Proceedings of the Workshop of the ACL Special Interest Group on Computational Morphology (SIGMORPHON)
Association for Computations Linguistics
Barcelona, July 2004

Vi-xfst: A Visual Regular Expression Development Environment for Xerox Finite State Tool

Kemal Oflazer and Yasin Yılmaz
Human Language and Speech Technology Laboratory
Sabancı University
Istanbul, Turkey
oflazer@sabanciuniv.edu, yyilmaz@su.sabanciuniv.edu

Abstract

This paper describes *Vi-xfst*, a visual interface and a development environment, for developing finite state language processing applications using the Xerox Finite State Tool, *xfst*. *Vi-xfst* lets a user construct complex regular expressions via a drag-and-drop visual interface, treating simpler regular expressions as "Lego Blocks." It also enables the visualization of the structure of the regular expression components, providing a bird's eye view of the overall system, enabling a user to easily understand and track the structural and functional relationships among the components involved. Since the structure of a large regular expression (built in terms of other regular expressions) is now transparent, users can also interact with regular expressions at any level of detail, easily navigating among them for testing. *Vi-xfst* also keeps track of dependencies among the regular expressions at a very fine-grained level. So when a certain regular expression is modified as a result of testing, only the dependent regular expressions are recompiled resulting in an improvement in development process time, by avoiding file level recompiles which usually causes redundant regular expression compilations.

1 Introduction

Finite state machines are widely used in many language processing applications to implement components such as tokenizers, morphological analyzers/generators, shallow parsers, etc. Large scale finite state language processing systems built using tools such as the Xerox Finite State Tool (Karttunen et al., 1996; Karttunen et al., 1997; Beesley and Karttunen, 2003), van Noord's Prolog-based tool (van Noord, 1997), the AT&T weighted finite state machine suite (Mohri et al., 1998) or the INTEX System (Silberztein, 2000), involve tens or hundreds of regular expressions which are compiled into finite state transducers that are interpreted by the underlying run-time engines of the (respective) tools.

Developing such large scale finite state systems is currently done without much of a support for the "software engineering" aspects. Regular expressions are constructed manually by the developer with a text-editor and then compiled, and the resulting transducers are tested. Any modifications have to be done afterwards on the same text file(s) and the whole project has to be recompiled many times in a development cycle. Visualization, an important aid in understanding and managing the complexity of any large scale system, is limited to displaying the finite state machine graph (e.g., Gansner and North (1999), or the visualization functionality in INTEX (Silberztein, 2000)). However, such visualization (sort of akin to visualizing the machine code of a program written in a high-level language) may not be very helpful, as developers rarely, and possibly never, think of such large systems in terms of states and transitions. The relationship between the regular expressions and the finite state machines they are compiled into are opaque except for the simplest of regular expressions. Further, the size of the resulting machines, in terms of states and transitions, is *very large*, usually in the thousands to hundreds of thousands states, if not more, making such visualization meaningless. On the other hand, it may prove quite useful to visualize the structural components of a set of regular expressions and how they are put together, much in the spirit of visualizing the relationships amongst the data objects and/or modules in a large program. However such visualization and other maintenance operations for large finite state projects spanning over many files, depend on tracking the structural relationships and dependencies among the regular expressions, which may prove hard or inconvenient when text-editors are the only development tool.

This paper presents a visual interface and development environment, *Vi-xfst* (Yılmaz, 2003), for the Xerox Finite State Tool, *xfst*, one of the most sophisticated tools for constructing finite state language processing applications (Karttunen et al., 1997).

Vi-xfst enables incremental construction of complex regular expressions via a drag-and-drop interface, treating simpler regular expressions as "Lego Blocks". *Vi-xfst* also enables the visualization of the structure of the regular expression components, so that the developer can have a bird's eye view of the overall system, easily understanding and tracking the relationships among the components involved. Since the structure of a large regular expression (built in terms of other regular expressions) is now transparent, the developer can interact with regular expressions at any level of detail, easily navigating among them for testing and debugging. *Vi-xfst* also keeps track of the dependencies among the regular expressions at a very fine-grained level. So, when a certain regular expression is modified as a result of testing or debugging, only the dependent regular expressions are recompiled. This results is an improvement in development time, by avoiding file level recompiles which usually causes substantial redundant regular expression compilations.

In the following sections, after a short overview of the Xerox *xfst* finite state machine development environment, we describe salient features of *Vi-xfst* through some simple examples.

