stack each time to understand and

remember the structure of the stack. Now it is all visible on the screen. By this way Vi-XFST encourages the test of each building block of the project at any time without any management penalty of the stack. It is handled automatically by Vi-XFST.

We locate the `BuyCoke` in the *Definition Browser*, right click and select "*Read Regex*". The definition will be compiled onto the stack and Vi-XFST will activate test tabs and place the cursor in the input string edit box.

## 3.4 Testing a Network

Inputs to a network on the stack can be entered using the *Input String* edit box, and pressing enter or clicking the *Apply* button just below the edit box. The direction of apply command can be set by *down* and *up* radio buttons near the *Apply* button. The results will be displayed in the *Results* edit box, and the input string will be added to the *Inputs* list.

For example, we enter two quarters and one dime and one nickel (`qqdn`) and press enter. The output is a can of drink: `PLONK`!

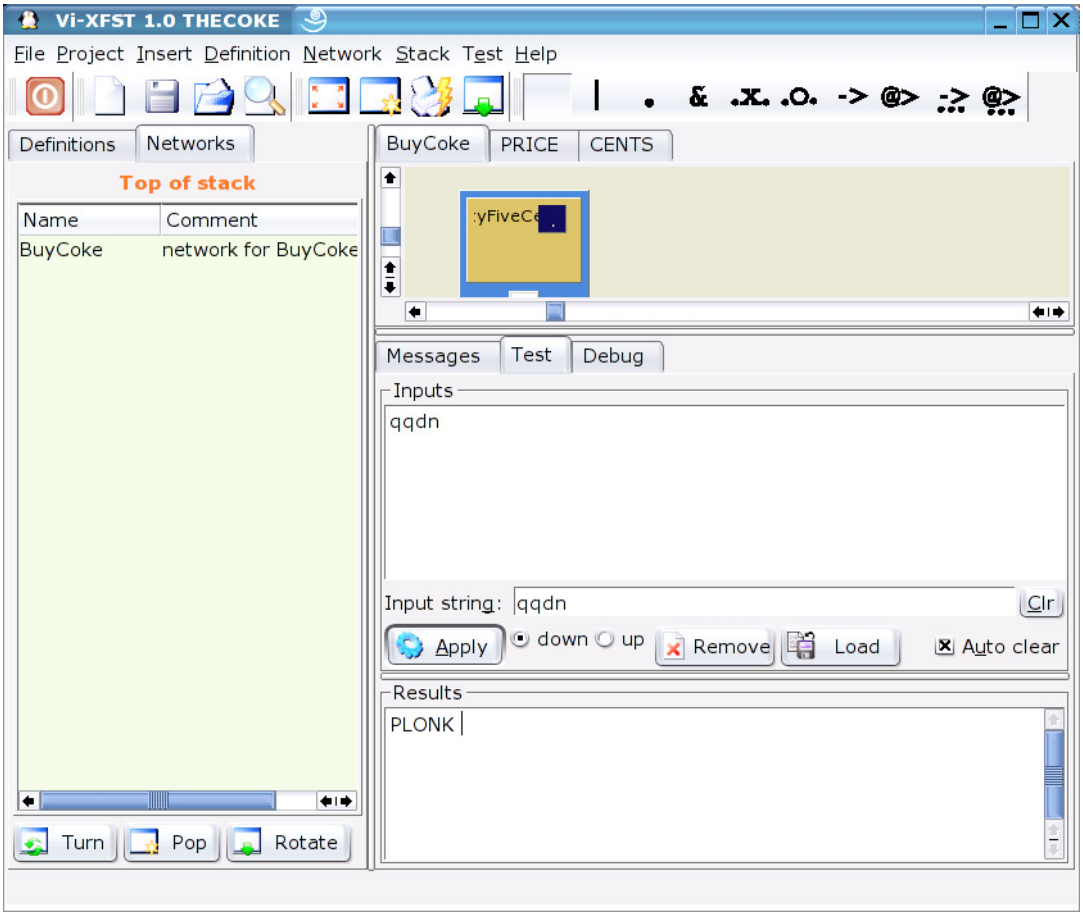


Figure 3.8: Testing a network.

A set of testing features are also available to help the testing phase of the project. Vi-XFST keeps track of user test inputs and outputs, so that they can be referred back, as the debugging goes on. Items in the *Inputs lists* can be removed, cleared, loaded from, or saved to a text file. These operations are available both through the buttons on the *test tab* and menu items under the **Test** menu. If auto-save option is set in the Vi-XFST settings, input strings are kept inside the source file when the active project is saved. They are also loaded when the project is reopened.

## 3.5 Modifying the Networks

Once we have tested the network on the stack, we may want to make some modifications. For example the prices of the coke may be updated to 100 cents. If this value is modified, then the networks that are affected by this change should also be recompiled. The dependency tree of whole coke machine is show in Figure 3.9.

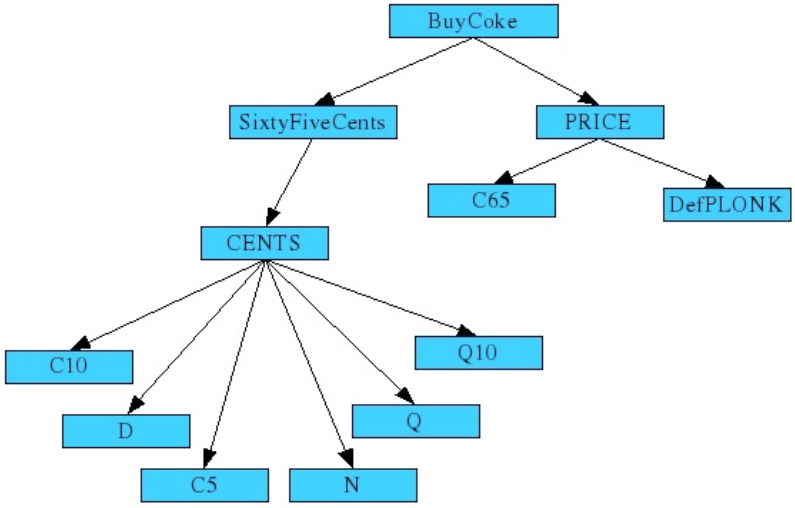


Figure 3.9: The dependency graph of the project

A dependency tree is automatically generated for every definition when it is viewed in *Definition Options* dialog. We select the top definition "`BuyCoke`" in the *Definition Browser* and right-click, select **Properties** option from the pull-down menu. On the *Definition Options* dialog in the *Ancestors* tab, the list of parent nodes of this definition is displayed. This tree is just another representation of Figure 3.9.

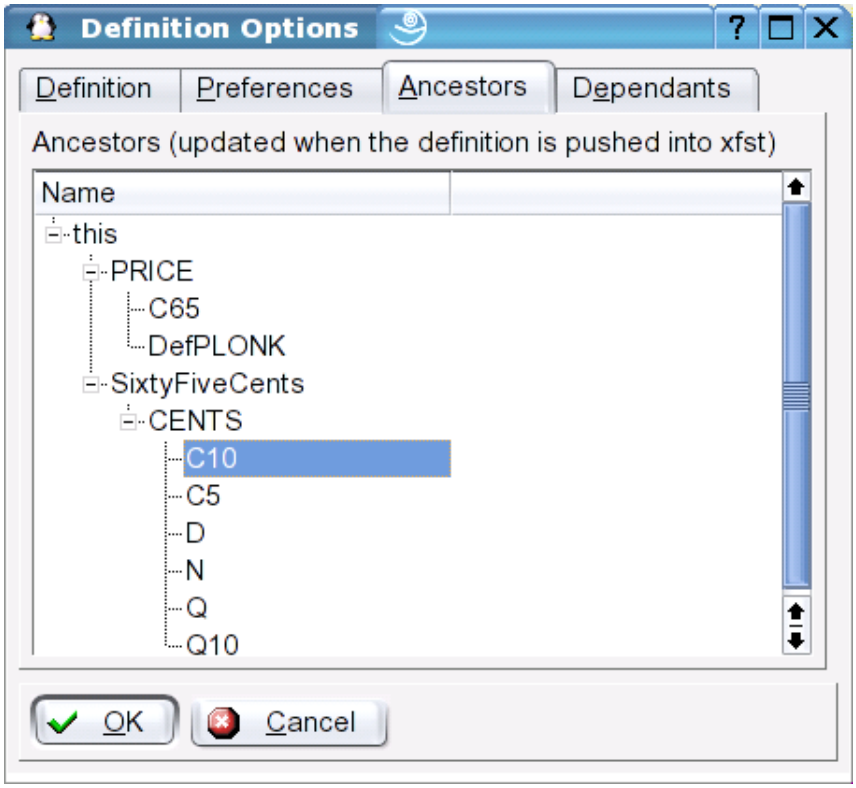


Figure 3.10: Ancestors tree view of a definition.

The reverse of an ancestors list is the dependents list. This is a tree of nodes that depends on a particular definition. To see which definitions depend on the price of the coke and will be updated, we select `C65` from the *Definition Browser* and click its properties in the pop-up menu. In the *Dependents* tab the list of dependents is given in a tree structure:

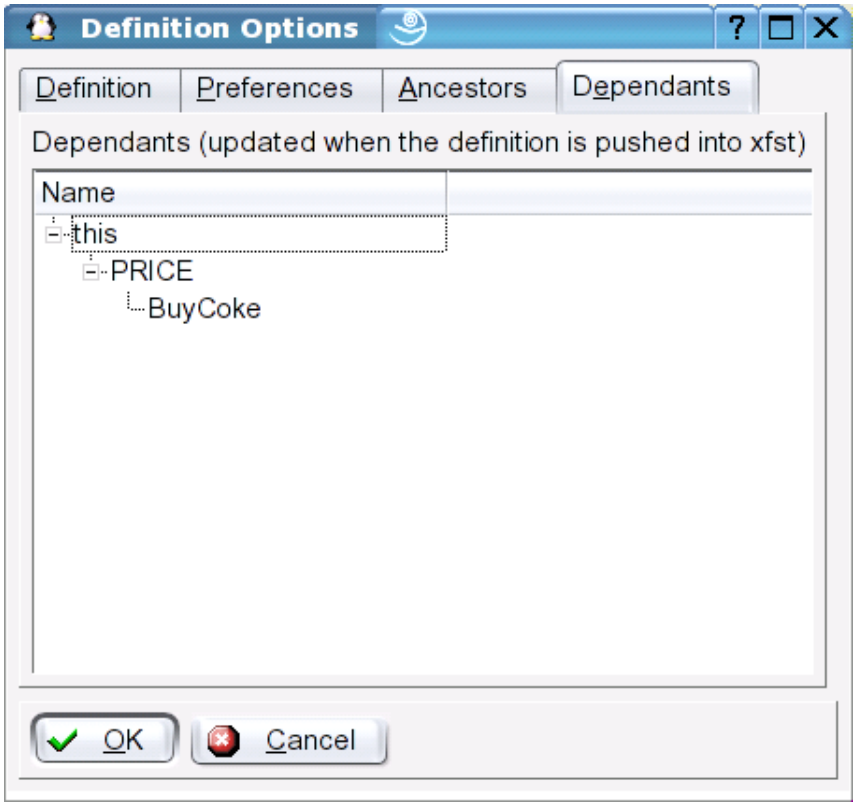


Figure 3.11: The dependants list of a definition.

Now we open the *Definition Options* dialog of `C65` if it is not already invoked. In the *Definition tab* expression "`c^65`" is changed to "`c^100`". Now a coke costs 100 cents. The dialog is closed with OK button to accept the update. As soon as the dialog is closed, Vi-XFST starts a sequence of compilations. The definitions in the dependents tree of the `C65` are now recompiled automatically by Vi-XFST. The results and the actual order of compilations can be viewed from the messages box.

One should observe that since these updated definitions are undefined and defined again, they are inserted at the top of the Definition Browser. By this way it is ensured that the ordering of browser items is consistent with the creation order of definitions, even if they are updated and not created.

To see how the ordering of definitions are changed as a recompilation takes place, we first get a list of definitions from XFST by using menu item **Definition|print|defined**. The ordering is shown in table:

```
N 340 bytes.  2 states, 1 arc, 1 path.
C5 452 bytes.  6 states, 5 arcs, 1 path.
C10 592 bytes.  11 states, 10 arcs, 1 path.
D 340 bytes.  2 states, 1 arc, 1 path.
Q 340 bytes.  2 states, 1 arc, 1 path.
Q10 1.0 Kb.  26 states, 25 arcs, 1 path.
CENTS 1.0 Kb.  26 states, 27 arcs, 3 paths.
SixtyFiveCents 1.0 Kb.  25 states, 27 arcs,
Circular.  C65 2.1 Kb.  66 states, 65 arcs, 1 path.
DefPLONK 340 bytes.  2 states, 1 arc, 1 path.
PRICE 2.1 Kb.  66 states, 65 arcs, 1 path.
BuyCoke 936 bytes.  14 states, 34 arcs, 634 paths.
```

Table 3.3: Output of print defined command.

Then we double click the definition `Q` in the Definition Browser, change its expression ”`q`” to ”`x`” and click OK. As the recompilation takes place, definitions are undefined and redefined automatically, one can observe that most recent updated definition is inserted at the top of definition list. After the process ends, we get a list of definitions again using **Definition|print|defined** menu. The ordering does not change in XFST.

```
N 340 bytes.  2 states, 1 arc, 1 path.
C5 452 bytes.  6 states, 5 arcs, 1 path.
C10 592 bytes.  11 states, 10 arcs, 1 path.
D 340 bytes.  2 states, 1 arc, 1 path.
Q 340 bytes.  2 states, 1 arc, 1 path.
Q10 1.0 Kb.  26 states, 25 arcs, 1 path.
CENTS 1.0 Kb.  26 states, 27 arcs, 3 paths.
SixtyFiveCents 1.0 Kb.  25 states, 27 arcs,
Circular.  C65 2.1 Kb.  66 states, 65 arcs, 1 path.
DefPLONK 340 bytes.  2 states, 1 arc, 1 path.
PRICE 2.1 Kb.  66 states, 65 arcs, 1 path.
BuyCoke 936 bytes.  14 states, 34 arcs, 634 paths.
```

Table 3.4: The ordering does not change although some definitions are redefined.

Unfortunately by using only this list generated by XFST, it is misleading to interpret that this is the creation order of definitions. It should not be referenced to figure out the dependency order of definitions created. But, the list in *Definition Browser* always shows the correct creation order of definitions.

## 3.6 Printing and Viewing the Source Code

The project source file can be viewed within the *Project Preview* dialog. We click **Project|View & Print** menu to invoke the dialog that will display the source code of our project.

We can use this dialog to export the project to a text file or print in various formats. *Hide all comments* checkbox can be used to hide/un-hide Vi-XFST inline control comments. Syntax highlighting can be enabled/disabled by the *Use syntax highlighting* checkbox. The **Print** button will call the system print dialog box and lets us choose the printing preferences and get a hardcopy of the project. If the underlying system permits, a postscript copy can also be generated from this printing dialog.

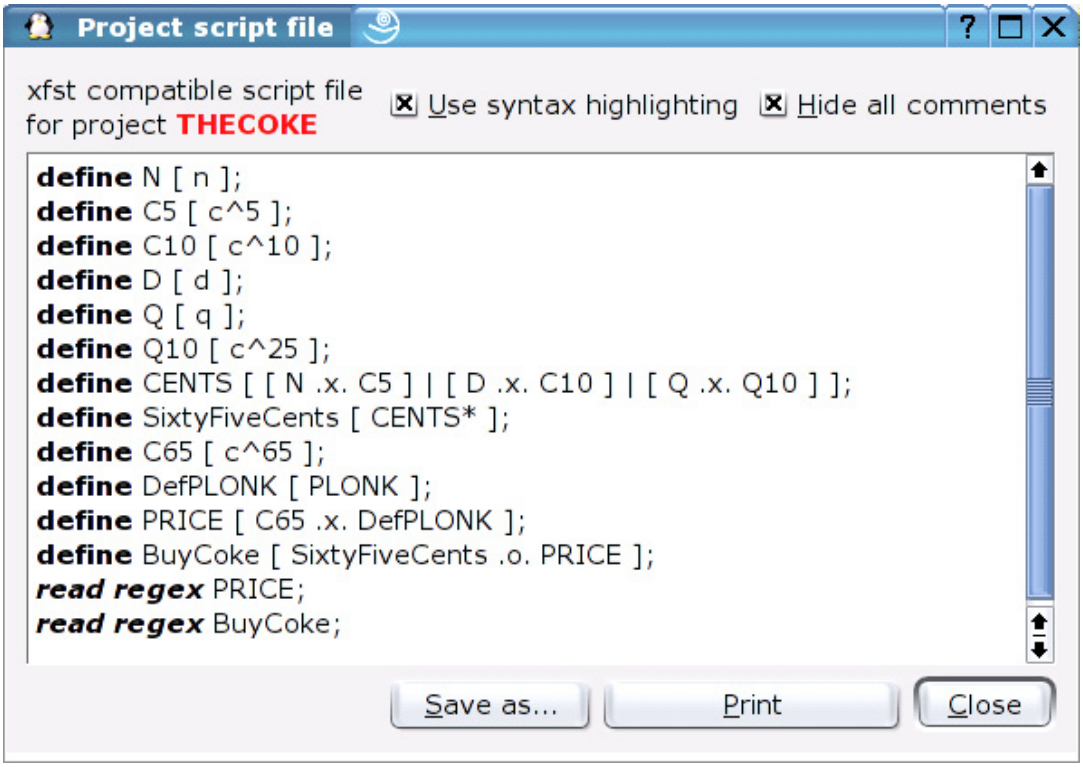


Figure 3.12: Project View dialog enables view, export and print of source file in different formats.

A copy of the project can be export a into a text file by using the **Save** button, according to the display criteria set in this dialog.

## 3.7 Exporting the Code and Binary Files

Under the project directory (see *Project Options* dialog), there are three files related to a project. These are:

**<ProjectName>.infproj** The source file for the project. It contains project information, options, network definitions and input strings. This file can be loaded into XFST with "-l" parameter. All the Vi-XFST generated codes are marked with "`##Vi-XFST##`" comment markers. But it is strongly advised not to edit this file manually. Instead, one should use the *Project View & Print* dialog described above to generate a user copy of the project source file.

**<ProjectName>.infdef** This binary file is created by the XFST "`save defined <filename>`" command automatically by Vi-XFST whenever the active project is saved. The binary file contains networks for all defined symbols in the project workspace. This file can be used in XFST with "`load defined <filename>`" command. Vi-XFST will try to locate this file when the project is loaded, but if it is not available, all definitions will be rebuild from the regular expression source file. But it cannot detect if this file is modified outside Vi-XFST, therefore the content of this file shall not be modified manually.

**<ProjectName>.infstack** This binary file is created by the XFST "`save stack <filename>`" command automatically by Vi-XFST whenever the active project is saved. The binary file contains networks on the stack of the project workspace. This file can be used in XFST with "`load stack <filename>`" command. Vi-XFST will try to locate this file when the project is loaded, but if it is not available, all networks will be rebuild from the source file. But it cannot detect if this file is modified outside Vi-XFST, therefore the content of this file shall not be modified manually.

