Using Genericity to Create Cutomizable Finite-State Tools

Sandro Pedrazzini, Marcus Hoffmann

Department of Computer Science, Basel University
Petersgraben 51
CH-4051 Basel, Switzerland
and
IDSIA
Corso Elvezia 36
CH-6900 Lugano, Switzerland
{sandro,marcuso}@idsia.ch

Abstract. In this article we present the realization of a generic finite-state system. The system has been used to create concrete lexical tools for word form analysis, word form generation, creation and derivation history, and spellchecking. It will also be used to create a finite-state transducer for the recognition of phrases. Producing a finite-state component with the generic system requires little effort. We will first emphasize its meaning and its architecture from a design point of view; then we will present some lexical finite-state tools created with it.

1 Introduction

The increasing need of finite-state components for different aspects in natural language processing has led us to the definition of a generic system for finite-state tools construction. An important aspect that should be considered comparing our resulting concrete finite-state automata with other existing ones (i.e. [5]) is that our automata are fed with data generated from an existing system, Word Manager ([1]; [3]), which is responsible for the specification, management and generation of morphological data. This means that the finite state component does not need a user defined regular expression input, instead it receives the extended paradigms, optimizing them following its internal needs. Another aspect to consider is the embedding of the single elements of the finite-state tools into a portable object-oriented framework, the architecture of which assures the reuse, the flexibility and the customization of the different parts. According to [4], a framework is more than a simple toolkit. It is a set of collaborating classes that make up a reusable design for a specific class of software. The purpose of the framework is to define the overall structure of an application, its partitioning into classes and objects, their collaboration, and the thread of control. These predefined design parameters allow the programmer to concentrate on the specifics of his application. He will customize the framework for a particular application by creating application specific subclasses of classes (eventually abstract) from the framework. The framework itself can be viewed as an abstract finite-state element. Only the definition of some concrete classes can generate from it a usable finite-state tool. The main design decisions have therefore already been taken, and the applications (finite-state elements) are faster to implement and even easier to maintain. The reasons why we have defined a framework are essentially two:

1. We wanted to achieve a broad software functionality with a small shared consistent structure.

2. We wanted to offer the opportunity to customize our work simply by subclassing parts of it and reusing other parts (hopefully most of them) as they are.

The aim of the project was not only the realization of the framework. The implementation of concrete subclasses that you can put to work immediately has also played an important role. First, as an example of how you can adapt the framework classes to your needs, and, second, as a realization of the specialized morphology processing programs mentioned before. The description is divided in two main parts. In the first one (section 2) we will describe the framework, emphasazing its meaning, its design and its ability to create concrete finite-state tools. In the second part (section 3) we will show the different functionalities that we have realized with the finite-state elements created with the framework.

2 Customization and Reuse

Instead of presenting the overall architecture of the system, we propose concentrating on the main parts of the system which can be easily modified for customization purposes. Explaining them will at the same time allow us to understand to what extent the remaining parts of the system are reusable. There are three main parts of the abstract finite-state element (framework) which must be customized:

- Node structure.
- Traversing algorithm.
- Information extraction, i.e. the operation applied to each single node of the finite-state element during the traversal.

For each of these customization steps some concrete classes already exists. The user who wants to create a new finite-state tool can decide to switch to one of them or to define a completely new class.

2.1 The Node Structure

Each finite-state tool can have a different kind of node, depending on the kind of information it must code and on its use, unidirectional or bi-directional. The opportunity to define a new node represents therefore a first level of customization. The new kind of node should take advantage of the existing managing algorithms, using them as they are, without further modifications. There are two methods for realising such a design: parameterized types and common classes. Parameterized types let you define an algorithm without specifying all types it uses. The unspecified types are supplied as parameters at the point of use. In our case all managing algorithms (insert, traverse, etc.) could be parameterized by the type of node they use. In C++, the language we used in our project, this can be easily performed with templates. The second method makes use of inheritance and polymorphism. It defines a common (abstract) class Node, which serves to specify the interface of all possible nodes. Each implementation can specify a different concrete subclass of Node, able to respond to the requests defined in the interface. All managing algorithms only refer to the abstract class for their operations. They work with concrete nodes just at run-time. The C++ template method is more efficient and probably easier to understand. However it presents some drawbacks. First, at the moment of deciding the design (and still at present), it did not guarantee a complete portability of the code over different C++ compilers, whereas the rest of the framework code did. Second, it is