2 Overview of *xfst*

xfst is a sophisticated command-line-oriented interface developed by Xerox Research Centre Europe, for building large finite state transducers for language processing applications. Users of *xfst* employ a high-level regular expression language which provides an extensive palette of high-level operators.[1] Such regular expressions are then compiled into finite state transducers and interpreted by a run-time engine built into the tool. *xfst* also provides a further set of commands for combining, testing and inspecting the finite state transducers produced by the regular expression compiler. Transducers may be loaded onto a stack maintained by the system, and the topmost transducer on the stack is available for testing or any further operations. Transducers can also be saved to files which can later be reused or used by other programs in the Xerox finite state suite.

Although *xfst* provides quite useful debugging facilities for testing finite state networks, it does not provide additional functionality beyond the command-line interface to alleviate the complexity of developing large scale projects. Building a large scale finite state transducer-based application such as a morphological analyzer or a shallow finite state parser, consisting of tens to hundreds of regular expressions, *is also a large software engineering undertaking.* Large finite state projects can utilize the *make* functionality in Linux/Unix/cygwin environments, by manually entering (file level) dependencies between regular expressions tered into a *makefile*. The *make* program then invokes the compiler at the shell level on the relevant files by tracking the modification times of files. Since whole files are recompiled at a time even when a very small change is made, there may be redundant recompilations that may increase the development time.

3 *Vi-xfst* – a visual interface to *xfst*

As a development environment, *Vi-xfst* has two important features that improve the development process of complex large scale finite state projects with *xfst*.

1. It enables the construction of regular expressions by combining previously defined regular expressions via a drag-and-drop interface.
2. As regular expressions are built by combining other regular expressions, *Vi-xfst* keeps track of the topological structure of the regular expression – how component regular expressions relate to each other. It derives and maintains the dependency relationships of a regular expression to its components, and via transitive closure, to the components they depend on. This structure and dependency relations can then be used to visualize a regular expression at various levels of detail, and also be used in very fine-grained recompilations when some regular expressions are modified.

3.1 Using *Vi-xfst*

In this section, we describe important features *Vi-xfst* through some examples.[2] The first example is for a simple date parser described in Karttunen et al. (1996). This date parser is implemented in *xfst* using the following regular expressions:[3]

[1] Details of the operators are available at `http://www.xrce.xerox.com/competencies/content-analysis/fsCompiler/fssyntax.html` and `http://www.xrce.xerox.com/competencies/content-analysis/fsCompiler/fssyntax-explicit.html`.

[2] The examples we provide are rather simple ones, as length restrictions do not allow us to include large figures to visualize complex finite state projects.

[3] The `define` command defines a named regular expression which can then be subsequently referred to in later regular expressions. `|` denotes the union operator. `0` (without quotes) denotes the empty string traditionally represented by ϵ in the literature. The quotes `"` are used to literalize sequence of symbols which have special roles in the regular expression language.

```
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 LeftBracket  [ "[" ];
define RightBracket [ "]" ];
define AllDates [Day|[def16 Month
             SPACE Date DateYear]];
define AllDatesParser
      [AllDates @->
         LeftBracket ... RightBracket];
read regex AllDatesParser;
```

The most important regular expression above is `AllDates`, a pattern that describes a set of calendar dates. It matches date expressions such as `Sunday, January 23, 2004` or just `Monday`. The subsequent regular expression `AllDatesParser` uses the *longest match downward bracket operator* (the combination of `@->` and `...`) to define a *transducer* that puts `[` and `]` around the longest matching patterns in the input side of the transducer.

Figure 1 shows the state of the screen of *Vi-xfst* just after the `AllDatesParser` regular expression is constructed. In this figure, the left side window shows, under the `Definitions` tab, the regular expressions defined. The top right window shows the template for the longest match regular expression slots filled by drag and drop from the list on the left. The `AllDatesParser` regular expression is entered by selecting the *longest-match downward bracket* operator (depicted with the icon `@->` with `...` underneath) from the palette above, which then inserts a template that has empty slots – three in this case. The user then "picks up" regular expressions from the left and drops them into the appropriate slots. When the regular expression is completed, it can be sent to the *xfst* process for compilation. The bottom right window, under the `Messages` tab, shows the messages received from the *xfst* process running in the background during the compilation of this and the previous regular expressions.

Figure 2 shows the user testing a regular expression loaded on to the stack of the *xfst*. The left window under the `Networks` tab, shows the networks pushed on to the *xfst* stack. The bottom right window under `Test` tab lists a series of input, one of which can be selected as the input string and then applied up or down to the topmost network on the stack.[4] The result of application appears on the bottom pane on the right. In this case, we see the input with the brackets inserted around the longest matching date pattern, `Sunday, January 23, 2004` in this case.