Any modification on the stack will be effective in this binary file. So if one wants to prepare a binary transducer file to distribute without the source code, he can freely do any modification with the operators in Network menu. But he should remember that these modifications are not saved into project source file.

All of the files listed above, are compatible with XFST program. Any of them can be distributed to other users. But only the project file (with extension *.infproj*) can be loaded back to Vi-XFST.

If the project file seems confusing with many inline comment blocks put by Vi-XFST, a more tidy file copy may be produced by *Project Preview* dialog described in Section 3.7.

# Chapter 4

# Vi-XFST Development Issues

## 4.1 Software Design

The software design of the project is to address the requirements of developing large-scale finite-state networks and to ease the development process. In the software created, we have demonstrated these solutions. However, this first version is not still a fully comprehensive development environment that can encapsulate all the functionality of the XFST system. For example not all the XFST calculus is implemented in the visual expression building feature, or some XFST commands are excluded from the project because of some implimentation restrictions.

It should be kept in mind that the software designed here, is not a new finite-state toolkit, and is not a replacement for XFST. The Vi-XFST project is a supplementary tool to manage the XFST compiler. Therefore, the features of the software are strictly related to the XFST program, and should not be evaluated by neglecting this relation. On the other hand, it is relatively eash to adapt this software and its model to other finite-state toolkits such as Van Noord's FSA (Finite State Automata Utilities) [6] or FSM Library tools from AT&T [5].

XFST is available in various platforms such as for Sun Solaris, Linux and Microsoft Windows systems. Therefore we have designed our development application compatible with many operating systems. The graphical interface is based on a QT library which supports various platforms. This library also provided some important base classes used in process management and thread support in our application. Please see Section 5.3.1 for more information about QT library.

### 4.1.1 Concepts in Vi-XFST

Some key concepts in Vi-XFST design and development environment have to be defined prior to discussing the detail of the software. These are:

**Project:** A project is a session in which the user can create, load, modify and run an XFST script file. For Vi-XFST, a project is not only the XFST file. It is a development session, with graphical objects, definition entries, input and output string lists and more. A project session can be saved and loaded from a project file. Unfortunately not all activities are saved in this version of Vi-XFST, such as the print commands, or network operations like *epsilon-remove*. The project concept is coded in `CProject` class.

**XFST Process:** Whenever a project is activated in Vi-XFST, an XFST must be started in the background as a separate process to access the finite-state operations. The process can be viewed by system tools such as "*ps*" command on Unix systems. The regular expression operations are carried out with this XFST process. A better approach may be to use a application programming interface (API). But unfortunately such an API is not available for XFST. Vi-XFST passes commands to this process and receives outputs form it. If this process killed by some way, Vi-XFST will not be able to complete user commands and it will give an error messages.

**Definition:** It is an entity that symbolically represents a regular expression in XFST process. It also includes many additional concepts, other than a simple regular expression, such as comments and dependency lists associated with the class instance. It is discussed in more detail and technically in Section 4.2.3.5.

**Network:** is an entity that symbolically represents a finite-state machine in XFST process. A network instance in Vi-XFST also has comments, dependency lists and associated definition that is used in construction of this network. It is discussed in more detail and technically in Section 4.2.3.6.

**Test Phase:** is the process of applying strings to the top network on the stack. Test phase is initiated by status of the stack. If the stack is empty most of the test functions are disabled automatically.

**Definition Dependency:** If a regular expression definition is composed using another definition, then the new definition is called as a dependent of the previous definition, or the previous definition is an ancestor of the new definition. The dependency of regular expression definitions is used in automatic recompilation procedure.

**Network Dependency:** A network only depends on the definition that it is composed of. When the definition is recompiled at some step, the dependent network is also recompiled in the stack.

### 4.1.2 Design Principles

The main tenet, for both for high level and for implementation designs, it to stick to object oriented techniques. The tasks of the project are divided into three main components: *the main*

*window, project class, and XFST process*. Their responsibilities and interactions are stated at the top level before the actual detailed design.

With the recursive application of the above principle to each main component, they are also divided into smaller classes, with additional new auxiliary ones. At the final depth, the behaviors lead to method and property definitions in the actual class bodies. More detail about class hierarchies and designs is provided in the following sections.

As the main classes can be grouped into three, the majority of methods in the actual project may be grouped into three to get a better understanding of the implementation:

1. XFST - Vi-XFST interprocess communication-handling and command functions.
2. Graphical components classes and functions.
3. User communication functions.

These groups are not necessarily restricted to unique classes. A class may possess methods from any of these groups. For example the `CProject` class has functions that handles XFST communications (such as acceptDefinitionDefine), graphical component functions (such as openAWorkPage), or from group 3, the user communication functions (like slot_define_definition). However, most of the functions that are defined in Vi-XFST can be included in one these groups.

The idea for defining such a virtual functional grouping in the design phase is that, for each of these groups, similar algorithms, design approaches and coding styles have been used. This has greatly increased the understandability of the code. Once the logic in one group has been understood, it is easy to handle the rest of the methods in the same group.

For example, in this version of Vi-XFST, there are 49 XFST commands implemented[1]. These command executions requires following execution steps which are carried out with different methods:

- User command initiation,
- CProject command execution preparation,
- CXfst command execution and result evaluation,
- CProject command execution success/failure actions.

[1]One can get the number of XFST access methods with:

```
#cat cxfst.h | grep run_xfst | wc -l
49
```

For every XFST command implemented in Vi-XFST, there is a path over these functions. And these functions are not necessarily same for all commands, because each of the commands may require different handlers than others. These methods are scattered in `CProject`, `CXfst`, `CDefinition` and `CNetwork` classes. For 49 commands, the approximate number of methods is 300, excluding many auxiliary methods that connect the execution flow to the other components of the Vi-XFST. Hopefully, the uniform design of similar functional methods reduces the overhead of maintaining and handling of this many numbers of functions in big software project.

The flow diagram for a "*print network*" command execution is no different than a more complex "*save network*" command. Understanding one flow diagram helps to understand the whole structure. This reduces the complexity of adding new commands to Vi-XFST and debugging phases.

### 4.1.3 Execution Flow

Main execution flow is a high-level design view of the key components of the system:

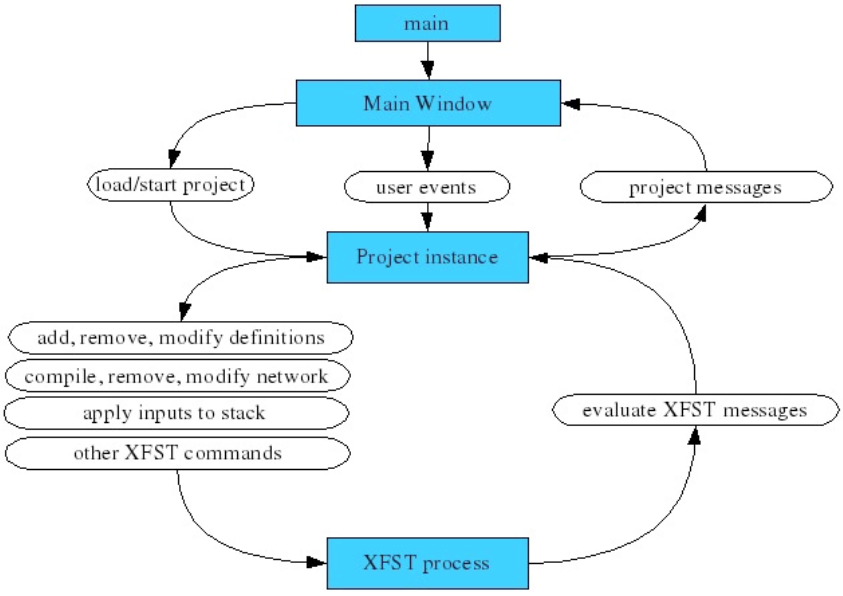


Figure 4.1: Main execution flow diagram of Vi-XFST

The internal flow paths of certain tasks, such as definition parsing on visual expression canvas, project saving and loading or XFST command execution, may be much more complex. The following figures illustrates a typical path of XFST command execution flow:

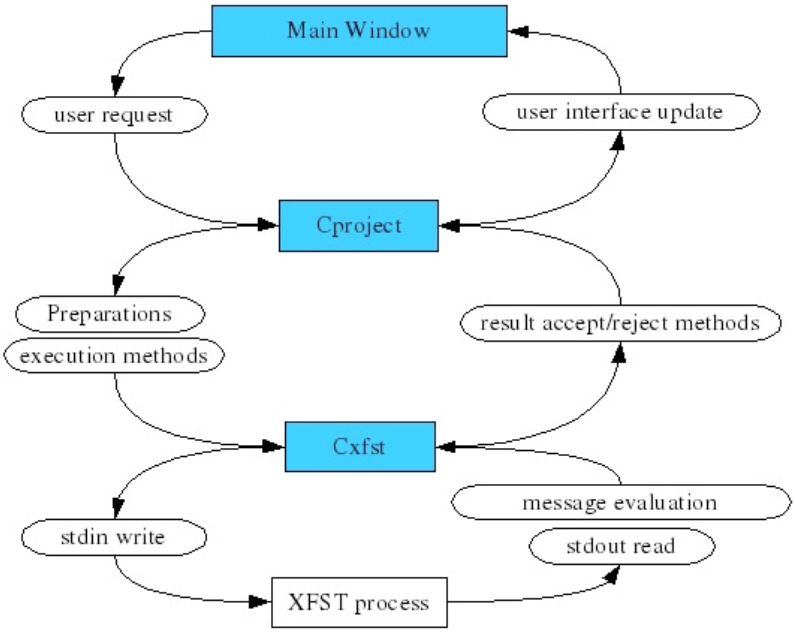


Figure 4.2: A XFST command execution flow path.

### 4.1.4 Informal Coding Rules

Beside design principles, there are also some informal rules used in the design and code to keep the project uniform and understandable:

1. Minimum number of global variables and static members.
2. Short function bodies. Functions should do only one job, but do it perfectly, no more or no less.
3. Meaningful and readable names for variable and methods.
4. Source file lengths less then ~1000 lines. Longer files are split into smaller ones.
5. Checking memory allocation with debug codes and checking for `NULL` pointers before accessing a parameter in a method.
6. Using get and set methods to access a class member, avoiding direct access to members.
7. Using only platform independent libraries.

### 4.1.5 Interprocess Communication

XFST program is run in the background as a process while the Vi-XFST interacts with the user. The XFST process is initiated when a project is loaded on the main window. The process remains active as long as the associated project is active.

Vi-XFST is only an interface and manager for XFST process. All actual regular expression operations are done inside this process. Vi-XFST prepares commands and writes them to the standard input of XFST process, and reads results from standard output and standard error. All of these interpretations are carried out in `CXfst` class. The entry points to this class are methods associated with a command to be executed. The result of execution is a signal that implies either a success or a failure. The signal also includes the XFST message returned. The internal steps of this execution flow involve command execution setup, command execution, flushing of standard output and errors, output evaluation and emitting result signals.

One of the important concepts in `CXfst` class, is the buffer flushing mechanism for *stdout* and *stderr*. To understand the mechanism, lets look at how we interpret a command sent to XFST:

After a command is written to the XFST program standard input, we cannot be sure when the command execution ended by looking at the output messages. Because not every command returns a reply. To ensure a reply is produced for every command, we send extra echo commands before and after each functional command. So we are expecting at least our echo messages, even though the actual command may return nothing.

Here is an example command execution step: the requested command is the line starting with "*define ...*"

*echo start_of_command;*
*define FAMILY [ yasin | aliye | yusuf | bilge ];*
*echo end_of_command;*

The expected output is:

*start_of_command*
*defined FAMILY: 260 bytes. 2 states, 4 arcs, 4 paths.*
*end_of_command*

So everything between a '*start_of_command*' and '*end_of_command*' keywords is the output returned by XFST. If we only see a '*start_of_command*', and not received an '*end_of_command*' yet, this means that XFST is still busy with our command, and we have to wait more.

Unfortunately interprocess communication channels between XFST process and the `QProcess` class has an internal buffer. QT library does not give access to these low level options to adjust internal buffer sizes. So when we send a command to XFST, we may not get a reply immediately. This is not what we want in an interactive development environment.

So we try to fill the stdout buffer with echoing dummy characters. The amount of this flush strings is determined with `adjust_xfst_buffer()` method with the startup of `CXfst` class. This echo commands flushes the *stdout* and *stderr* on Unix systems. But on Microsoft

Windows systems, we saw that this method only flushes the *stdout*. So we have developed another function to flush the stderr buffer specially designed for Windows systems, which is also fine for Unix systems. A simple solution to flush the standard error is to load a dummy binary network file with `"load stack"` command in XFST. This small network file is loaded and unloaded each time a command is run in XFST to get results from the standard error buffer. A better approach would be to use a application programming interface (API) which will not cause these kind of interprocess communication tricks. But unfortunately such an API is not available for XFST.

Once the messages are received from either from the standard output or error, the results are evaluated in the same class to understand whether it is an accept or an error message. And the calling class is informed accordingly.

### 4.1.6 Debug Techniques

Vi-XFST software contains additional debug lines to ease debugging of the code. QT library auxiliary function *"qDebug"* generates the debug messages. Nearly all debug messages are encapsulated between `"#ifdef _XXX_DEBUG_"` and `"#endif"` preprocessor commands, like:

```
if (isRunning())
{
#ifdef _CXFST_DEBUG_
    qDebug("Xfst didnot exit, we are going to kill it!");
#endif
    Process->tryTerminate();
    QTimer::singleShot(1000,Process,SLOT(kill()));
    qApp->processEvents(1000);
}
```

Table 4.1: A sample debug block

Therefore the *qDebug* messages are available only these `"_XXX_DEBUG_"` definitions are defined prior to the debug line. And also the compiler should be configured with the debug options on. Under X11, the debug messages are printed to stderr. Under Windows, the text is sent to the debugger. Or if the user clicks the *install message handler* button on Debug Tab the messages will be appended into the debug window.

### 4.1.7 Files

Auxiliary files used beside the Vi-XFST sources and binary are;

**Settings file or System Registry Database** On Unix systems Vi-XFST program stores its options in a settings text file under `$HOME/.qt/` directory in a file `analyzerrc`. On windows system the Vi-XFST settings are saved into system registry database under `"HKEY_CURRENT_USER/Software/analyzerrc"` path.

**Image files** There are various icons prepared for Vi-XFST program. Vi-XFST searches these icons in the *Images* directory in the same folder as the program executable binary.

**Stderr Flushing file** Stderr of XFST cannot be flushed with echo commands as done with flushing the standard output. A simple solution to flush the standard error is to load a dummy binary network file with `"load stack"` command in XFST. This small network file is loaded and unloaded each time a command is run in XFST to get results from stderr.

**Project files** All project files are saved in the active project directory (see *Project Options* dialog). There are three files related to a project. These are:

- *<ProjectName>.infproj:* The source file for the project. It contains project information, options, network definitions and input strings.
- *<ProjectName>.infdef:* This binary file is created by the XFST "save defined <filename>" command automatically by Vi-XFST whenever the active project is saved. The binary file contains networks for all defined symbols in the project workspace.
- *<ProjectName>.infstack:* A binary file created by the XFST "save stack <filename>" command automatically by Vi-XFST whenever the active project is saved. The binary file contains networks on the stack of the project workspace.

For more information about the usage of these files please refer to Section 3.7.

**Exported print files** Some of the results generated by XFST can be exported to a text file. These are the ”print” commands which give information about entities in the XFST, such as definitions, networks, languages accepted etc.

**Exported message files** The messages in the *message window* or *test results window* can be written into text files. These files can be used for future reference or debugging of the project.

## 4.2 Vi-XFST Classes

### 4.2.1 Class Hierarchies

The code design of Vi-XFST heavily uses object-oriented techniques, such as inheritance, polymorphism, virtual functions and more. Therefore it is necessary to understand the hierarchy of the classes and their relation with each other. Most of the classes are inherited from QT based classes, mainly the `QObject` which is the base class of all QT objects. `QObject` is the heart of the QT object model. To enable signal-slot communications between two classes, `QObject` must be inherited. For more information about signal-slot communications refer to QT documentation [9].

Another important class hierarchy tree is the one starting with `CShape`, shown in Figure 4.3. `CShape` is the base class of graphical components that are displayed within Vi-XFST. It has common properties for all shapes, such as size, location on the screen, background color, a pointer to parent class, and another pointer to the parent widget on the screen. `CRectangle` immediately follows this class in the inheritance hierarchy. Since all of the shapes in Vi-XFST are in rectangles, displayed with a `QFrame`, it is reasonable to have all shapes inheriting from `CRectangle`. This class introduces the drawing widget of a rectangle, which is the `QFrame` inside a CCanvasFrame class. So everything visible within Vi-XFST workspace are `QFrame's`, except some `QLabels` and `QButtons` on definition rectangles.

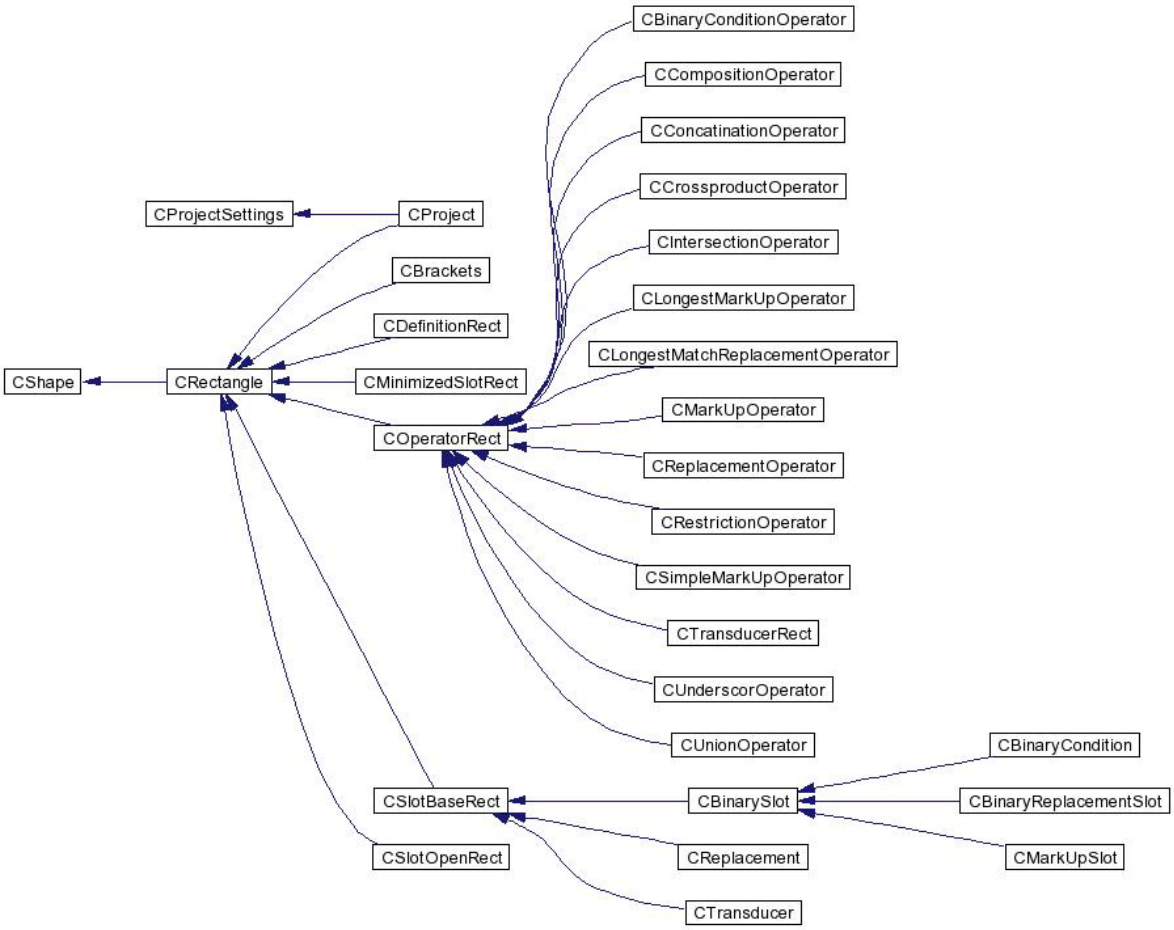


Figure 4.3: CShape class inheritance hierarchy.