just a type substitution and does not support any abstraction or hiding of some new specific data or functions (process of coding and decoding data, for example) in an object-oriented way. Third, it would require adapting other parts of the program (e.g. traversal) for every new type of node, introducing an undesired dependence between different customizable parts. Because of these disadvantages and because we judged the second method more flexible, we chose the second one. Moreover, parameterized types is a concept which is not known in every programming language, and this would restrict the generalization of our software design, which is intended to be independent of any programming language. Notice that the method used is also called Template in the design patterns terminology. The abstract Node class must define the interface, previewing all basic functionalities required for the nodes by the internal algorithms. The latter will use the concrete elements through Node references.

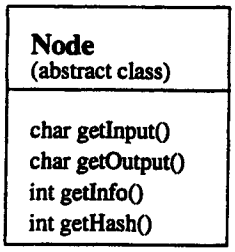

Figure 1. Node

Customization: In order to add a new particular kind of node, the customizer must write a new concrete subclass of Node, defining the content type and implementing all methods defined as interface in the abstract class. For example, the node used for the implementation of the lexical transducer contains two characters, input and output. New methods not included in the interface can also be specified, however they will only be used in customized parts.

2.2 The Traversing Algorithm

There are different kinds of traversing algorithms depending on the purpose of each single tool and on the knowledge it is supposed to code. A transducer used to generate word forms needs a non deterministic traversing method, because the same input will generate different output strings, a simple FSA used for spellchecking can use a deterministic traverse, the opportunity of looping over nodes will be useful for phrase recognition, some other traversals could need the recognition of a special character for some special purposes, etc. As we can see there are many algorithms to consider for traversing finite-state elements. Hard-wiring all of them into the class that may require them is not desirable. First because the class will get more complex if it has to include all possible algorithms, and different algorithms will be appropriate at different times;

second because it will become difficult to add new algorithms or to vary existing ones when traversal is an integral part of the class that uses it. We can avoid these problems encapsulating all different traversing algorithms in different classes, using the same interface. The interface is defined by a common superclass, the Strategy class. The intent of the Strategy pattern is to define a family of algorithms, encapsulate each one, and make them interchangeable.

Customization: In order to add a new kind of traverse, the customizer must simply write a new subclass of Strategy with its method traverse.

2.3 Information Extraction

The main feature here is the separation of information extraction performed during the traversal process, from the traversal algorithm itself. We must keep the responsibility of the action away from the traversal part. In this way we can use the same finite-state tool to deliver a different type of information. We used as model the Visitor pattern, although this pattern is merely thought for a use with different kinds of nodes at the same time. The information extraction process is embedded into the class Visitor. Subclassing the visitor means reusing the nodes of the finite-state system, building with it a new kind of answer. During the retrieval process the internal data in the nodes remains read-only, i.e. unmodified. The adaptation is in the way the data will be used for the external result. For example, the difference between the information extracted from a lemmatizer and the information extracted from a morphosyntactic analyzer can be coded uniquely distinguishing two different interpretations of the same data, i.e. modifying the action performed during the traversal. The traversal process is responsible for leading the control through the structure, whereas the action, which will be called for each node, involves accumulating information during it. This is particularly useful with lexical transducers, which store input and output information in the nodes. Separating the retrieval process from the internal structure will bring more flexibility and potential for reuse, because different kinds of retrieval often require the same kind of traversal. In addition, we will simplify the task of customizing the retrieval, restricting the modification to the action. The implementation is organized as follows: there is a class (called Fsa) that contains the main structure. Each node of the structure will receive an instance of the concrete Visitor during the traversal. The instance is used for accumulating information, creating the final result of the analysis. The acceptance of each visitor object, including the customized ones, is achieved through polymorphism. In order to be accepted, the concrete object must inherit from the abstract class Visitor, which defines the interface for the whole hierarchy. The overall pattern is shown in figure 2. Each box corresponds to a class with its own methods. The abstract class Visitor is shown with two (among many possible) inheriting concrete classes. The class Fsa has a reference to the whole internal data, represented here by the class Node.