3.2 Visualizing regular expression structure

When developing or testing a large finite state transducer compiled from a regular expression built as a hierarchy of smaller regular expressions, it is very helpful, especially during development, to visualize the overall structure of the regular expression to easily see how components relate to each other.

Vi-xfst provides a facility for viewing the structure of a regular expression at various levels of detail. To illustrate this, we use a simple cascade of transducers simulating a coke machine dispensing cans of soft drink when the right amount of coins are dropped in.[5] The regular expressions for this example are:[6]

```
define N [ n ];
define D [ d ];
define Q [ q ];
define DefPLONK [ PLONK ];
define CENT [ c ];
define SixtyFiveCents [ [ [ CENT ]^65 ]
                          .x.
                        DefPLONK ];

define CENTS [[N .x. [[ CENT ]^5 ]|
              [D .x. [[ CENT ]^10 ]]]|
              [Q .x. [[ CENT ]^25 ]]];
define BuyCoke [ [ [ CENTS ]* ]
                   .o.
                 SixtyFiveCents ];
```

[4] *xfst* only allows the application of inputs to the topmost network on the stack.

[5] See `http://www.xrce.xerox.com/competencies/content-analysis/fsCompiler/fsexamples.html` for this example.

[6] The additional operators in this example are: `.x.` representing the cross-product and `.o.` representing the composition of transducers, and caret operator (^)denoting the repeated concatenation of its left argument as many times as indicated by its right argument.