Another important hierarchy is in Figure 4.3 is the subtree starting with `CSlotBaseRect`. It is the main class for all operator rectangles. An operator base is the visible shape on the workspace, on which the user can insert/remove definition rectangles or other operation bases. `CSlotBaseRect` implements many of the common operations for all base classes. It manages the definition rectangle insertion and removals, pop-up options, basic drawing functions and more. But for every operation, there may be different requirements, for example to handle `getString()` function that returns the string represented by the definition slot. Or the `positionObjects()` method, which positions items contained within a definition slot. These methods are reimplemented in the descendant classes that inherit from `CSlotBaseRect`.

Multiple-inheritance is a feature of C++ object oriented model. It is not recommended since it sometimes makes the code a bit complicated. The only where it is used is for `CProject` class, which inherits both from `CRectangle` and `CProjectSettings`. Those two ancestor classes are quite different in their functionalities; therefore they cause less confusion in multiple-inheritance tree.

### 4.2.2 Compound Class List

Here are the classes defined in Vi-XFST project, with brief descriptions:

**CBinaryCondition** A graphical base for binary conditionals for replacement operators

**CBinaryCondition** Operator class for operators used in `CBinaryCondition` slots

**CBinaryReplacementSlot** Implements the graphical base class for replacement, restriction and markup operators. This class differs from `CBinarySlot` because of its fixed size parameters and pop-up menu items

**CBinarySlot** Base class for all binary slots. It is also the base class for most of ordinary operators like *union/concatenation/restriction* etc. Other bases for more complex operations may inherit from `CBinarySlot`

**CBrackets** Base class for a bracket on the canvas. A bracket can hold only one object inside. It can be a definition, or a `CSlotBaseRect`, and it is held in variable *BracketRect*

**CCanvasFrame** Handles all operations about a canvas, on which the graphical components are drawn. The mouse move event handlers, drag drop operations, chain-object delete orders of destructors are implemented in this class. Any `QFrame` used in the project workspace must inherit from this class

**CCompositionOperator** Base class for the composition icon

**CConcatinationOperator** Base class for the concatenation icon

**CCrossproductOperator** Base class for the crossproduct icon

**CDefinition** Base class for any regular expression definition. All definition related functionalities are implemented here

**CDefinitionList** Holds the list of definitions used in this project. Definitions are held in this class by value not by reference. Any method that requires access to a definition takes a copy of it from the list, it does not access the actual definition

**CDefinitionParser** Implements a static parser method to parse a definition into graphical presentation. The definition can be converted from string representation into graphical hierarchical objects on a `CSlotBase` object. The `parse()` method is where this parsing is initiated

**CDefinitionRect** Base class for displaying a definition on any canvas that inherits from `QWidget`. Usually this widget is one of the operator bases, like `CSlotBaseRect`. All user interactions with a definition can be accessed using this class, with pop-up menus, drag-drop operations, mouse clicks etc

**CFormEditDefinition** Display a dialog box for the o class properties. This dialog box can is also used to define a new `CDefinition` object or modifying an existing one

**CFormEditNetwork** Display a dialog box for the `CNetwork` class properties. This dialog box can is also used to define a new `CNetwork` object or modifying an existing one

**CFormEditProject** Display a dialog box for the `CProject` class properties. This dialog box can is also used to define a new `CProject` object or modifying an existing one

**CFormMain** This is the main graphical form of Vi-XFST. Most of the project functions are initiated by the user from this class. This class may have bidirectional interactions with other class instances, such as `CProject`

**CFormName** Implements a dialog box to get a filename. Inherits from `formFileName`

**CFormProjectView** Implements a dialog box to preview the project script file. This class is also used to saving and printing of the project file in many different formats. Inherits from `FormProjectView`

**CFormXfst** Implements a progress dialog box while the XFST process is active. Inherits from `FormXfst`

**CfrmAbout** Inherits from `formAbout` class. That opens the about dialog box

**CfrmOptions** This class is used to implement a graphical dialog box to set and get the system settings

**CIconView** This is a derived class from `QIconView`. The main purpose is to add additions to the mouse events

**CIntersectionOperator** Base class for the intersection icon

**CLabel** This is a derived class from `QLabel`. The main purpose is to add additions to the mouse events

**CLongestMarkUpOperator** Base class for the longest markup operator icon

**CLongestMatchReplacementOperator** Base class for the longest-match-replacement operator icon

**CMarkUpOperator** Base class for the markup operator icon

**CMarkUpSlot** This class implements a markup operation slot

**CMinimizedSlotRect** This class implements a base rectangle to display minimized graphical components

**CMyList** A derived class from `QIconView`. The main purpose is to add additional functionalities to the standard `QListView` such as mouse and drag/drop events

**CNetwork** This class is the base for XFST networks in the stack

**CNetworkList** This class holds the list of networks on the stack

**COperatorRect** Defines a class that will represent the operator objects displayed on canvas

**COptions** Holds all the application settings and methods to access, save and load them. This settings-class is operating system independent. It uses a file in *$HOME/.qt* on Unix machines and the system registry for the Microsoft Windows operating systems. `COptions` class also holds the define variables used in the project source code. Most of these constants are settings default values, which are used if the Vi-XFST settings are not available for any reason

**CProject** The main class of the active Vi-XFST project. All project functions are managed within this class

**CProjectSettings** Holds the settings of a project in a `QSettings` based class

**CRectangle** Base class of any rectangle on the canvas. This class activates many base slots on the frame such as mouse actions. If they are needed, one should just override them in his class

**CReplacement** This class implements a base for the multi-line operators such as replacement, conditional replacement, longest match etc

**CReplacementOperator** Base class for the replacement operator icon

**CRestrictionOperator** Base class for the restriction icon

**CShape** This is the base class of any visible shape on the canvas. All of the graphical objects used by Vi-XFST must inherit from this class. The virtual functions of this class must be implemented on derived classes, such as the `'draw()'` method

**CSimpleMarkUpOperator** Base class for the simple markup operator icon

**CSlotBaseRect** This is the base class for a *slot canvas*

**CSlotOpenRect** This class represents a rectangle on a slot rectangle, where it is possible to drag and drop definitions inside

**CTest** This class implements an object to hold the properties of a test on XFST. The string applied to XFST is considered a test item. This class also implements methods to save/load tests from project files

**CTransducer** This is the base class for operator slots for transducers

**CTransducerRect** This class implements the icons of a transducer that points the input/output locations of the transducer. The class is implemented by inheriting from the `COperatorRect` class

**CUnderscorOperator** Base class for the underscore operator icon

**CUnionOperator** Base class for the union icon

**CWorkTab** A workspace class that is connected to every tab page of the active workspace

**CXfst** XFST interprocess management class. XFST process is controlled inside this class

### 4.2.3 Main Classes

#### 4.2.3.1 CFormMain

This is the main window of Vi-XFST. Most of user interactions are initiated from this class.

`CFormMain` inherits from class `formMain` that is prepared by QT user interface designer tool: *Designer*. Once the design of the GUI is completed, `CFormMain` overrides the associated slots of user interface events, such as `slot_btnReplace_clicked()`. Rewriting the contents of these overridden virtual functions, now `CFormMain` can accept the user events emitted by the QT main event handler and process them.

Most of the methods of `CFormMain` do not nothing but just pass the request to the active `CProject` class instance. `CProject` is the main class where all the actual work is done. For example when the user clicks compile option, the `slot_ActionCompile_activated()` slot is triggered inside `CFormMain`. This slot just passes the request to the active `CProject` as:

```
if (getActiveProject())
{
    getActiveProject()->slot_Base_DefinitionCompile(NULL);
}
```

`CProject` can also access back to `CFormMain`. It has a pointer to the main window. The action results can be displayed on the main window by accessing the public slots of `CFormMain` using this pointer.

At this moment `CFormMain` can handle only one active project. But the class is designed with the future extensions so that it can handle more than one project at a time. With a little modification it may be possible to work on more than one project file. `CFormMain` will be one of the few globally shared objects between `CProject` instances (Another global class is the `COptions`, where options for Vi-XFST are accessed). So it will be wiser to control access to these classes in a multi-project version of Vi-XFST.

Other tasks of `CFormMain` are to enable/disable screen controls such as some buttons available only for certain states of the project. Most of these updates are group into three methods:

- `prepare_definition_gui()`
- `prepare_network_gui()`
- `prepare_recentProjects()`

`CFormMain` also handles the Vi-XFST initializations on startup and some necessary actions on close of the application. These processes are initiated in constructor and destructor of the class. For example, loading the `COptions` class from settings database (system registry on windows platforms) is a task done in constructor of `CFormMain`. In the same way, `COptions` is saved back to settings system, in the `closeEvent()` of `CFormMain` instance.

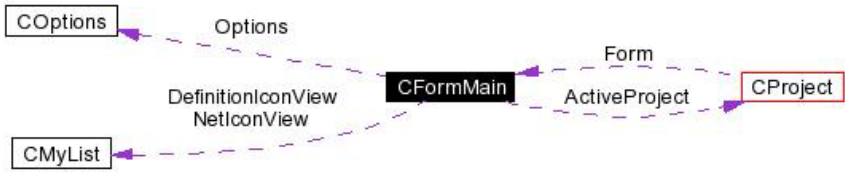


Figure 4.4: Collaboration diagram for CFormMain

#### 4.2.3.2 CProject

The main class of the active Vi-XFST project. All project functions are managed within this class.

For every project opened in Vi-XFST, a new `CProject` class is constructed. Regular expression definitions, network operations, testing, generating XFST scripts and most of the other functions of Vi-XFST are handled here. The `CProject` class has access points to other components of the Vi-XFST design. Most important of these entities are:

1. **The user:** Through the `CFormMain` class, user may initiate actions, by pressing a button, or using the mouse. The `CFormMain`, which is the main Vi-XFST window, interacts with the user, and passes user requests to `CProject` with its corresponding public methods. After `CProject` precesses these requests, to return the results back to the user, `CProject` must be able to communicate to the main window. This is communication is established by passing a pointer to the main windows instance, in the constructor of the class. `CProject` uses this pointer to access public methods and public slots of `CFormMain` to adjust the user interface to display the execution results and current state of the project.

2. **XFST:** Through `CXfst` class. When the `CProject` class is activated and displayed with Display() method, an instance of `CXfst` is constructed and XFST process is started. The commands from the user are passed to `CXfst` public methods. `run_xfst_MinimizeNet`). The results of these command executions are returned back to `CProject` with signals of `CXfst` to the slots of `CProject`. The sign of `CXfst` that `CProject` listens are:
   `CXfst::signal_Reject()`
   `CXfst::signal_accept()`
   And the corresponding listening slots of CProject:
   `CProject::slot_Reject()`
   `CProject::slot_Accept()`

3. **Graphical Objects:** These are the graphical components on the workspace, which represent the regular expressions. The base class of these components is the `CSlotBase` class. `CProject` gets signals from `CSlotBase` class, to its associated slots. These signals may refer to user requests, such as adding a new `CDefinitionRect` to the active base, or may refer to component requests such as for redrawing the active slot on the screen.

4. **File system:** `CProject` interacts with the file system to load and save a project. Process of saving or loading a project is always under the control of `CProject`. When the process is started, the execution is sequentially transferred to relevant other classes such as `CDefinition`, `CTest` etc. Once all descendant classes complete their work with the file I/O, `CProcess` returns success from the process, and adjust the state of Vi-XFST accordingly.

Other functions of the class `CProject` are for managing the project options, definition lists, dependency controls of definitions, printing support and test-phase controls.

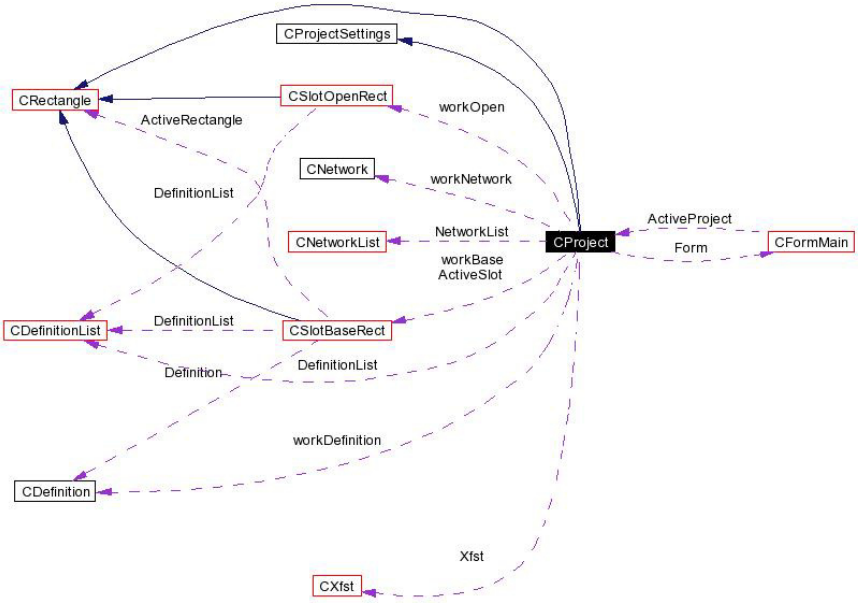


Figure 4.5: Collaboration diagram for CProject:

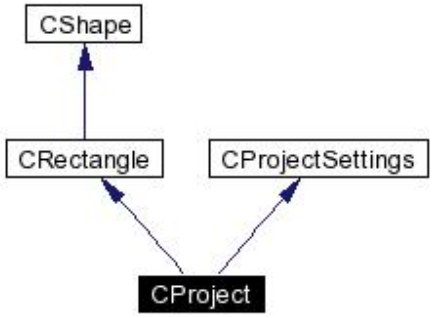


Figure 4.6: Inheritance diagram for CProject

#### 4.2.3.3 CXfst

XFST interprocess management class. XFST process is controlled inside this class.

The main purpose of this `CXfst` is to encapsulate XFST program as a system process and to control the interprocess communications with this process. Once the `CXfst` is constructed, it has access to `COptions` class, where it can get the XFST program location on the file system.

The `startXfst()` method initiates and runs the XFST program in a QT `QProcess` class. The standard input, standard error and standard output of the process are linked to appropriate slots of the `CXfst` class. The commands are written to the stdin, while the outputs are read from the stderr or stdout of this process.

One of the important concepts in `CXfst` class is the buffer flushing mechanism for stdout and stderr. To understand the mechanism, lets look at how we interpret a command sent to XFST.

After a command is sent to XFST program, we cannot be sure when the command execution ended. Because every command does not return a reply that we can decide the end of execution with. To ensure a reply is produced for every command, we send an extra echo command before and after each functional command. So we are expecting at least our echoes, even the actual command may return nothing.

Here is an example command execution step, the requested command is the line starting with "*define ...*":

*echo start_of_command;*
*define NOUN [ osman | zeynep | kazim | defter ];*
*echo end_of_command;*

The expected output is:

*start_of_command*
*defined NOUN: 260 bytes. 2 states, 4 arcs, 4 paths.*
*end_of_command*

So everything between a '*start_of_command*' and '*end_of_command*' keywords is the output returned by XFST. If we see a '*start_of_command*' only, this means that XFST is still busy with our command, and we have to wait.

Unfortunately interprocess communication channels between XFST process and the `QProcess` class has an internal buffer. QT library does not give access to these low level options. So when we send a command to XFST, we may not get a reply immediately. This is not what we want in an interactive development environment.

So we try to fill the stdout buffer with echoing dummy characters. The amount of this flush strings is determined with `adjust_xfst_buffer()` method with the startup of `CXfst` class. This echo commands flushes the *stdout* and *stderr* on Unix systems. But on windows systems, we saw that this method only flushes the *stdout*. So we have developed another function to flush the *stderr* buffer specially designed for Windows systems, which is also fine for Unix systems. A simple solution to flush the standard error is to load a dummy binary network file with `"load stack"` command in XFST. This small network file is loaded and unloaded each time a command is run in XFST to get results from the standard error buffer. A better approach would be to use a application programming interface (*API*) which will not cause these kind of interprocess communication tricks. But unfortunately such an API is not available for XFST.

Once a command is run, we expect an output. After we receive the output on stdout, we have to decide whether the command is a success, or a failure. A failure may be because of a syntax or a logic error. Before every command execution, a regular expression is set to decide an output is an error message or not. For example a definition accept regular expression is:

```
"\b([dD]efined [\S]*:  )
```

With the same way, a regular expression error message is detected by the following expression:

```
"\b\*\*\* Regex error\b"
```

The output evaluation is done inside `evaluateOutput()`. According to the result of the evaluation the one of the two signals is emitted:

```
signal_accept()
signal_Reject()
```

The `CProject` slots accept these signals. Emitting these signals is the final job of `CXfst` in command execution. Then `CXfst` class returns execution to calling function. XFST process stays in the memory waiting for more commands, until it is stopped by `stopXfst()` method.

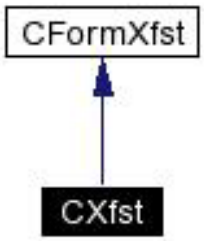


Figure 4.7: Inheritance diagram for CXfst

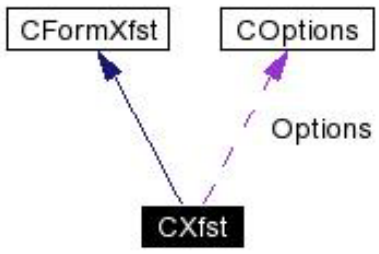


Figure 4.8: Collaboration diagram for CXfst

#### 4.2.3.4 CSlotBaseRect

This class is the base of other graphical operator classes used in visual regular expression construction. It includes most of the common methods and properties of an operator base, such as child lists, many of the command slots, drawing methods, definition rectangle handling functions any many more. The classes that inherit from `CSlotBaseRect` have to modify only a few methods for their needs. The rest of the code is the same for all.