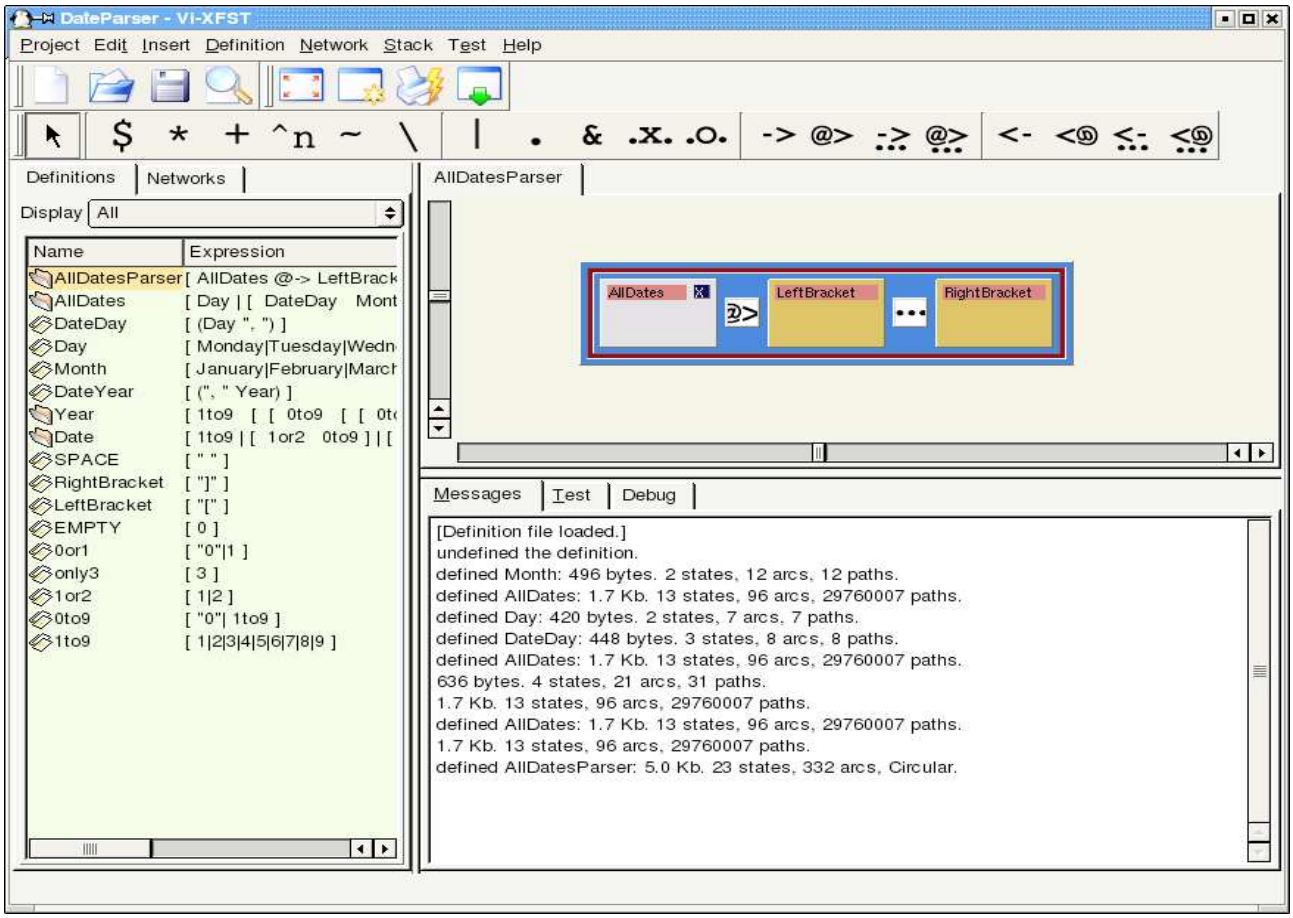

Figure 1: Constructing a regular expression via the drag-and-drop interface

The last regular expression here `BuyCoke` defines a transducer that consist of the composition of two other transducers. The transducer `[ CENTS ]*` maps any sequence of symbols `n`, `d`, and `q` representing, nickels, dimes and quarters, into the appropriate number of cents, represented as a sequence of `c` symbols. The transducer `SixtyFiveCents` maps a sequence of 65 `c` symbols to the symbol `PLONK` representing a can of soft drink (falling).

Figure 3 shows the simplest visualization of the `BuyCoke` transducer in which only the top level components of the compose operator (`.o.`) are displayed. The user can navigate among the visible regular expressions and "zoom" into any regular expressions further, if necessary. For instance, Figure 4 shows the rendering of the same transducer after the top transducer is expanded where we see the union of three cross-product operators, while Figure 5 shows the rendering after both components are expanded. When a regular expression laid out, the user can select any of the regular expressions displayed and make that the active transducer for testing (that is, push it onto the top of the *xfst* transducer stack)

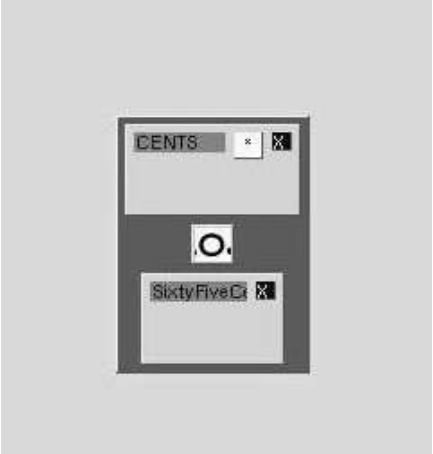

Figure 3: Simplest view of a regular expression

and rapidly navigate among the regular expressions without having to remember their names and locations in the files.

As we re-render the layout of a regular expression, we place the components of the compose and cross-product operators in a vertical layout, and others in

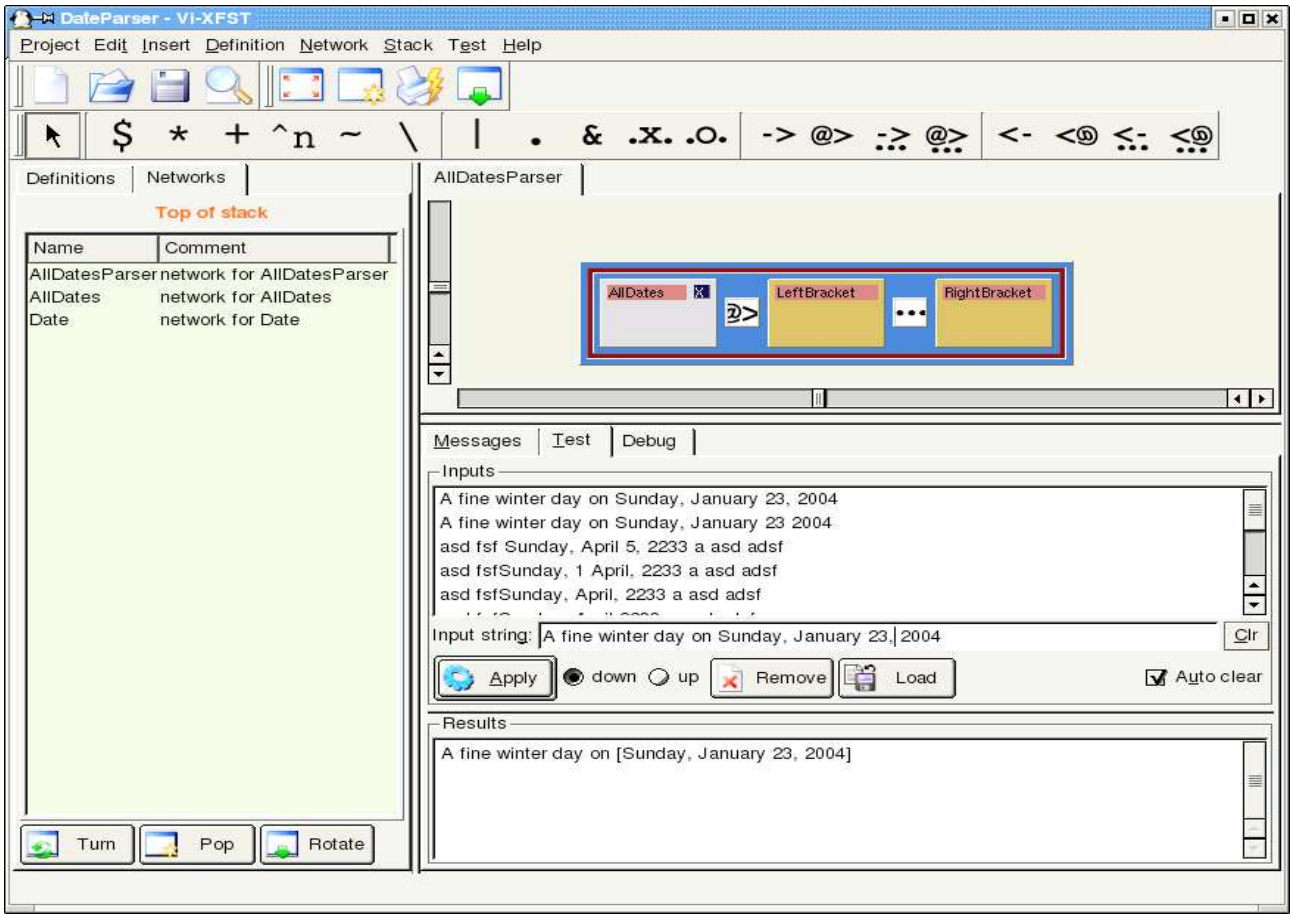

Figure 2: Testing a regular expression.

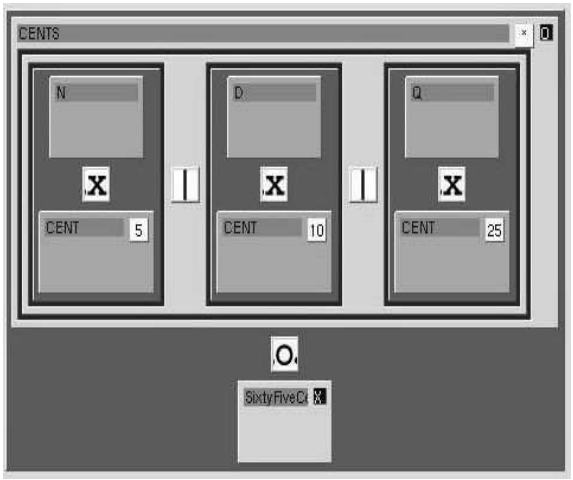

Figure 4: View after the top regular expression is expanded.

a horizontal layout and determine the best layout of the components to be displayed in a rectangular bounding box. It is also possible to render the upward and downward replace operators in a vertical layout, but we have opted to render them in a horizontal layout (as in Figure 1). The main reason for this is that although the components of the replace part of such an expression can be placed vertically, the contexts need to be placed in a horizontal layout. A visualization of a complex network employing a different layout of the replace rules is shown in Figure 6 with the Windows version of *Vi-xfst*. Here we see a portion of a Number-to-English mapping network[7] where different components are visualized at different structural resolutions.

3.3 Interaction of *Vi-xfst* with *xfst*

Vi-xfst interacts with *xfst* via inter-process communication. User actions on the *Vi-xfst* side get translated to *xfst* commands and get sent to *xfst* which maintains the overall state of the system in its own universe. Messages and outputs produced by *xfst* are piped back to *Vi-xfst*, which are then parsed

[7]Due to Lauri Karttunen; see `http://www.cis.upenn.edu/~cis639/assign/assign8.html` for the *xfst* script for this transducer. It maps numbers like 1234 into English strings like *One thousand two hundred and thirty four*.