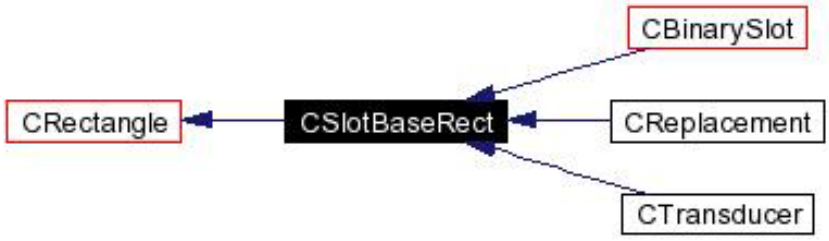


Figure 4.9: Inheritance diagram for CSlotBaseRect

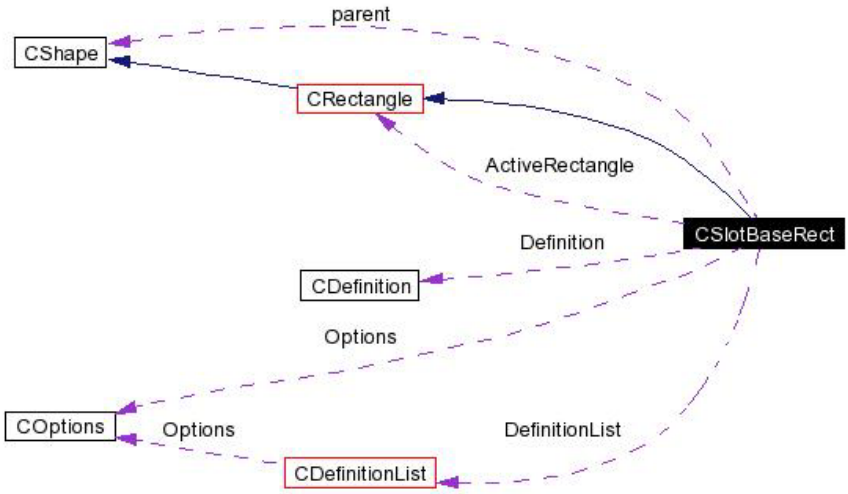


Figure 4.10: Collaboration diagram for CSlotBaseRect

#### 4.2.3.5 CDefinition

This class represents a single regular expression definition entity inside Vi-XFST. A `CDefinition` in `CDefinitionList` class represents the definition instance in XFST stack. All settings, functions related to a definition are handled in this class.

A definition's main attributes are name, expression and the comment fields. *"name"* and *"expression"* values are also available in XFST. When the user creates a definition as:

```
define BROTHERS [ Ibrahim | Cevdet | Yasin | Azim ];
```

The *"BROTHERS"* is the unique name of the definition and string after the name is the *"expression"* value for the definition. Vi-XFST introduces another important attribute to every definition in the project: the comment field. These primary values of `CDefinition` can be set either in the constructor or later with appropriate set methods.

The name of a definition is always unique. This restriction is not checked by XFST and the previous definition is overwritten with the new one. This may cause confusions and errors, because the user is not given any warning at all when a second definition with the same name is defined. In Vi-XFST each definition name is indexed in `CDefinitionList`::`DefNames` so that no two `CDefinition` with same name can be defined into XFST stack, until the previous one is undefined.

A definition can be composed by using previous definitions. In this case, we say; it has dependency on these ancestors. Any change in the ancestor definitions should affect the children. But this is not the case in XFST. Once a definition is compiled using other definitions, it is not updated until it is redefined manually. For example:

```
define GRADE_LETTERS [ A | B | C | D | E ];

define GRADES GRADE_LETTERS[-|+];
```

The definition *GRADES* can accept `A+,B-,C,` etc. But we realized that we forget the great *'F'*. So we update *GRADE_LETTERS* as:

```
define GRADE_LETTERS [ A | B | C | D | E | F ];
```

But unfortunately this modification is not reflected in *GRADES* definition. It still points to the previous finite-state machine. It has to be redefined:

```
define GRADES GRADE_LETTERS[-|+];
```

This introduces a problem, which is not handled in XFST. Suppose our definition is much more complex; lets say we use definition *GRADE_LETTERS* in many other expressions. We try to update them, but it is possible that we may also forget some of them. Vi-XFST solves this problem by checking dependencies of each definition to handle these updates. Once a definition is modified, Vi-XFST redefines its dependents, without any user intervention. Even the networks that depend on one of these redefined definitions are re-compiled by Vi-XFST.

#### 4.2.3.6 CNetwork

This class represents a network in the XFST stack. All settings, functions related to a network are handled in this class.

A network's main attributes are name, definition name it is created from and the comment fields. In XFST a network can be compiled as:

```
read regex GRADE_NET
```

or

```
read regex [ A | B | C | D | E | F ];
```

In Vi-XFST a network can only be defined from a single definition as in the first example above:

```
read regex GRADE_NET
```

This does not reduce any functionality on XFST, but enforces the user to follow a unique development style.

As mentioned in Section 4.2.3.5, network's dependencies are handled as a part of `CDefinition` dependency compilations. When a definition is modified and redefined, its dependent networks are also re-compiled into XFST stack.

#### 4.2.3.7 CDefinitionParser

This class implements a static parser method to parse a definition string into graphical presentation. The definition can be converted from string representation into graphical objects on a `QWidget` drawing canvas. The `parse()` method is where this parsing is initiated.

The `getString()` function of classes that inherit from `CSlotBase`, and `CDefinition`, returns the string represented by the graphical object tree on the canvas. The `CDefinitionParser` class introduces the reverse functionality; it can build the graphical object tree back on to the canvas from a given string.

The `parse()` function is the entry point to this definition parser. This method includes a finite-state machine to do the parsing. The state diagram is shown in Figure 4.11.

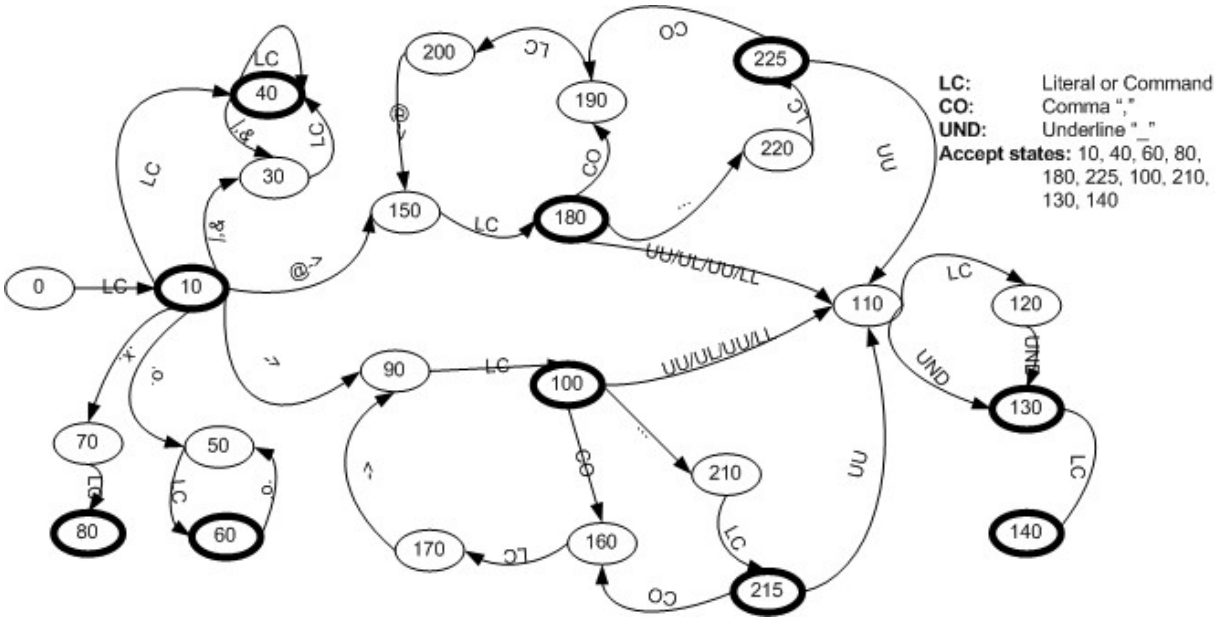


Figure 4.11: CDefinitionParser::parse() state diagram to parse a definition string.

The parser can parse a definition string created by Vi-XFST, which is read from project file. Any addition to the operator set of Vi-XFST also requires modification to this method. So it is wise to understand the state diagram of the parser before any change in the operator set of Vi-XFST.

## 4.3 Bugs

Beside our best efforts to release a bug-free software with version 1.0, there are still some known problems, along with many undiscovered ones hiding in the code.

Abnormal termination of the Vi-XFST program leaves the XFST process as a zombie in the system. It has to be killed using system tools; ”kill” command on Unix or Task Manager in Microsoft Windows. Otherwise it will stay resident in the system memory and will not release system resources.

On the *Expression Canvas* a definition may be moved in two ways: using the scrollbars around the canvas or using the mouse to drag the operator base. The synchronization between position of the operator base and the scrollbars is not perfect at the moment. This causes some flickering while dragging the operator base. It does not cause any serious problem or any inconsistency and to be fixed in the next release.

A bug we encountered in XFST during our tests is that; on windows systems XFST ”`print directory`” command or alias ”`dir`” does not work properly. It gives the following message:

```
Copyright Xerox Corporation 1997-2003 Xerox Finite-State Tool,
version 8.1.3
Type "help" to list all commands available or "help help" for
further help.
xfst[0]: print directory
'ls' is not recognized as an internal or external command,
operable program or batch file.
```

Table 4.2: *"print directory"* command on XFST, gives an error for windows version.

It seems that, XFST uses the Unix "`ls`" command on the system to get the directory content, which is not available on Windows systems. Because of this bug in XFST, we had to exclude "`print directory`" command from our project.

# Chapter 5

# Development Environment

## 5.1 Introduction

The main component of the development was the Integrated Development Environment (IDE) tools. Most of the development was carried out using *gcc (version 3.3 20030226, prerelease)* on Linux systems using KDevelop from the KDE project. On Microsoft platforms, *Visual C++ Version 6.0* was the choice for compiler.

Beside compilers there is a library that has to be mentioned here. The QT library from *Trolltech Corparation* [1] was heavily used through out the Vi-XFST source code. The QT library will be discussed in more detail in the following sections.

The project design and intentions lead us to use many other auxiliary tools to support the development phase of the thesis. These are version control (*CVS*), source code beautifiers (*indent*), project source code documentation generators (*doxygen*) and *replace*, *replacehex* tools.

## 5.2 Target Operating Systems

### 5.2.1 Unix Environment

The first designs of the Vi-XFST have targeted for an application running on Unix platforms, in particular for Sun Solaris machines on the university campus. The close architectural similarities among Unix machines lead us to move the design and development environment to Linux operating systems without losing any features from the project. Since then, the primary project development environment system has been Linux. Today, all of the source code maintenance, functional tests and documentation are done on Linux machines. The choice of distribution is the SuSe Linux (which is at version 8.1 at the time of writing this document). On Linux systems, the project development is carried out on the *KDevelop IDE*.

[1] `http://www.trolltech.com`

### 5.2.2 Windows Environments

XFST program is also available on Microsoft Windows operating systems. So Vi-XFST should be, too. Thankfully QT library proved itself, as the best choice library for a multi-platform graphical C++ application. Within a few days, the Vi-XFST project was successfully ported to Microsoft Windows operating systems. Neither feature reduction nor any major changes in the source code was needed. Now more than 30,000 lines of Vi-XFST source code is the same for all platforms. The same code can be compiled and run on all the supported platforms. Development can also be done on any of these systems without any conflict. Occasionally, development of Vi-XFST is done on Microsoft Windows platforms using *Visual C++ 6.0*.

The only restriction is that QT library used in Vi-XFST is not freely available on Microsoft Windows platforms. *Trolltech* offers educational and academic licenses at a lower cost or for free to be used in non-profit applications. For more information about QT licensing, *Trolltech* should be contacted.

## 5.3 Dependencies and Auxiliary Tools

### 5.3.1 The QT Library

While designing our software, multiple operating system support was one of our principles. With this intention, after looking for a multi-platform graphical user interface application framework, eventually QT library has been chosen as the main graphical user interface library. As stated in QT Documentation [9] QT supports the following platforms:

- MS/Windows – 95, 98, NT 4.0, ME, 2000, and XP
- Unix/X11 – Linux, Sun Solaris, HP-UX, Compaq Tru64 UNIX, IBM AIX, SGI IRIX
- Macintosh – Mac OS X
- Embedded – Linux platforms with framebuffer support.

This list of supported platforms makes QT our best choice among user interface libraries. Using QT library, we have successfully created a software that runs on all platforms that XFST binary is available.

This library is freely available on Unix environments, but unfortunately the required version of QT is not available on windows systems for free. There are trial versions, or some low-priced academic licenses. For the availability of a license for a specific platform, more information may be found in Trolltech home page (`http://www.trolltech.com`)

### 5.3.2 KDevelop

KDevelop is an Integrated Development Environment for Unix Systems. The version we have used for our software development is 2.1 on SuSe Linux 8.1 distribution. Just like the Vi-XFST project, KDevelop wraps the functionalities of many difficult to manage command line Unix tools such as compiler and debuggers, adds many more features and offers a very powerful development environment to the open source community. It is one of the best open source IDE's available today. It is distributed within most of the Linux distributions and runs on Sun Solaris machines as well.

KDevelop also handles installation scripts like used like configure, Makefile etc. of the Vi-XFST project. The whole project can be compiled simply by:

```
./configure && make && make install
```

command sequence on most of the Linux systems. These scripts are auto-generated by KDevelop for every opened project.

### 5.3.3 Concurrent Versions System: CVS

> *"I can't imagine programming without it... that would be like parachuting without a parachute!"*
>
> *Brian Fitzpatrick on CVS*

CVS is used in this project mostly for record keeping purposes rather than collaboration of source between developers because the code is developed by a single person.

Record keeping became necessary when it was necessary to compare current state of the code with a version at some point in the past. For example, in the normal course of implementing a new feature, sometimes code is brought into a thoroughly broken state. Rolling back from this position to, or comparing changes with a stable state is quite simple using CVS. It is possible to get any version in time if the source code history is kept under CVS.

Also, instead of backing up a snapshot of the code, taking the whole CVS root directory, secures the whole development stages of the project.

The collaboration facility is one of the most "*magical*" features of CVS. CVS enables developers to edit the same source code on their local system and merge their changes in the repository. For example, suppose two developers edit the different parts of the same class file. When they commit their sources, CVS merges these changes into the same file and most of the time without any conflict.

The only place, where this collaboration facility is used is, when the code is developed on more than one computer: one on Linux and one on Windows operating systems, simultaneously. With the help of CVS, the code edited on any of these platforms may be sent to the CVS server, without loosing any control on the version of the active code on any of the systems.

### 5.3.4 Indent

Indent is a C++ source code beautifier for Unix. It has been used to keep the code indentation consistent throughout the 133 source files of the project.

### 5.3.5 Doxygen

Doxygen is a documentation system for C++, C, Java, IDL (Corba, Microsoft, and KDE-DCOP flavors) and to some extent PHP.

To create API documentation, some of which is appended at the end of this document as an appendix, Doxygen (Version: 1.2.17) toolkit is used. It is a very powerful tool to create documentation from project source codes. It can extract class references, dependency graphs, and document indexes in many document formats like HTML, LATEX, PostScript and more. The documentation is extracted from source codes, which requires some special formatting to have a better output. In a well-documented source code each class, method, variable, enumerator or type definition, has a definition comment block. Using Doxygen on this code will produce a better and usable documentation.

Vi-XFST source code is developed with this intention. All classes, majority of methods, and most of the important variables have formatted documentation blocks as comments.

```
/*!
Parses a given definition into graphical objects.
@param parent Parent of this class.  This pointer is used in chain delete
operations.
@param options Pointer to COptions object, where the all the application
settings are held.
param string String to be parsed.
@param baseactiongroup Pointer to the action group of definitions in a
pulldown menu.
@param definition Definition to be parsed into.
@param definitionList Pointer to the global list of definitions used in
this project.  The definitions
in this list must be used to reference any definition.
@return Returns the rectangle created for the parse.
*/
static CRectangle *parse(CShape * parent, const COptions * options, QWidget
* widget, const QString & string, QActionGroup * const baseactiongroup,
const CDefinition * definition, CDefinitionList * definitionList);
```

Table 5.1: A sample comment block specially formatted to produce Doxygen.

### 5.3.6 Replace and Replacehex

"*replace*" is a command line tool on Unix machines that searches and replaces a string though a list of input files. It comes with the *MySQL* source code package[10]. It is very useful for changing some variable names or keywords through the list of Vi-XFST source code files.

During this project, a tool, *"replacehex"* is also created to address some requirements that the previous replace command cannot handle. This new tool can replace any given hex code sequence with another hex code sequence. Therefore it is very flexible and powerful with the texts it can handle. For example sometimes it is necessary to replace Microsoft Windows end of line characters with the Unix end of line character, or to replace some sequence of tab characters with a list of space characters. The only difficulty is to find the hex values of search and replace strings. ”*man ascii*” gives the list of ASCII character set encoded in octal, decimal, and hexadecimal which can be referenced.

Replacehex project had started as an auxiliary tool to Vi-XFST project, then it is realized that it is unique on the Internet with these features. Now it is public with GNU license, and its source code is available on SourceForge servers at address *http://sourceforge.net/projects/replacehex/*.

```
replace -s '/*' '/*!'  -- $i.temp.2
replacehex -r $i -w /tmp/$i -s "0d" -p "0a"
```

Table 5.2: Usage of replace and replacehex commands in indent.sh

# Chapter 6

# Conclusions and Future Work

## 6.1 Conclusion

In this thesis a new finite-state project management model has been introduced. We have focused on creation of a powerful and extensible architecture that acts as an interface for Xerox Finite State Toolkit (XFST). We have also implemented a multi-platform software application using this model that facilitates XFST as a compiler in the background. Our software handles finite-state projects in a workspace and controls all activities between the user and the XFST application. By this encapsulation of the XFST process from the user, we have successfully integrated a user interface to control the finite-state development process. This control gave us the opportunity to introduce our management model.

With the proposed design, a textual file editing is replaced with a project-building concept similar to modern software development tools. The benefits of adopting an integrated development environment designed for finite-states include productivity gains and advantages due to substantial reduced time to debug and management costs of a research project. The visual supporting features of Vi-XFST, enable viewing complex networks at different levels of detail and make even large projects manageable and comprehensible. Regular expression building is carried out on the workspace canvas with mouse clicks or drag and drops. Network building, testing, modification and many more commands of XFST prompt are now associated with toolbar buttons or many easy to use pop-up menus.

Also, many manual tasks are now carried out automatically. For example, when a definition is modified by the user, any other definitions that depend on the modified one are recursively recompiled. This feature is a great enhancement in finite-state project development since it was a very complex task for users to carry out manually.

New controls are introduced to the finite-state calculations such as restricting the definition names to be unique on the stack. Another control is to prevent un-defining networks that have dependents on the stack to reduce ambiguities and increase understandability of the network structure by the researchers.

## 6.2 Future Work

Our intention in this first version of management model and software implementation is to illustrate our approach for finite-state development process. Therefore, in our application (Vi-XFST), only the most commonly used regular expression operators have been supported in visual regular expression development interface. Some more complicated commands and features of XFST have been excluded. In future extensions to the model and the software project, these excluded features of XFST may be implemented.

Vi-XFST cannot handle project files edited manually. This ability to handle external source files requires a complete XFST source file parser. Also, Vi-XFST enforces some syntactic restrictions on regular expressions; such as every literal should be defined in a definition before it is used in a visual regular expression component. Therefore, a project import feature should include algorithms to convert a manually edited source file into an acceptable format.

This version of Vi-XFST supports only one workspace on its main window. Working with multiple workspaces will greatly enhance the project handling capabilities of the development environment. The user shall be able switch between projects, compare their finite-state networks and reuse each other's components in one main window.

XFST supports working on multiple source or networks files. A source file can be included in another file. This feature is important to manage very large project files, since it allows splitting long input files. In an actual development environment for finite-state projects, it should be possible to work on many files in a single project workspace.

A final extension that is being planned is to add a capability of defining regular expression operators at runtime to Vi-XFST operator set. As there are many formats of commands and regular expression operators in XFST, and always new ones are being added to this list, it may be a good approach to include this flexibility in our application.

# Bibliography


[1] Lauri Karttunen, Tamas Gaal, Andre Kempe, Xerox Finite-State Tool, Xerox Corporation, 1997.

[2] Xerox Corporation, Syntax and Semantics of Regular Expressions, http://www.xrce.xerox.com

[3] Xerox Corporation, Examples of Networks and Regular Expressions, http://www.xrce.xerox.com

[4] Lauri Karttunen, Application of Finite-State Transducers in Natural-Language Processing, Xerox Research Center Europe, Meylan, France.

[5] AT&T Labs-Research, FSM Library, http://www.research.att.com/sw/tools/fsm/

[6] Gertjan van Noord, FSA Utilities: A Toolbox to Manipulate Finite-state Automata, 1998

[7] Scot Meyers, More Effective C++, Addison Wesley.

[8] Karl Fogel, Open Source Development With CVS, The Coriolis Group

[9] QT Documentation, http://www.trolltech.com

[10] MySQL, http://www.mysql.com

# Chapter 7

# Appendix - Statistics About the Code

Number of lines of total code: **39067**

```
akdere@:/infinity20> cat *.h *.cpp *.ui| wc -l
39067
```

Number of lines of handwritten code: **30438**

```
akdere@:/infinity20> cat c*.h c*.cpp main.cpp | wc -l
30438
```

Number of lines of computer generated code: **8629**

```
akdere@:/infinity20> cat f*.h *.ui| wc -l
8629
```

Number of source code files: **133**

```
akdere@:/infinity20> ls *.h *.cpp *.ui | wc -l
133
```

Number of characters: **872856**

```
akdere@:/infinity20> cat *.h *.cpp | wc -c
872856
```

Compilation time (on Celeron 800MHz, 128MB Ram, Suse Linux 8.1 with default conf.): **16 Min 31 Sec**

```
akdere@:/infinity20> date && make > /dev/null && date
Sat Jun 21 22:01:21 EEST 2003
Sat Jun 21 22:17:52 EEST 2003
```

Number of method declarations: **1075**

```
akdere@:/infinity20> cat c*.h | grep ");" | wc -l
1075
```

Number of ”/*” and ”//” sequences, the command blocks: **2927**

```
akdere@:/infinity20> cat c*.h c*.cpp main.cpp | grep -E "(/\*)|(//)"
| wc -l
```

```
    2927
```

Number of debug messages: **548**

```
    akdere@:/infinity20> cat c*.h c*.cpp main.cpp | grep -E "(qDebug)|
(qWarning)|(qFatal)" | wc -l
    548
```

Number of pointer checks: **356**

```
    akdere@:/infinity20> cat c*.h c*.cpp main.cpp | grep -E "Q_CHECK_PTR"
| wc -l
    356
```

Number of asserts: **34**

```
    akdere@:/infinity20> cat c*.h c*.cpp main.cpp | grep -E "ASSERT" |
wc -l
    34
```

# Chapter 8

# Appendix - Vi-XFST User Guide

## 8.1 Introduction

This is a user guide documentation for Vi-XFST project. This documentation introduces the software implementation of the XFST management model prepared by Prof. Kemal Oflazer[1] and Yasin Yilmaz[2] in Sabanci University. Please contact the designers of the project for the other related documents on the proposed model.

Intention of Vi-XFST project is to wrap the functionalities of XFST and to provide graphical editing, management and testing features to the researchers. XFST is a very powerful tool to manipulate finite-state networks, with its large set of commands. Vi-XFST is designed to make these commands invisible to the user and serve their functionalities through graphical components of its interface.

## 8.2 Installation and Requirements

Vi-XFST code can be compiled on any UNIX or Microsoft Windows platforms where QT library (version >= 3.0.0) is available. The installation process may vary according to the platform, but it is quite straight forward if these given steps are followed.

A script is prepared that uses QT tool *qmake* to compile and install the source code. This is useful where *autoconf* and *automake* tools are not installed on the target platform. The only requirement is the QT library. Just type:

```
# sh ./compile.qmake.sh
```

to start compilation, in the source directory. The default prefix value points to the users home director. Please adjust it prior to compilation by editing the "*DESTDIR*" variable in the "*project*"

[1] oflazer@sabanciuniv.edu
[2] yasin@uekae.tubitak.gov.tr

file in the "*infinity20/infinity20*" directory. The output binary file is named "*vixfst*" and installed in the prefix directory with the necessary auxiliary files, such as documentation (in *$prefix/docs*) and images (in *$prefix/images*).

The other way to build the code on Unix systems is mostly for development purposes. The original code is organized using *KDevelop IDE* and project source contains *autoconf* and *automake* compatible compilation and installation Makefiles. The *compile.autoconf.sh* script is prepared to automate this installation procedure, which generates a binary with *debug options on*. The output file name is "***infinity20***", which is the code name of the project during development phase. The binary is created in *infinity20/infinity20* directory. It is strongly recommended to use this compilation if you intent to debug your code.

Just gunzip and un-tar the source packet and enter the *infinity20* directory, type:

```
# sh ./compile.autoconf.sh
```

to start compilation. The installation prefix is *"/usr/local/"* by default. To install binaries type *"make install"* after compilation successfully ends. There are other options for the auto-generated "*configure*" script created during the compilation. These options may require advanced knowledge of GNU *autoconf* and *automake* scripts; therefore it is mostly used for development purposes only.

If you wish to debug or scroll through the classes and see the structure of the project code, you should load the project file for *KDevelop* "*infinity.kdeproj*" in *infinity20* directory. *KDevelop* also handles the projects with the above *autoconf* generated makefiles.

On Microsoft Windows platforms, Vi-XFST project file under *"Infinity20_Win32"* directory, should be loaded using *Microsoft Visual C++ (version >= 6.0)*. Once the compilation is done, the image and flush_network file should be in the same directory as the output executable file.

As it is mentioned above, the only requirement of Vi-XFST source code is the QT library. There are freeware versions for Unix systems. For Microsoft Windows platforms a freeware license is not available. But it is possible to obtain academic licenses for educational purposes at lower prices. For more information see Section 5.3.1.

Our project code is not tested yet on other Unix operating systems (like FreeBSD, OpenBSD or HP Unix). But the design and implementation of the project avoids depending on any system specific libraries, calls or functions that may reduce the portability of the code. So there should not be any trouble porting Vi-XFST code on other UNIX platforms.

Of course, to build your own projects with Vi-XFST, XFST executable should be available in the target platform. For more information about availability of XFST binaries for your platform, please refer to http://www.xrce.xerox.com

## 8.3 The Integrated Development Environment

When you start Vi-XFST, you are immediately placed within the integrated development environment. This main window provides all the tools you need to design, compile and test your finite-state networks.

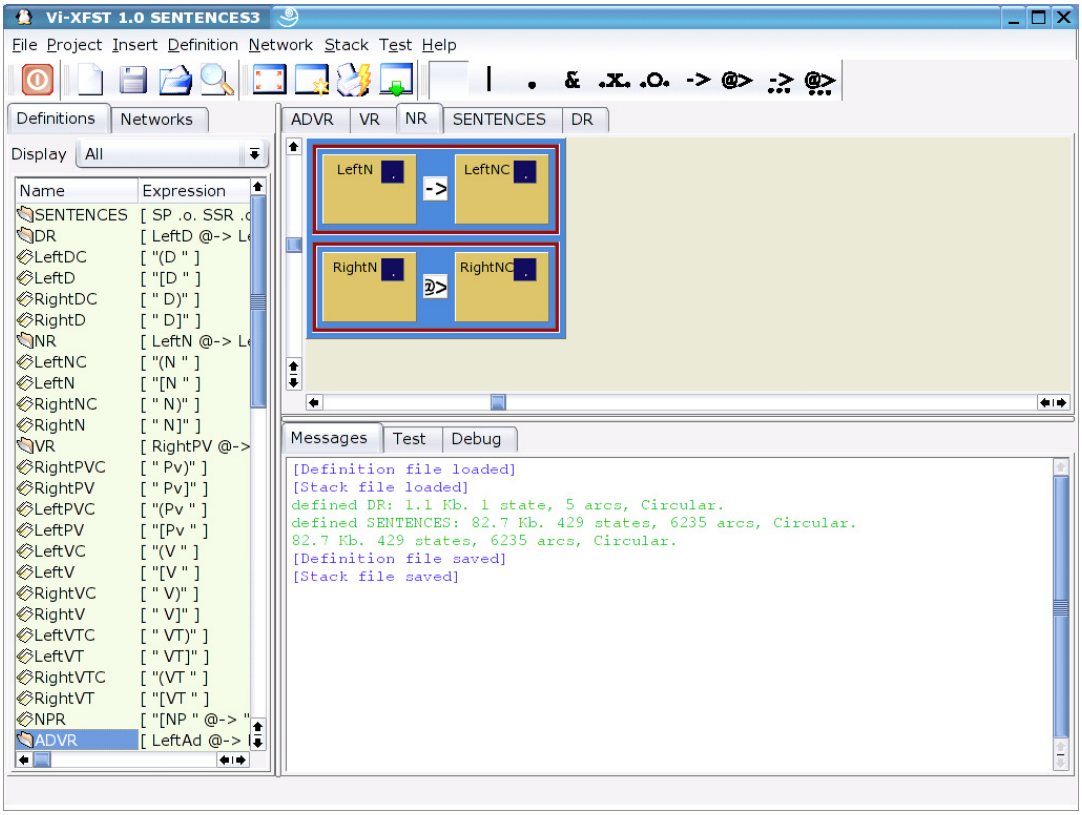


Figure 8.1: A sample screen-shot from the Vi-XFST IDE.

The development environment is composed of many graphical functional components; such as menu items, tabs, dialog boxes, which are used during different steps of finite-state project development. In the following sections these components are discussed in detail.

### 8.3.1 Main Window

Vi-XFST development environment main window is activated when the program is started. This window is the main control panel of the development process. For better understanding, main window can be detailed into the following components; *Definition Browser, Network Browser, Expression Canvas, Messages Tab, Test Tab Debug Tab and Menubar* components, each of which will be explained in more detail:

### 8.3.2 Definition Browser

The defined symbols in XFST that are constructed by "`define`" command are referred as regular expression definitions in Vi-XFST. Any regular expression created on Vi-XFST is added into the *Definition Browser*. It will remain there until it is undefined. *Definition Browser* is the main component to access a regular expression definition in Vi-XFST.

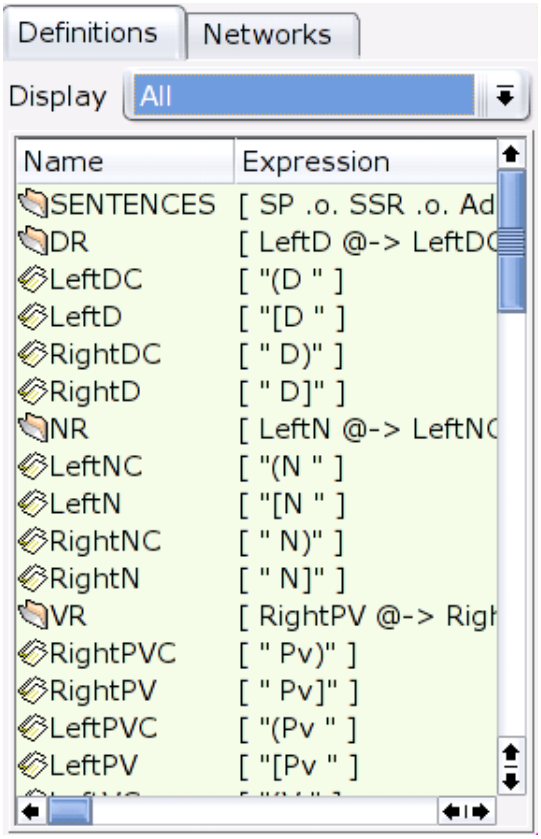


Figure 8.2: The Definition Browser

There are many functions available in the pop-up menu of *Definition Browser*. Select a definition item and right click to invoke the *Definition Browser* pop-up. The available functions are:

**New Definition** Invokes the *Definition Option* dialog to create a new definition. See Section 8.3.10 for more information.

**Properties** Displays the properties of a definition in *Definition Option* dialog. You can modify the definition properties in this dialog. See Section 8.3.10 for more information.

**Read regex** Creates the network for the definition and pushed it onto the stack. The *Network Browser* and *Test Tab* are activated after a successful compilation. The network appears in the *Network Browser* and becomes the top network. You can test it using the *Test Tab*.

**Undefine** Un-binds the symbol and removes the definition from the *Definition Browser*.

The items in the *Definition Browser* can be dragged and dropped into empty slots of an operator base on the *Expression Canvas*. See Section 8.4.2 for an example usage of drag and drop functions in Vi-XFST.

### 8.3.3 Network Browser

Every network on the XFST stack is listed in the *Network Browser*. The top network of the stack is the top item in the browser. Once a network is created, it is added to this browser where it will remain until it is popped up. *Network Browser* is the main component to access a network in Vi-XFST.

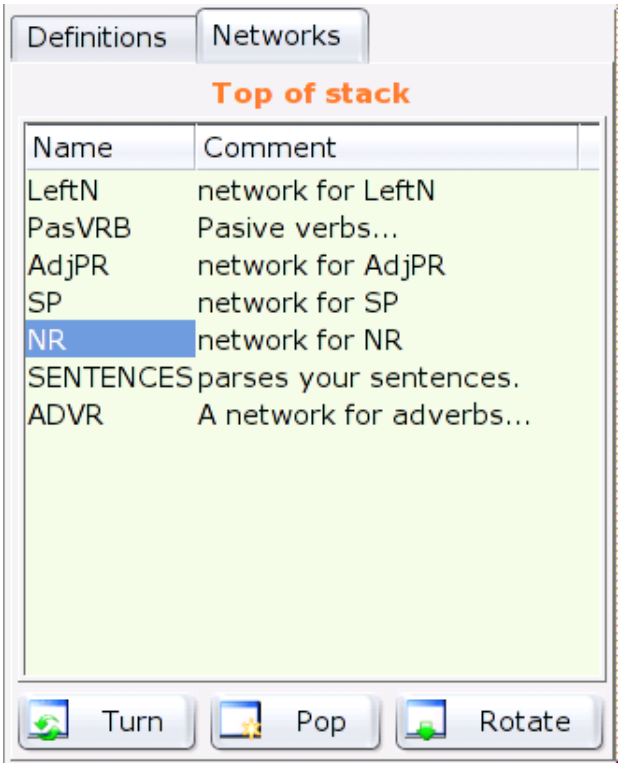


Figure 8.3: The Network Browser

To access the properties of a network, select and right click on it in the *Network Browser*. The *Network Options* dialog will be invoked in which various network settings can be adjusted. There are also other operations on the stack under **Stack** menu, which are *turn, rotate, pop, clear* and *print stack*. These operations modifies the stack in the XFST process, then Vi-XFST synchronizes its *Network Browser* with the XFST stack at the same time.

### 8.3.4 Expression Canvas and Workspace Tabs

*Expression Canvas* is where the visual regular expression development can be done when a project is active on Vi-XFST main window. There can be more than one *Expression Canvas* but only one of them is active at a time. These canvases are held in the *Workspace Tabs*. The user can switch between these tabs to activate the desired canvas and edit the regular expression inside it.

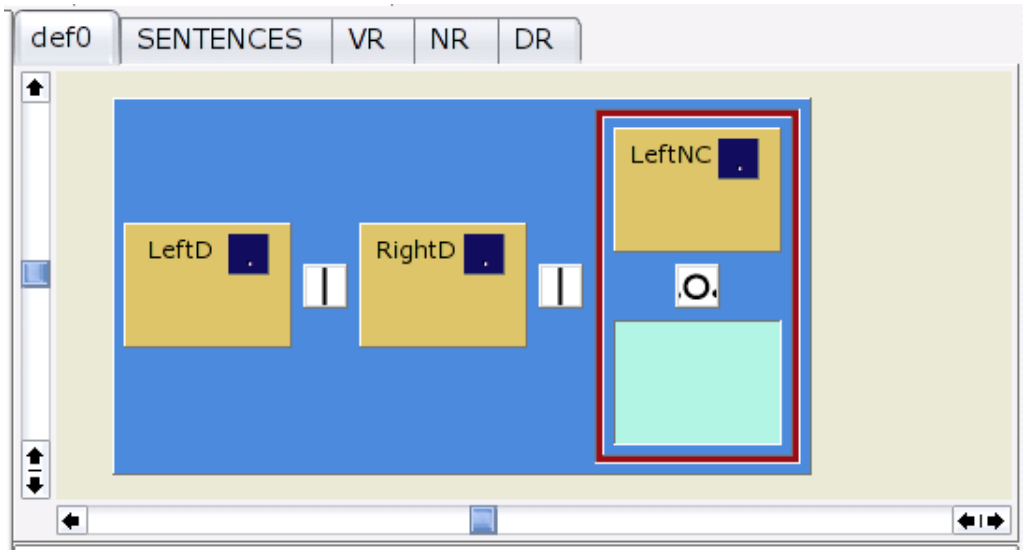


Figure 8.4: The Expression Canvas and Workspace Tabs

Each *Expression Canvas* can contain only one regular expression object. When this object is removed the associated canvas and workspace tab is also closed. To start a new regular expression in a new page, just select your operator from the tool bar and click on any *Expression Canvas*; Vi-XFST will open an empty workspace tab and canvas for you.

### 8.3.5 Message Tab

Vi-XFST also handles XFST messages on behalf of the user. They are filtered and displayed in the *Message Tab*.