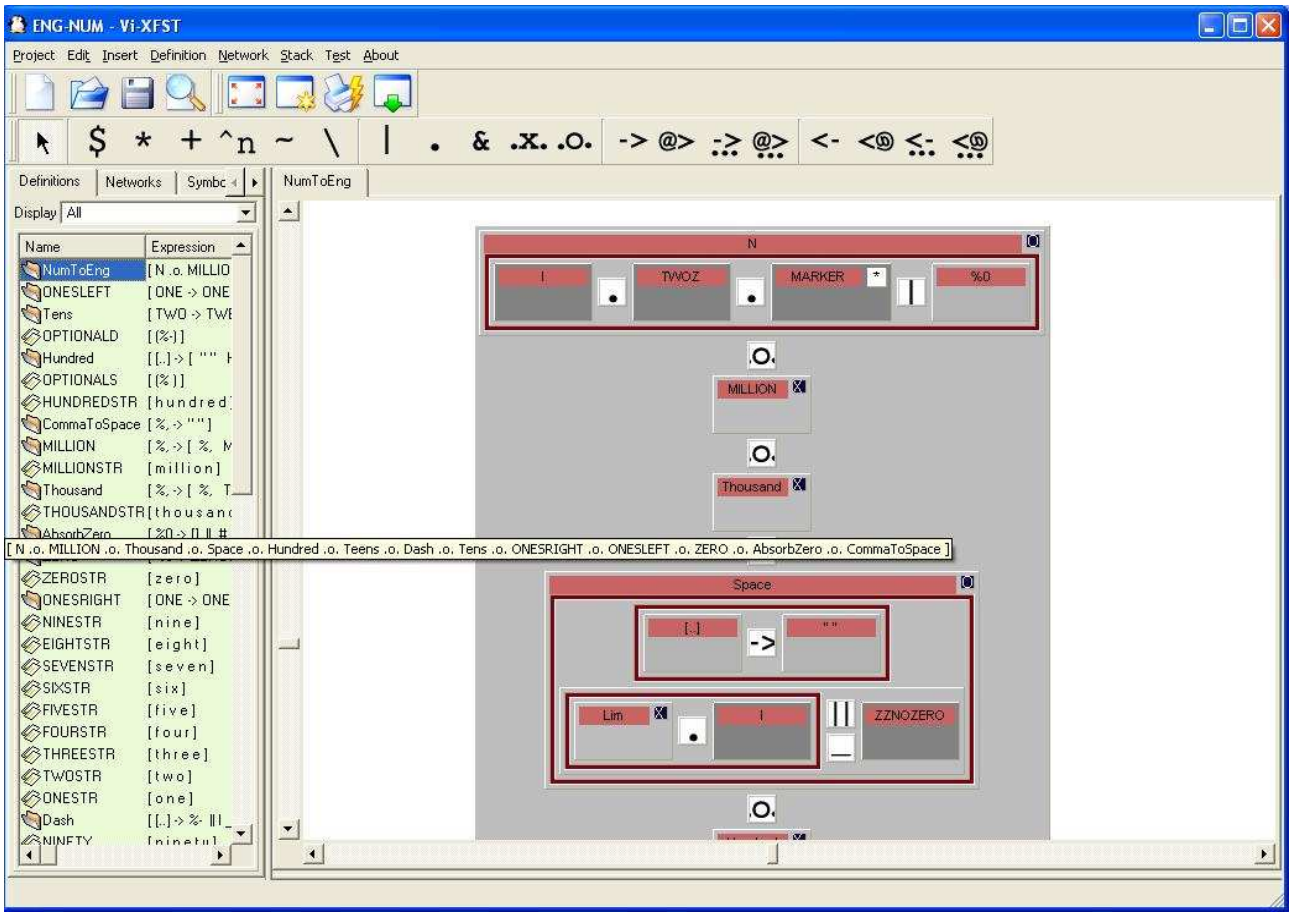

Figure 6: Mixed visualization of a complex network

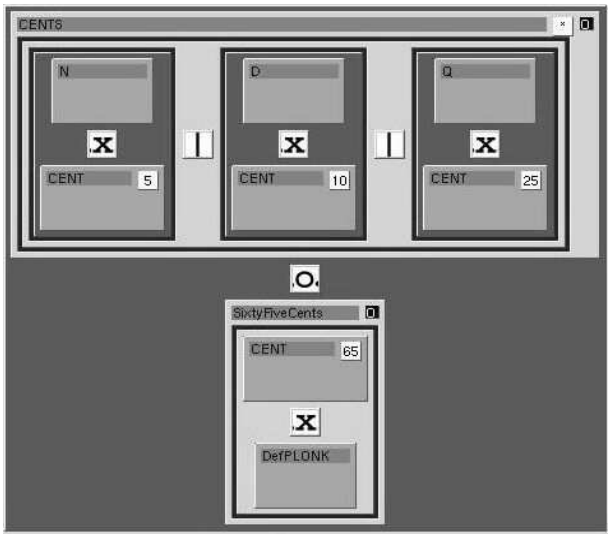

Figure 5: View after both regular expressions are expanded.

and presented back to the user. If a direct API is available to *xfst*, it would certainly be possible to implement tighter interface that would provide better error-handling and slightly improved interaction with the *xfst* functionality.

All the files that *Vi-xfst* produces for a project are directly compatible with and usable by *xfst*; that is, as far as *xfst* is concerned, those files are valid regular expression script files. *Vi-xfst* maintains all the additional bookkeeping as comments in these files and such information is meaningful only to *Vi-xfst* and used when a project is re-loaded to recover all dependency and debugging information originally computed or entered. Currently, *Vi-xfst* has some primitive facilities for directly importing hand generated files for *xfst* to enable manipulation of already existing projects.

4 Selective regular expression compilation

Selective compilation is one of the simple facilities available in many software development environments. A software development project uses selective compilation to compile modules that have been modified and those that depend (transitively) in some way (via say header file inclusion) to the modified modules. This selective compilation scheme,

typically known as the *make* operation, depends on a manually or automatically generated *makefile* capturing dependencies. It can save time during development as only the relevant files are recompiled after a set of modifications.

In the context of developing large scale finite state language processing application, we encounter the same issue. During testing, we recognize that a certain regular expression is buggy, fix it, and then have to recompile all others that use that regular expression as a component. It is certainly possible to use *make* and recompile the appropriate regular expression files. But, this has two major disadvantages:

- The user has to manually maintain the *makefile* that captures the dependencies and invokes the necessary compilation steps. This may be a non-trivial task for a large project.
- When even a singular regular expression is modified, the file the regular expression resides in, and all the other files containing regular expressions that (transitively) depend on that file, have to be recompiled. This may waste a considerable amount of time as many other regular expressions that do not need to be recompiled, are compiled just because they happen to reside in the same file with some other regular expression. Since some regular expressions may take a considerable amount of time to compile, this unnecessarily slows down the development process.

Vi-xfst provides a selective compilation functionality to address this problem by automatically keeping track of the regular expression level dependencies as they are built via the drag-and-drop interface. This dependency can then be exploited by *Vi-xfst* when a recompile needs to be done.

Figure 7 shows the directed acyclic dependency graph of the regular expressions in Section 3.1, extracted as the regular expressions are being defined.

A node in this graph represents a regular expression that has been defined, and when there is an arc from a node to another node, it indicates that the regular expression at the source of the arc *directly* depends on the regular expression at the target of the arc. For instance, in Figure 7, the regular expression `AllDates` directly depends on the regular expressions `Date`, `DateYear`, `Month`, `SPACE`, and `def16`.

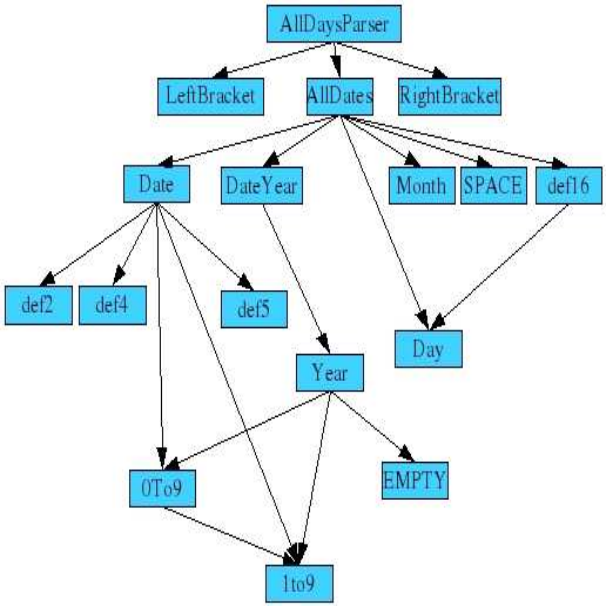

Figure 7: The dependency graph for the regular expressions of the `DateParser`.