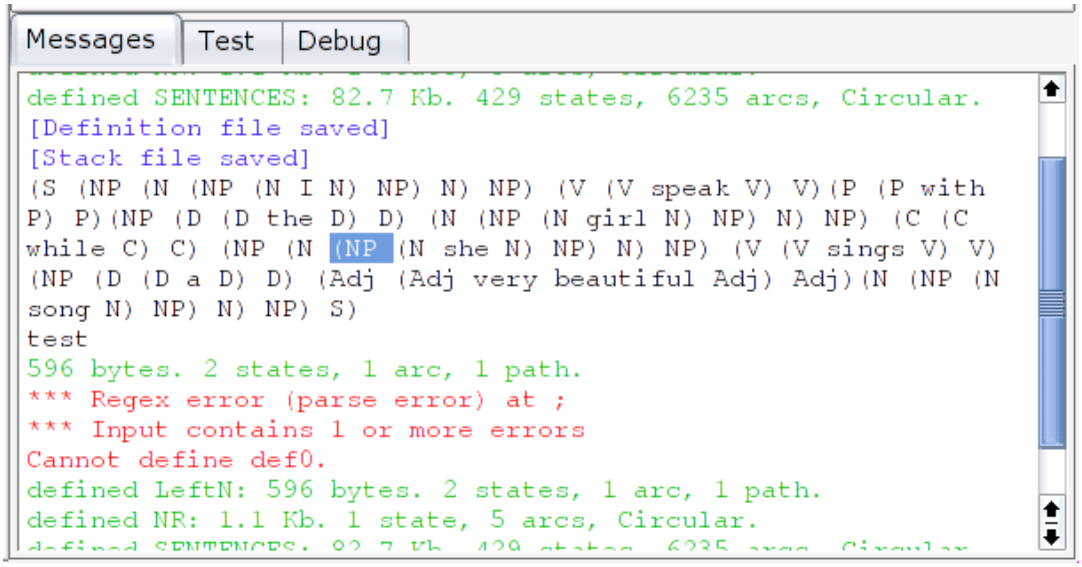


Figure 8.5: The Message Tab

The messages displayed in this tab can be grouped in to three:

**XFST success messages** They display successful command execution messages, such as a successful regular expression or network compilation. They are printed in green color.

**XFST error messages** They are printed in red, and inform error events. Most of the time, to take the attention of the user, a pop-up information dialog box may be invoked about the error.

**Print Messages** These are the outputs of *"print"* commands of XFST, such as *"print defined, print stack, print random-lower etc"*. They are displayed in blue.

**Test Results** Test result messages are generated by the "*apply*" commands. They are displayed in both *Test Tab* and *Message Tab*. These messages are printed in black fonts.

**Vi-XFST Messages** These messages are not XFST generated. Some of the actions of Vi-XFST produces these outputs, such as loading a project, saving or loading the stack file etc. These messages are in dark blue.

### 8.3.6 Test Tab

*Test Tab* is where you can apply strings to your networks on the stack.

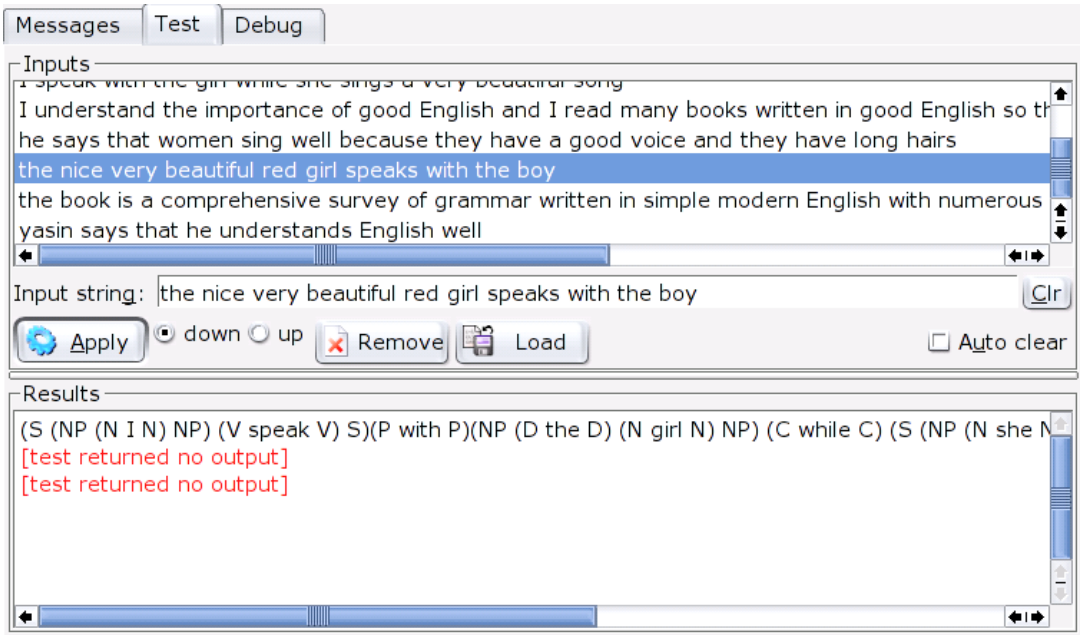


Figure 8.6: The Test Tab

When a network is compiled onto stack, Vi-XFST will automatically activate *Test Tab* and place the cursor in the *Input String Edit Box*. Enter your inputs into the *Input String Edit Box*, and press enter or click the *Apply* button just below. The direction of apply command can be set by *down* and *up* radio buttons near the *Apply* button. The results will be displayed in the *Results Edit Box*, and your input string will be added to the *Inputs* list. You can adjust maximum number of input strings kept in the Inputs list and some other settings about *Test Tab* in the *Preferences* dialog.

Items in the Inputs lists can be removed, cleared all, loaded from, or saved to a text file by the buttons on this tab. These actions are also associated with menu items in the *Test* menu.

### 8.3.7 Debug Tab

The Vi-XFST project is still under development. Inevitably, despite our hard efforts on debugging the code, there may still be software bugs or logic errors. Therefore the *debugging window*, which is heavily used in the development process, is not removed from this release version. This window, when the message handler is installed and debug option is enabled during compilation, displays various debug messages from Vi-XFST execution flow.

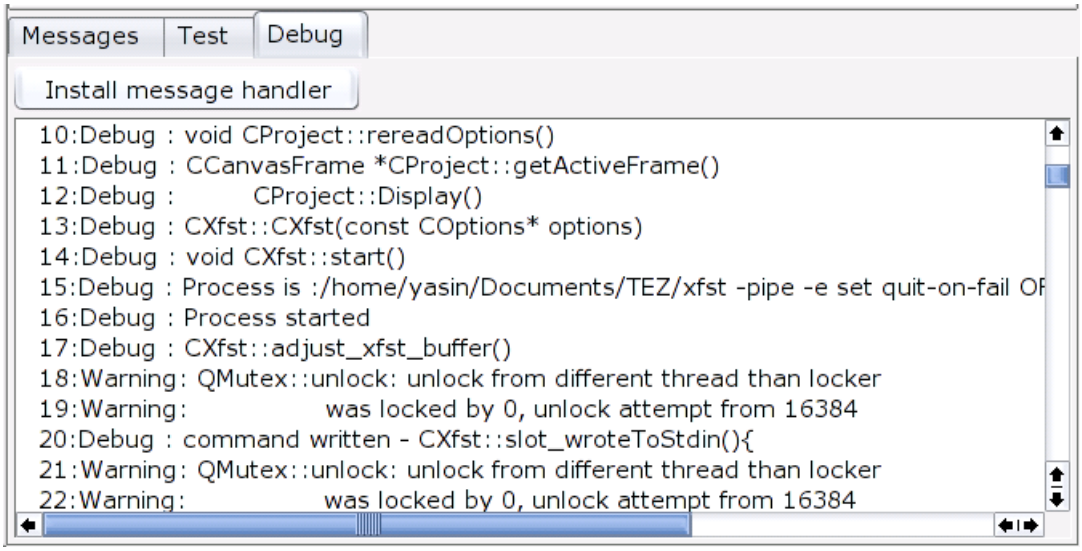


Figure 8.7: The debugging window is useful only when the debug option is set during compilation.

The user can include the outputs of this window to report bugs to the project developer that will be used to track down the cause of the problem.

### 8.3.8 Menubar Commands

**File Menu**

There are functions for editing Vi-XFST preferences and shutting down Vi-XFST, under file menu item.

**Preferences** Opens the *Preferences* dialog. See Section 8.3.12 on page 86 for more information.

**Exit (Ctrl-Q)** Initiates the shutdown sequence of Vi-XFST, which can be canceled if the user does not confirm the invoked confirmation dialog box. If there are modifications in the active project, the user will be prompted to save the changes. The Vi-XFST will close all open files, try to close and then try to kill XFST process, and save all Vi-XFST options and history to the operating system settings structure. For more information about Vi-XFST Settings refer to Section **??**.

**Project Menu**

**New (Ctrl+N)** Opens the *New Project* dialog, allowing to create a new project workspace and associated project file.

**Save (Ctrl+S)** Saves the active project file and associated binary definitions and network files to the project directory.

**Load (Ctrl+L)** Opens the *Load Project* dialog, allowing the user to select an existing project file to load into Vi-XFST.

**Close (Ctrl+W)** Closes the active project.

**View & print project file (Ctrl+P)** Opens the *Project Preview* dialog, where the project source file can be viewed, saved or printed in various formats.

**Open recent project** Contains a sub-menu with the last 10 opened projects. The user can open a project more easily using this recent project menu.

**Save xfst messages** Saves the *Message Tab* content to a text file. Invokes a *Save File* dialog.

**Clear xfst messages** Clears the contents of *Message Tab.*

**Properties (F2)** Opens the *Project Options* dialog that lets the user change various settings for the project. This could be a project name to save the project with another name or some user interface options.

**Insert Menu**

**Release selection (F2)** Click this menu option to de-select any selected insert toolbar button. This stops the pointer to insert new operators on the *Expression Canvas*.

**Concatenation** Click this menu option and then click on the *Expression Canvas* or an open slot of an expression to place a Concatenation operator.

**Crossproduct** Click this menu option and then click on the *Expression Canvas* or an open slot of an expression to place a Crossproduct operator.

**Intersection** Click this menu option and then click on the *Expression Canvas* or an open slot of an expression to place an Intersection operator.

**Longest match replacement** Click this menu option and then click on the *Expression Canvas* or an open slot of an expression to place a Longest Match Replacement operator.

**Longest match markup** Click this menu option and then click on the *Expression Canvas* or an open slot of an expression to place a Longest Match Markup operator.

**Replacement** Click this menu option and then click on the *Expression Canvas* or an open slot of an expression to place a Replacement operator.

**Simple markup** Click this menu option and then click on the *Expression Canvas* or an open slot of an expression to place a Simple markup operator.

**Composition** Click this menu option and then click on the *Expression Canvas* or an open slot of an expression to place a Composition operator.

**Crossproduct** Click this menu option and then click on the *Expression Canvas* or an open slot of an expression to place a Crossproduct operator.

**Definition Menu**

**New definition (F4)** Invokes the *Definition Options* dialog to get definition properties and then pushes a new definition into the stack with these values. The created definition will be added to the *Definition Browser*.

**Push (F5)** Defines the active definition on the *Expression Canvas* and adds it to the *Definition Browser*.

**Read regex (F7)** Compiles the active definition on the *Expression Canvas* adds it to the *Network Browser*.

**Undefine (F8)** Undefines the active definition on the *Expression Canvas* and removes it from the *Definition Browser*.

**Properties (F9)** Opens the *Definition Options* dialog box for the active definition on the *Expression Canvas*. Definition name, expression and comment are some of the editable values of a definition through this dialog box.

**Save print outputs to file** This is a toggle-menu item. If it is selected, the print command outputs will be written to a file instead of the *Message Tab*. This option invokes a *Save File* dialog.

**Do not print to file** This is a toggle-menu item. If it is selected, the print command outputs will be written to the *Message Tab* instead of a file.

**Print-Defined** prints each defined symbol and the size of the network it stands for.

**Network Menu**

**Save print outputs to file** This is a toggle-menu item. If it is selected, the print command outputs will be written to a file instead of the *Message Tab*.

**Print first N outputs** This is a toggle-menu item. If it is selected, the print command outputs will be written to the *Message Tab* instead of a file.

**Print Random-lower** Generates random words from the lower side of top network on the stack.

**Print Random-upper** Generates random words from the upper side of top network on the stack.

**Print Words** Prints the paths in the top network of the stack.

**Print Lower Words** Displays the words in the lower side language of the top network on the stack.

**Print Upper Words** Displays the words in the upper side language of the top network on the stack.

**Print Net** Prints a text representation of the top network on the stack.

**Print Sigma** Prints the sigma alphabet of the top network on the stack.

**Print Random Words** Generates random words from the top network on the stack.

**Print Size** Prints the size of the top network on the stack.

**Tests Equivalent** Returns 1 if the topmost two networks contain the same language.

**Tests Lower-Bounded** Returns 1 if the lower side of the top network has no epsilon cycles.

**Tests Lower-Universal** Returns 1 if the lower side of the top-level network represents the universal language.

**Tests Overlap** Returns 1 if the languages of the two topmost networks have a non-empty intersection.

**Tests Sublanguage** Returns 1 if the language of the topmost network is a sublanguage of the second network on the stack.

**Tests Upper-Bounded** Returns 1 if the upper side of the top level network contains no epsilon cycles.

**Tests Upper-Universal** Returns 1 if the upper side of the top-level network contains the universal language.

**Prune** Removes all paths leading to non-final states in the top network on the stack. This operation is not included in the project file.

**Reverse** Replaces the top network on the stack with the one that accepts the reverse language. This operation is not included in the project file.

**Sigma** Replaces the top network on the stack with a network that encodes the "sigma language" of the original network, that is, the union of all symbols in the sigma alphabet. This operation is not included in the project file.

**Sort** Reorders the arcs of the top network on the stack in a canonical order. This operation is not included in the project file.

**Substring** Replaces the top network on the stack with a network that accepts every string that is a substring of some string in the language of the original network. This operation is not included in the project file.

**Optimize** Runs a heuristic algorithm that tries to reduce the number of arcs. This operation is not included in the project file.

**Unoptimize** Reverses the effect of *optimize* command. This operation is not included in the project file.

**Complete** Extends the top network until it has a transition for every symbol in sigma in every state. This operation is not included in the project file.

**Determinize** Replaces the top network with an equivalent deterministic network. This operation is not included in the project file.

**Epsilon-remove** Replaces the top network with an equivalent one that has no epsilon transition. This operation is not included in the project file.

**Invert** Exchanges the two sides of the top network on the stack.

**Lower side** Extracts the lower language of the top network on the stack.

**Minimize** Replaces the top network with an equivalent one that has minimal number of states.

**Negate** Replaces the top network with a network that accepts all and only those strings rejected by the original.

**Stack Menu**

**Clear** Removes all networks on the stack.

**Pop** Removes the top network on the stack.

**Print** Displays the content of the stack.

**Rotate** Pushes the top element of the stack to the bottom.

**Turn** Reverses the order of the networks on the stack.

**Test Menu**

**Clear outputs** Clears the outputs generated by previous tests.

**Save outputs** Saves the output messages in the *Test Messages Window* into a text file. This option invokes a Save File dialog box.

**Apply up** Simulates the composition of the input string with the lower side of the top network on the stack and extracts the result from the upper side. Any output message is displayed in the *Test Messages Window*.

**Apply down** Simulates the composition of the input string with the upper side of the top network on the stack and extracts the result from the lower side. Any output message is displayed in the *Test Messages Window*.

**Load text file as inputs** Invokes a File Open dialog box, and loads the selected ASCII file as a list of input strings into the *Test Inputs Listbox*.

**Save test inputs** Saves the test inputs in the *Test Inputs Listbox* into a text file. This option invokes a Save File dialog box.

**Clear test inputs** Clears the entries in the *Test Inputs Listbox*.

**Help Menu**

**About Vi-XFST** Invokes the About Vi-XFST dialog, which gives the author contact information, version number, and some licensing information about Vi-XFST.

### 8.3.9 Project Options Dialog

Click **Project**|**Properties** or **Project**|**New** to invoke the *Project Options* dialog.

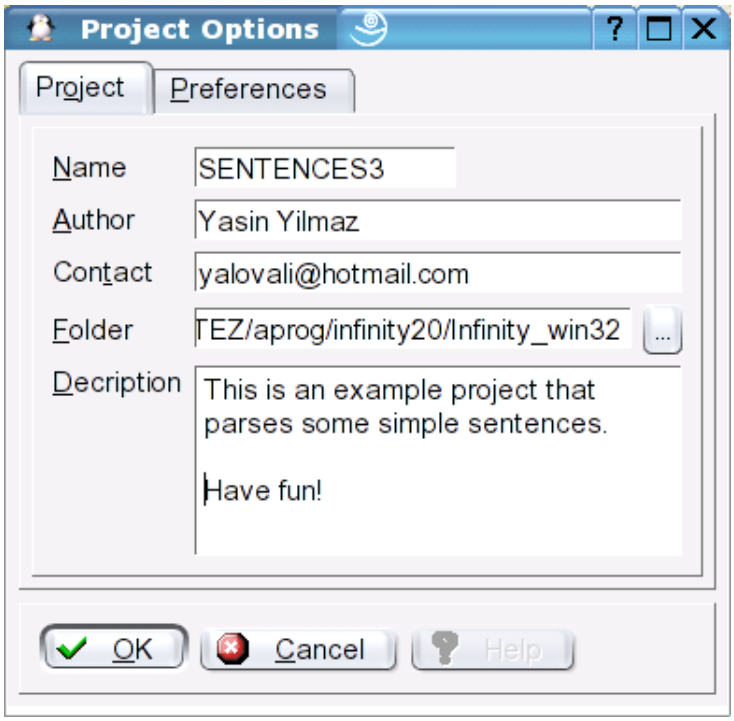


Figure 8.8: Project Options dialog

This dialog is used to edit properties a project session in Vi-XFST. The values that can be set in this dialog are:

**Project Name** This the descriptive name for the active project. It is also used to generate the filename of the source file with the extension "**.infproj**". Changing the project name, functions as a ”*save as*” operation. The next save command will create all project files according to this new name.

**Author** The author of the project file.

**Contact** The contact information for the project, probably an email address or a web site.

**Folder** The folder in which the project files will be saved. Default value is the current directory.

**Description** Intention of this field is a short description for the project purpose, structure or any other useful information to the others.

**Canvas Color** This option sets the *Expression Canvas* background color for this project.

### 8.3.10 Definition Options Dialog

Click **Definition|Properties** or **Definition|New** to invoke the *Definition Options* dialog.