After one or more regular expressions are modified, we first recompile (by sending a *define* command to *xfst*) those regular expressions, and then recompile all regular expressions starting with immediate dependents and traversing systematically upwards to the regular expressions of all "top" nodes on which no other regular expressions depend, making sure that

- all regular expressions that a regular expression depends on and have to be recompiled, are recompiled before that regular expression is recompiled, *and*
- every regular expression that needs to be recompiled is recompiled only once.

To achieve these, we compute the subgraph of the dependency graph that has all the nodes corresponding to the modified regular expressions and any other regular expressions that transitively depends on these regular expressions. Then, a topological sort of the resulting subgraph gives a possible linear ordering of the regular expression compilations.

For instance for the dependency subgraph in Figure 7, if the user modifies the definition of the network `1to9`, the dependency subgraph of the regular expressions that have to be recompiled would be the one shown in Figure 8. A (reverse) topological sort of this depen-

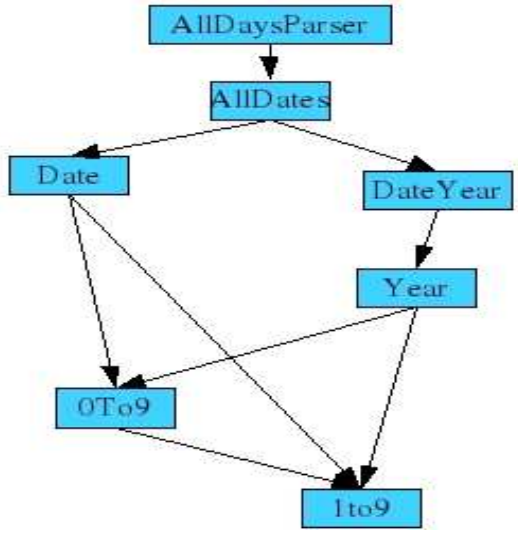

Figure 8: The dependency subgraph graph induced by the regular expression `1to9`.

dency subgraph gives us one of the possible orders for recompiling only the relevant regular expressions as: `1to9`, `0To9`, `Date`, `Year`, `DateYear`, `AllDates`, `AllDatesParser`

5 Conclusions and future work

We have described *Vi-xfst*, a visual interface and a development environment for the development of large finite state language processing application components, using the Xerox Finite State Tool *xfst*. In addition to a drag-and-drop user interface for constructing regular expressions in a hierarchical manner, *Vi-xfst* can visualize the structure of a regular expression at different levels of detail. It also keeps track of how regular expressions depend on each other and uses this dependency information for selective compilation of regular expressions when one or more regular expressions are modified during development.

The current version of *Vi-xfst* lacks certain features that we plan to add in the future versions. One important functionality that we plan to add is user customizable operator definitions so that new regular expression operators can be added by the user as opposed to being fixed at compile-time. The user can define the relevant aspects (slots, layout) of an operator in a configuration file which can be read at the program start-up time. Another important feature is the importing of libraries of regular expressions much like symbol libraries in drawing programs and the like.

The interface of *Vi-xfst* to the *xfst* itself is localized to a few modules. It is possible to interface with other finite state tools by rewriting these modules and providing user-definable operators.

6 Acknowledgments

We thank XRCE for providing us with the *xfst* and other related programs in the finite state suite.

References

Kenneth R. Beesley and Lauri Karttunen. 2003. *Finite State Morphology*. CSLI Publications, Stanford University.

Emden R. Gansner and Stephen C. North. 1999. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*.

Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328.

Lauri Karttunen, Tamas Gaal, and Andre Kempe. 1997. Xerox Finite-State Tool. Technical report, Xerox Research Centre Europe.

Mehryar Mohri, Fernando Pereira, and Michael Riley. 1998. A rational design for a weighted finite–state transducer library. In *Lecture Notes in Computer Science, 1436*. Springer Verlag.

Max Silberztein. 2000. Intex: An fst toolbox. *Theoretical Computer Science*, 231(1):33–46, January.

Gertjan van Noord. 1997. FSA utilities: A toolbox to manipulate finite state automata. In D. Raymond, D. Wood, and S. Yu, editors, *Automata Implementation*, number 1260 in Lecture Notes in Computer Science. Springer Verlag.

Yasin Yılmaz. 2003. Vi-XFST: A visual interface for Xerox Finite State Toolkit. Master's thesis, Sabancı University, July.