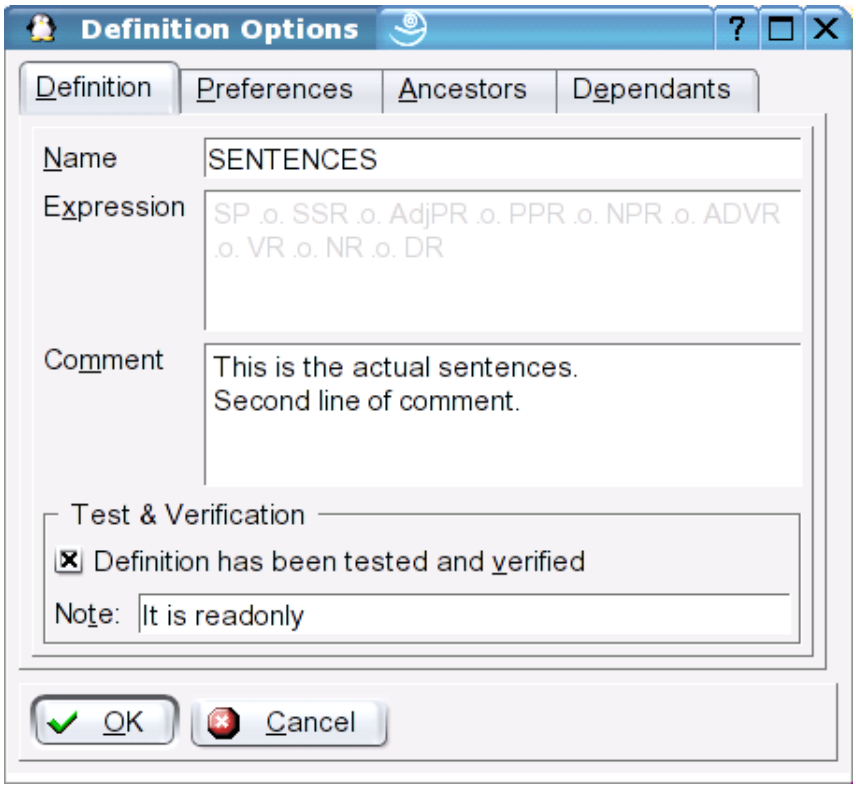


Figure 8.9: Definition Options dialog

This dialog is used to edit properties of an existing or a newly created definition. The values that can be set in this dialog are:

**Name** This is the name of the definition. If it is a new definition Vi-XFST will generate a random definition name for the user. In Vi-XFST definition names are kept unique and overriding a definition name is not allowed. If a definition has dependents, changing its name is also not allowed.

**Expression** It is the regular expression that this definition is composed of. Definitions in Vi-XFST can be created in two ways; either by typing an expression in this *Definition Options* dialog, or by using graphical components on the *Expression Canvas*. The first kind of definitions are marked as *non-visual*, they cannot be displayed on the *Expression Canvas* graphically. Their expressions can only be edited in this dialog. But the second type of definitions are called visual definitions, and their expression is extracted from their graphical representation on the canvas. These definitions can only be edited on the *Expression Canvas*. Therefore their expressions are read-only on in this dialog.

**Comment** Any comment on the definition can be typed in here.

**Canvas Color** This option sets the *definition rectangle* background color for this definition only.

**Tested & Verified** A checkbox to indicate that this definition is verified and tested, and possibly bug-free.

**Verify Note** A short note on verification of definition.

**Ancestors** A tree view of the ancestors of this definition. The root node is this definition and items below are the ones, which it depends on.

**Dependents** A tree view of definitions that depends on this definition. The root node is this definition and items below are the ones that depend on it.

### 8.3.11 Network Options Dialog

Click **Properties** menu item of the pop-up menu in the Network Browser to invoke the *Network Options* dialog.

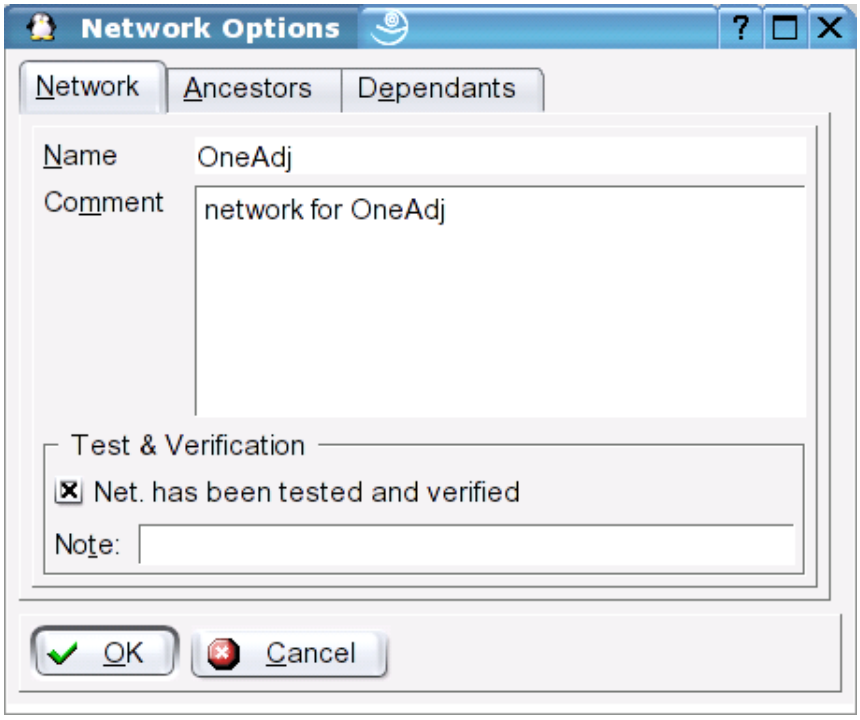


Figure 8.10: Network Options dialog

This dialog is used to edit and view properties of a network on the stack. The values that can be set in this dialog are:

**Name** This is the name of the network, which is the same as the definition that this network is compiled from. This value is read only.

**Comment** Any comment on the network can be typed in here.

**Tested & Verified** A checkbox to indicate that this network is verified and tested, and possibly bug-free.

**Verify Note** A short note on verification of this network.

**Ancestors** A tree view of the ancestors of this network, which is the same as ancestors of the definition of the network. The root node is this network and items below are the ones, which this network depends on.

**dependents** A tree view of definitions that depends on the definition of the network. The root node is the network and items below are the ones those depend on it.

### 8.3.12 Preferences Dialog

Click **File|Preferences** to invoke the *Preferences* dialog. The following dialog will appear, to let various Vi-XFST settings to be changed.

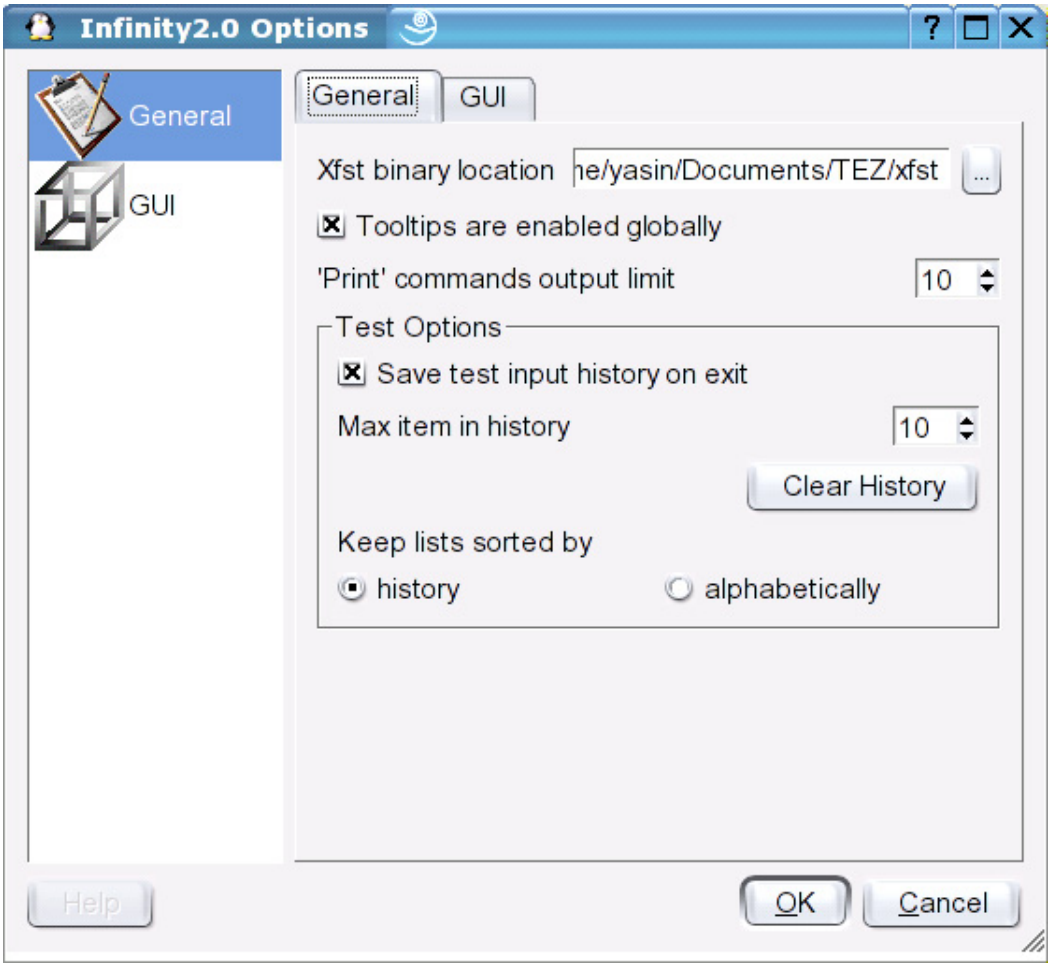


Figure 8.11: Preferences dialog

The setting in this dialog box are saved in *"$HOME/.qt/infinityrc"* file on UNIX systems. On Microsoft Windows operating systems it is in the registry database under *'infinityrc'* key. For the first execution of Vi-XFST, some default values will be set to these options. The most important of them that has to be reset by the user is the XFST (or XFST.exe on Microsoft Windows systems) binary location. If this location is not entered or invalid, Vi-XFST will not be able to load XFST process.

The options that can be set in this dialog are:

**XFST Binary Location** This is the location of the executable XFST binary file on the system. User should have proper access rights to execute this file.

**Enable Tooltips** Enables/disables the tooltips available for many components on Vi-XFST. These are little help messages displayed in a small yellow box below the mouse cursor, that appears when the user points to a menu item, definition box on the canvas or toolbar items.

**Print Commands Output Limit** Sets a limit on the upper limit of output lines generated by print commands available in *definition* and *network* menus.

**Save Test Inputs On Exit** Enables/disables automatic saving of test inputs list into your project file.

**Max Number of Inputs** Puts an upper limit on the number of test inputs that will be kept in your project file.

**Clear History** This button clears the test input list.

**Keep Inputs Sorted by History|Alphabetically** Sorts the input list according to the given criteria.

**Definition Canvas Color** This option sets the *definition rectangle* background color for this definition.

**Project Canvas Color** This option sets the *Expression Canvas* background color for this project.

**Font Options** These are the font settings used in the canvas of the Vi-XFST. They can be changed to whatever settings are available on the underlying operating system.

### 8.3.13 Project Preview Dialog

Click **Project|View & Print** menu to invoke the *Project Preview* dialog. The following dialog will display the source code of your project.

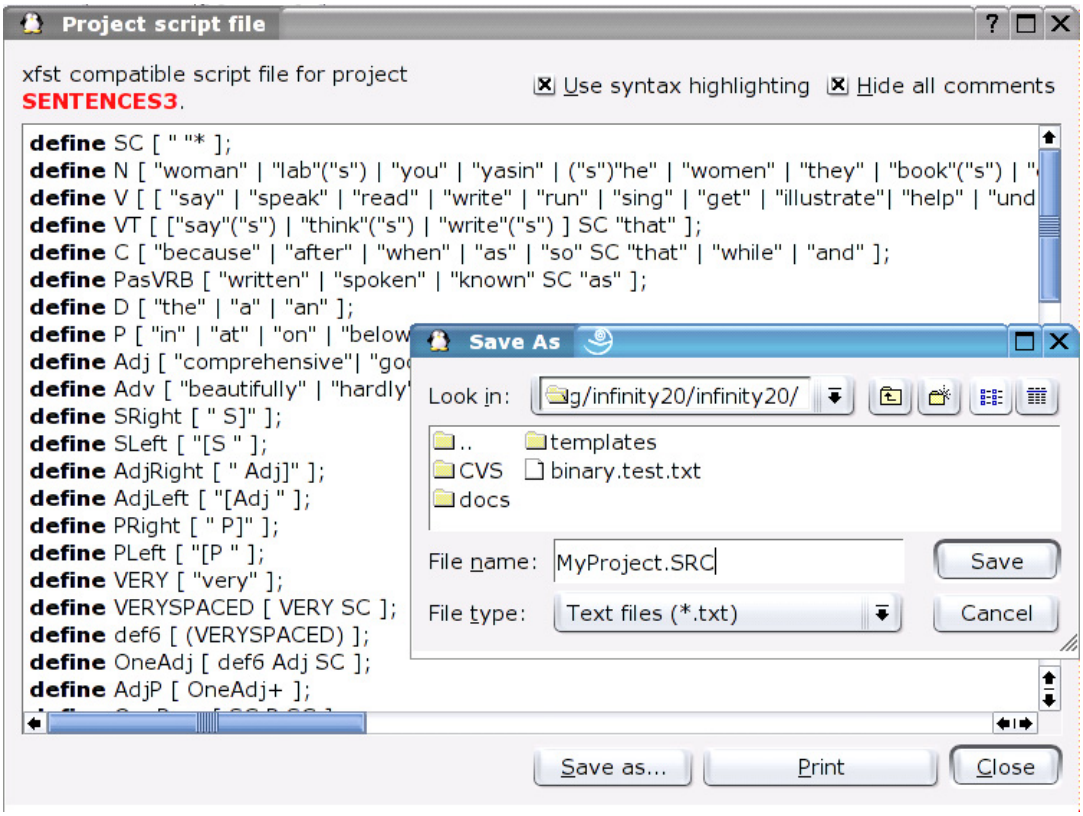


Figure 8.12: Project Preview dialog

You can use this dialog to export the project to a text file or print in various formats. Click *Hide all comments* checkbox to hide/unhide Vi-XFST in-line control comments. Or you can enable/disable syntax highlighting by the *Use syntax highlighting* checkbox. The **Print** button will call the system print dialog box and let you choose printing preferences and get a hardcopy of your project. If your underlying system permits, a postscript copy can also be generated with this printing dialog.

The **Save** button lets you export a copy of the project into a text file, according to the display criteria set in this dialog.

## 8.4 Project Development Process

### 8.4.1 Starting a New Project

Vi-XFST handles a development session with XFST in a *Project Workspace*. For each project workspace, a project file is created. Also when the project is saved, a binary definition and/or network file will be created in the same directory. Definition and network settings, regular expressions, test inputs and user comments are kept in the project file which has a name created by concatenation of project name and "*.infproj*" file extension.

To start a project workspace, click **Project|New** menu item, or the associated tool button. The *Project Options* dialog will be invoked. Enter a descriptive name for your project, and select a project directory. The default value points to the current directory, but it is probable that you will want your project files saved in a more reasonable location.

You can also enter some values for *Author, Contact information* and for *project description*. These are optional fields and you can change these values at any time later, just select **Project|Properties** menu to bring this dialog back.

When you click the OK button, the *Project Options* dialog will be closed and a new project workspace will be initiated. A XFST process will be loaded and menu items, browsers and workspace canvas will be initialized. After the initializations, you can start adding definitions to your workspace.

### 8.4.2 Building Regular Expressions

There are two ways to define a regular expression definition in Vi-XFST. The quickest way is to type in a regular expression using the keyboard. Simply click **Definition|New Definition** menu item or associated keyboard shortcut (F4) to open *Definition Options* dialog.

In the dialog a definition name is already generated for you. Just type a regular expression, -some comment is optional but recommended- and click **OK**. The *XFST Progress Dialog* will

appear and try to define your expression. If no error occurs, your regular expression definition will appear in the *Definition Browser*. The *Message Tab* will be popped up if it is not visible. Check these messages for your definition. If there has been an error, Vi-XFST would have noticed that and display the error message generated by XFST.

The other way to create your definition, is to use the *Expression Canvas* to build your regular expression in a more controlled and user friendly way. Vi-XFST offers a powerful visual interface for regular expression construction.

Select an *operator base* type from the toolbar and click on the *Expression Canvas*. Vi-XFST will draw the operator base with empty slots. These slots are where you will insert existing regular expression definitions. You can select and drag a definition from the *Definition Browser*. Or write click the empty slot and select **Insert Definition** menu item and select an existing definition from the list. The selected definition will be added into this empty slot you have right-clicked.

You can double click an empty slot to create a new definition more quickly. *Definition Options* dialog will be invoked and the created definition will be inserted into this slot automatically.

For more information about working with graphical representation of regular expression see Section 8.5.

### 8.4.3 Compiling a Regular Expression

Once a regular expression is defined in XFST and added to the *Definition Browser*, now it can be compiled as a network onto the stack. There are many ways of compiling a regular expression, you can just right click a definition in the *Definition Browser*, and select **read regex** in the invoked pop-up menu. Or if your regular expression definition is on the expression canvas, right click the definition there and select **read regex** menu item in the pop-up. There is also another menu item to do same task under the *Definition* menu.

During the regular expression definition compilation, *XFST Progress* dialog will be displayed. If there is no error, your network item will appear in the *Network Browser*. The *Test Window* will be popped up if it is not visible. If there has been an error, Vi-XFST would have noticed that display the *Message Window* instead of the test window.

Compiling a regular expression is just one mouse click as described above, and you can quickly start entering inputs to the network. Now, please proceed to the next section.

### 8.4.4 Testing a Network

To apply input strings on the transducer at the top of the stack, switch to the *Test Tab* if it is not already activated. Enter your inputs into the *Input String* edit box, and press enter or click the *Apply* button just below. The direction of apply command can be set by *down* and *up* radio buttons near the *Apply* button. The results will be displayed in the *Results* editbox, and your input string will be added to the *Inputs* list.

Items in the inputs lists can be removed, cleared and loaded from or saved to a text file. These operations are available both through the buttons on the *Test Tab* and menu items under the **Test** menu. If auto-save option is set in the Vi-XFST settings, input strings are kept inside the source file when the active project is saved. They are also loaded when the project is opened back.

### 8.4.5 Modifying The Stack

Items on the stack are the networks compiled by Vi-XFST. You can remove (pop-up), change position and ordering (turn, rotate) of these items with buttons below the *Network Browser*. There are various operations over a network on the stack. Most of them are available under the **Network** menu. For this version of Vi-XFST these network modification commands are not kept in your project file. But the binary network file saved by "`save stack`" command, will contain your most recent stack including these modifications. So keep in mind this issue and beware that Vi-XFST will not re-run these modification commands when it recompiles the networks on the stack.

### 8.4.6 Printing and Viewing the Source Code

The project source file can be viewed within the *Project Preview* dialog. Click **Project|View & Print** menu to invoke the dialog that will display the source code of your project.

You can use this dialog to export the project to a text file or print in various formats. Click *Hide all comments* checkbox to hide/unhide Vi-XFST inline control comments. Or you can enable/disable syntax highlighting by the *Use syntax highlighting* checkbox. The **Print** button will call the system print dialog box and let you choose the printing preferences and get a hardcopy of your project. If your underlying system permits, a postscript copy can also be generated from this printing dialog.

Click the **Save** button to export a copy of the project into a text file, according to the display criteria set in this dialog.

### 8.4.7 Exporting the Code and Binary Files

Under the project directory (see *Project Options* dialog), there are three files related to a project. These are:

**<ProjectName>.infproj** The source file for your project. It contains project information, options, network definitions and input strings. This file can be loaded into XFST with "-l" parameter. All the Vi-XFST generated codes are marked with "##Vi-XFST##" comment markers. But it is strongly advised not to edit this file manually. Instead, use the *Project View & Print* dialog described in Section 8.3.13 to generate a user copy of the project source file.

**<ProjectName>.infdef** This binary file is created by the XFST "`save defined <filename>`" command automatically by Vi-XFST whenever the active project is saved. The binary file contains networks for all defined symbols in the project workspace. You can use this file in XFST with "`load defined <filename>`" command. Vi-XFST will try to locate this file when the project is loaded, but if it is not available, all definitions will be rebuild from the regular expression source file. But it cannot detect if this file is modified outside Vi-XFST, therefore you should not change the content of this file. You must work on your own copy of this binary file.

**<ProjectName>.infstack** This binary file is created by the XFST "`save stack <filename>`" command automatically by Vi-XFST whenever the active project is saved. The binary file contains networks on the stack of the project workspace. You can use this file in XFST with "`load stack <filename>`" command. Vi-XFST will try to locate this file when the project is loaded, but if it is not available, all networks will be rebuild from the source file. But it cannot detect if this file is modified outside Vi-XFST, therefore you should not change the content of this file. You must work on your own copy of this binary file.

Any modification on the stack will be effective in this binary file. So if you want to prepare a binary transducer file to distribute without the source code, you can freely do any modification with the operators in Network menu. But remember that these modifications are not saved into project source file.

All of the files listed above, are compatible with XFST program. Any of them can be distributed to other users. But only the project file (with extension *.infproj*) can be loaded back to Vi-XFST.

If the project file seems confusing with many inline comment blocks put by Vi-XFST, you can get a tidier file by *Project Preview* dialog as described above.

### 8.4.8 Bug Reporting and Debugging Vi-XFST

The Vi-XFST project is still under development. It lacks many features of a comprehensive integrated development environment. Inevitably, despite our hard efforts on debugging the

code, there may still be bugs, logic errors, or even crashes while using Vi-XFST. Therefore the debugging window, which is heavily used in the development process, is not removed from this release version. This window, when the message handler is installed and debugging enabled during compilation, displays various debug messages from Vi-XFST execution flow.

If you experience a bug to report to the authors, please send a copy of these messages that will let to track down the bug. We appreciate every bit of help to improve the code.

## 8.5 Graphical Representation Of a Regular Expression

One of the most powerful features of Vi-XFST is the graphical representation of regular expressions. On the *Expression Canvas*, it is possible to build complex regular expression with simple mouse clicks.

When a project is opened on Vi-XFST, there is always an active workspace tab that contains an empty expression canvas. You can use this canvas to place and construct expressions on. First step is to select an operator from the toolbar that will be the main operator base. Then operands should be added to slots of this base. Each slot on Vi-XFST's operator bases is a like a pair of brackets (`"[" ... "]"`) in regular expression text. Vi-XFST places your definitions in these boxes. Figure 8.13 shows an example definition and the corresponding expression.

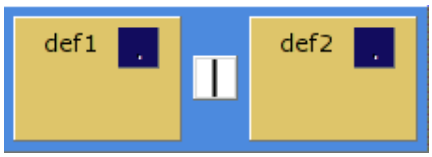


Figure 8.13: A definition base with two open slots: `[ def1 | def1 ]`

Operator bases like union, concatenation or composition may take more than default number of operands. To add additional slots select **New Slot** from the pop-up menu of the base.

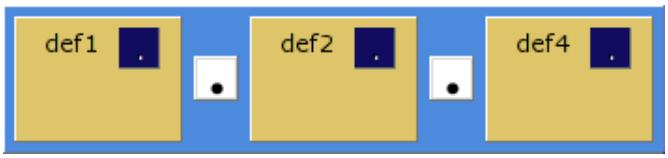


Figure 8.14: A definition base with two open slots: `[ def0 def1 def4 ]`

Just the same way an empty slot may be removed. Right click an empty slot and select **Remove** from the invoked pop-up menu to remove it from the operator base.

There are three ways to insert a definition into an empty slot. If you want to create and add a new definition, just double click into an empty slot. Vi-XFST will create a definition for you and pop-up the *Definition Options* dialog. Freely enter any expression you like, there is no restriction in regular expressions for definitions created using this dialog. Click **OK** to accept your changes. Vi-XFST will automatically push your expression into XFST, add it to the *Definition Browser* and replace the empty slot with this definition. This is a very fast way to build up complex expressions.

Another way is to use an existing definition from the *Definition Browser.* Select it with mouse, drag and drop it into an empty slot. This is also another comfortable way of building expression within Vi-XFST.

The last way is to select the definition to be inserted, from the pop-up menu of the empty slot under **Insert Definition** item. Sub-menu of this item is the list of definition available in Vi-XFST. The selected definition will be inserted into the empty slot.

It is also possible to insert an operator base into an empty slot of another base. This feature enables to build complex regular expressions. It is important to remember that each slot is a pair bracket. Therefore your new operator base is enclosed within a pair of brackets as show in Figure 8.15.

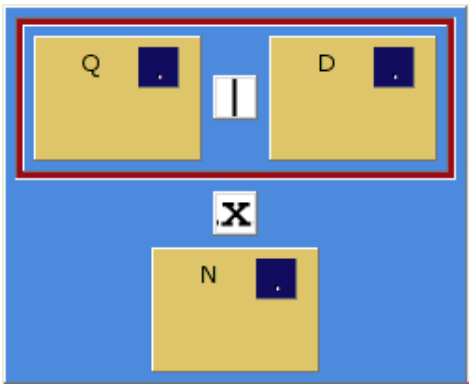


Figure 8.15: Nesting operator bases in each other: `[ [ Q | D ] .x. N ]`

There is no limit in the depth of nesting operators inside each other. You can freely build large regular expressions. If things get confusing on the canvas, try the **Minimize** option in the pop-up menus of operators to shrink the operator you are working on.

Once you are done with your regular expression, it has to be defined in XFST process. Select **Push definition** in the pop-up menu of the operator base. Vi-XFST will define and store it in the *Definition Browser*. For more information about defining a definition see Section 8.4.2.

### 8.5.1 Operator Base Object

The basic component of an expression is the base rectangle that defines a pair of brackets (¨[¨ ... ¨]¨) and the selected operator. You add other components (definitions) onto this base.

You can give it a name that will be used to refer to this network once it is compiled. When a base is added to the workspace canvas a unique definition name is already generated for you. Also the name of a base can be modified from the *Definition Dialog* that is invoked with the Properties item inside pop-up menu of the base. When this definition is compiled, it will appear in the definitions browser of Vi-XFST with this new name. The following drawing is a simple definition object:


Figure 8.16: A sample operator base

In Figure 8.13 the main operator is an intersection. This base has two operands; `def1` and `PRICE`, which are also previously defined definitions. These definitions can be removed from the base by selecting **Remove** operator from the pull-down menu invoked by right clicking on them.

The symbols on the upper right corner of these definition rectangles denote if the definition can be enlarged inside the operator base. A ”`.`” means that this definition was not built using the visual expression canvas; therefore it cannot be enlarged using graphical components on the screen. Also double clicking on the definition rectangle only opens this definition’s properties.

The ”`x`” sign on the upper right corner of a definition, as for the ”`PRICE`” in Figure 8.13, means that the definition can be enlarged into its components. When a definition is enlarged by clicking on this icon, this symbol changes into an ”`O`”. Clicking again in this symbol shrinks the definition back to its original state. Here is an example of viewing a definition in enlarged form:

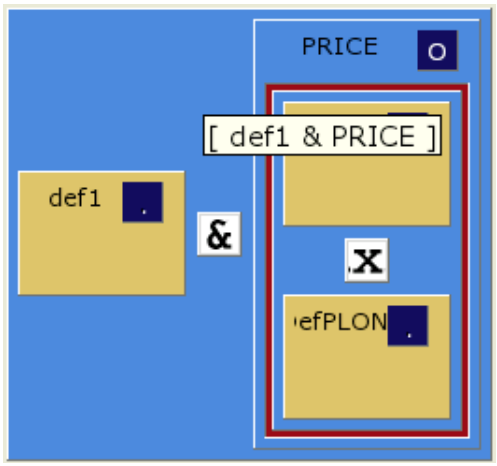


Figure 8.17: The PRICE definition is enlarged inside another definition.

By using this feature of Vi-XFST, it is possible to view a transducer back into its sub-components. It is also possible to compile an enlarged definition, and apply input string to this network on the stack to debug it.

## 8.6 Expression Arithmetic

In this version of Vi-XFST, only most commonly used regular expression operators are supported on the expression canvas. Regular expressions that require the other operators can be still built using the *Definition Options* dialog that is invoked by **Definition**|**New** menu. We hope to release support for these excluded operators in the next version.

Once an operation is defined you can still change it, replace operands, or delete it later. It is possible to add new slots to an operator if it can take more than default number of operands. For example the default Union operator comes with two empty slots. But you can always add new slots for additional operands. Also some operators, such as markup and replacement, have "*conditional*" operator bases that have special meanings for them. You can add a conditional base to them from the pull down menu by right clicking on these operators. All of these features are accessible through the pop-up menus of the operator slots.

The following sections in this chapter are a list of available operators in Vi-XFST.

### 8.6.1 Union

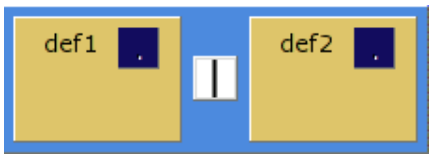


Figure 8.18: `Union` operator base. Displayed regular expression:
`[ def1 | def2 ]`

Opens with 2 default open slots and more slots can be added. Operator icon can be changed into: Concatenation, Intersection.

### 8.6.2 Concatenation

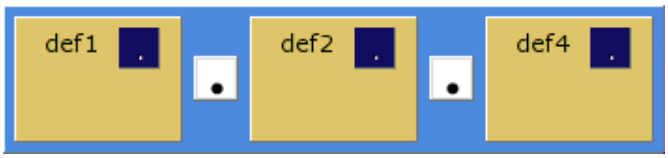


Figure 8.19: `Concatenation` operator base. Displayed regular expression:
`[ def1 def2 def4 ]`

Opens with 2 default open slots and more slots can be added. Operator icon can be changed into: Union, Intersection.

### 8.6.3 Intersection

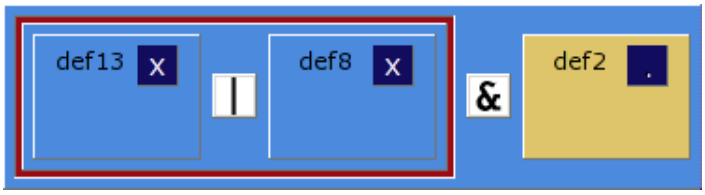


Figure 8.20: `Intersection` operator base. Displayed regular expression:
`[ [def13 | def8 ] & def2 ]`

Opens with 2 default open slots and more slots can be added. Operator icon can be changed into: Union, Concatenation.

### 8.6.4 Composition

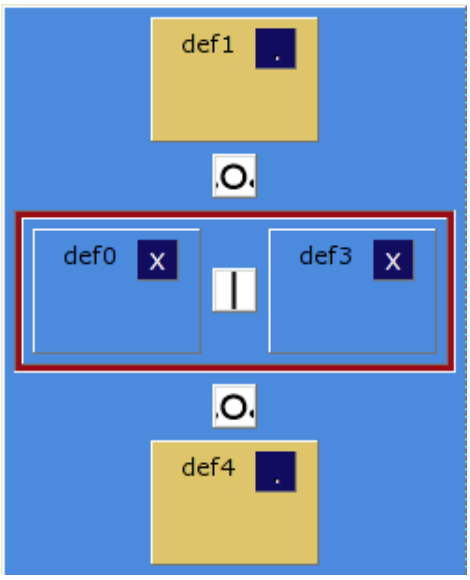


Figure 8.21: `Composition` operator base. Displayed regular expression:
`[ def1 .o. [ def0 | def3 ] .o. def4 ]`

Opens with 2 default open slots and more slots can be added. Operator icon cannot be changed into another operator.

### 8.6.5 Crossproduct

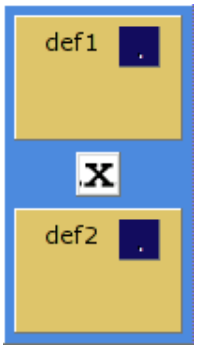


Figure 8.22: `Crossproduct` operator base. Displayed regular expression:
`[ def1 .x. def2 ]`

Opens with 2 default open slots and no more slots can be added. Operator icon cannot be changed into another operator.

### 8.6.6 Replacement

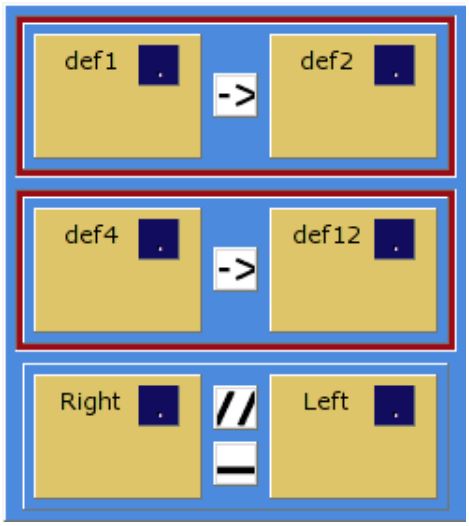


Figure 8.23: `Replacement` operator base. Displayed regular expression:
`[[ def1 -> def2 ],[def4 -> def12 ] // Right _ Left ]`

Opens with 2 default open slots. Possible to add another pair of slots for parallel replacement. Operator icon can be changed into: Longest-match Replacement.

**Special option:** *New Condition* adds a condition with two open slots to the replacement operation. Only one condition can be defined per operator base.

### 8.6.7 Left-to-right, Longest-Match Replacement

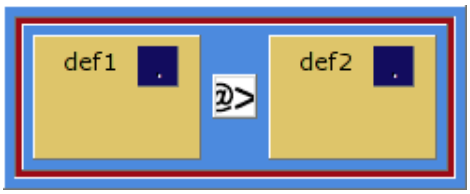


Figure 8.24: `Left-to-right,Longest Match Replacement` operator base. Displayed regular expression:
`[ def1 @-> def2 ]`

Opens with 2 default open slots. Possible to add another pair of slots for parallel replacement. Operator icon can be changed into: Replacement.

**Special option:** *New Condition* adds a condition with two open slots to the replacement operation. Only one condition can be defined per operator base.

### 8.6.8 Simple Markup

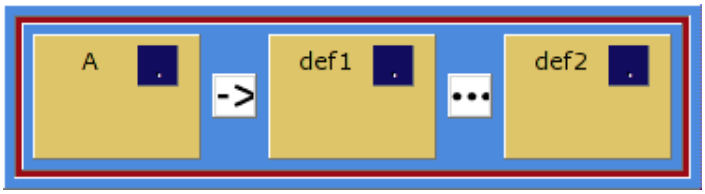


Figure 8.25: `Markup` operator base. Displayed regular expression:
`[ A -> def1 ... def2 ]`

Opens with 3 default open slots. Possible to add another triple of slots for parallel markup. Operator icon can be changed into: Longest-match markup.

**Special option:** *New Condition* adds a condition with two open slots to the markup operation. Only one condition can be defined per operator base.

### 8.6.9 Left-to-right, Longest-match Markup

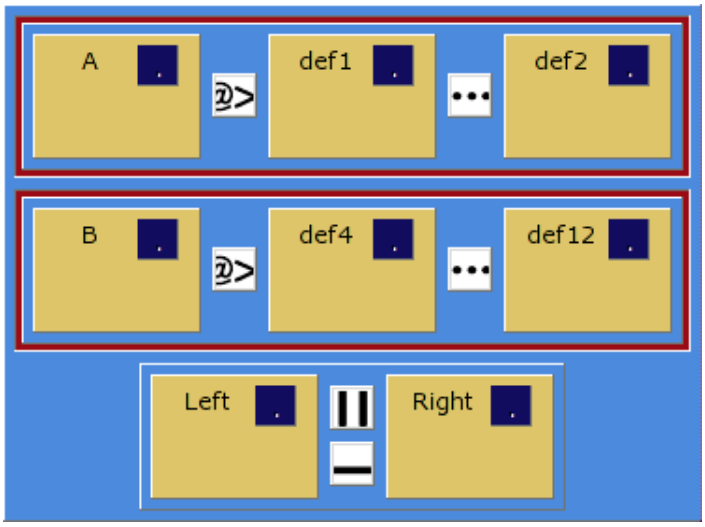


Figure 8.26: `Left-to-right, Longest-match Markup` operator base. Displayed regular expression:
`[[A @-> def1 ... def2],[B @-> def4 ... def12] || Left_Right ]`

Opens with 3 default open slots. Possible to add another triple of slots for parallel markup. Operator icon can be changed into: Simple markup.

**Special option:** *New Condition* adds a condition with two open slots to the markup operation. Only one condition can be defined per operator base.

## 8.7 Bug Reporting

The Vi-XFST project is still under development. Inevitably, despite our hard efforts on debugging the code, there may still be bugs. Therefore the debugging window, which is heavily used in the development process, is not removed from this release version. This window, when the message handler is installed and debugging enabled during compilation, displays various debug messages from Vi-XFST execution flow.

If you experience a bug to report to the authors, please send a copy of these messages that will let us track down the bug. We appreciate every bit of help to improve the code.

## 8.8 Authors

**Project Supervisor:**

Prof. Kemal Oflazer <koflaz@sabanciuniv.edu>

**Main Developers:**

Yasin Yilmaz <yalovali@hotmail.com>

# Bibliography


[1] Lauri Karttunen, Tamas Gaal, Andre Kempe, Xerox Finite-State Tool, Xerox Corporation, 1997.

[2] Xerox Corporation, Syntax and Semantics of Regular Expressions, http://www.xrce.xerox.com

[3] Xerox Corporation, Examples of Networks and Regular Expressions, http://www.xrce.xerox.com

[4] Lauri Karttunen, Application of Finite-State Transducers in Natural-Language Processing, Xerox Research Center Europe, Meylan, France.

[5] AT&T Labs-Research, FSM Library, http://www.research.att.com/sw/tools/fsm/

[6] Gertjan van Noord, FSA Utilities: A Toolbox to Manipulate Finite-state Automata, 1998

[7] Scot Meyers, More Effective C++, Addison Wesley.

[8] Karl Fogel, Open Source Development With CVS, The Coriolis Group

[9] QT Documentation, http://www.trolltech.com

[10] MySQL, http://www.mysql.com