Any FSA specific data structure remains separated and hidden for the visitor object, simplifying the task of the customer.

Customization: In order to add a new kind of information extraction to the structure, the customizer must write a new subclass of Visitor with its method visitNode, or a new subclass of an already existing concrete class inheriting from Visitor. In the first case he will customize the system reusing the design, in the second case he will adapt it reusing both, design and code.

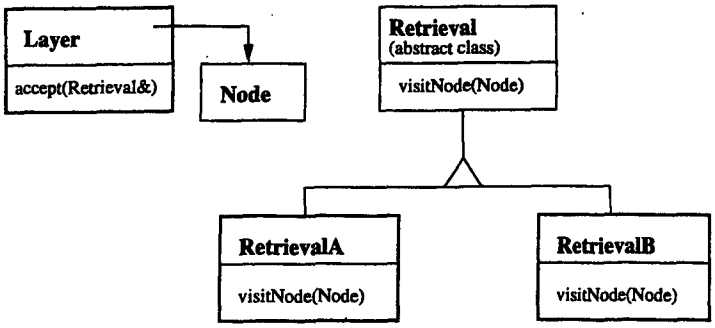

Figure 2. Visitor

3 Realized tools

In this section we will describe some concrete lexical tools realized using the described framework. The tools take as source data a Word Manager database with morphological and lexical entries. They read it and they generate their independent internal structure, efficient in space. For example, the source file encoding the inflection information of about 100,000 German lexemes (1 million word forms, including wordformations) occupies 26 Mb, but the file for the corresponding finite-state transducer, used as morphosyntactic analyzer and generator, was less than 1.8 Mb. The same transducer used as morphosyntactic analyzer reaches a speed of 8,000 words/s to 12,000 words/s (SPARC 20), depending on the requested kind of answer. Another example is the generative spellchecker, which reaches a similar speed (up to 14,000 words/s), but at a compression rate of less than one byte per word (about 800 kbytes for 1 million word forms).

3.1 Transducers

A first set of examples consists of finite-state transducers based on inflection. The tools are illustrated with forms of the verb gehen ('go'). The same transducer can be used for the following four functionalities:

- Lemmatizer.
  It takes as its input a word form as found in a text and yields a set of identifiers of lexemes to which it may belong. Often the set will consist of a single element.

```
IN "ging"
OUT "gehen" (Cat V)
```

- Paradigm generator.
  It generates the word forms of a lexeme identified in the input.

```
IN "gehen" (Cat V)
OUT
gehen, gehe, gehst, geht, ging, gingst, gingen, gingt, gehest, gehet, ginge
gingest, ginget, geh, gehend, gegangen
```

- Morphosyntactic analyser.
  The input is the same as in the former example, but the output specifies the position in the paradigm of the lexeme.

```
IN "ging"
OUT
"gehen" (Cat V)(Mod Ind)(Temp Impf)(Num SG)(Pers 1st)
"gehen" (Cat V)(Mod Ind)(Temp Impf)(Num SG)(Pers 3rd)
```

- Morphosyntactic generator.
  It is the reverse of the previous example.

```
IN "gehen" (Cat V)(Mod Ind)(Temp Impf)(Num SG)(Pers 1st)
OUT "ging"
```

There is also the opportunity to restrict the answer producing a partial set of word forms on the basis of a partial feature specification. Obviously inflection information can be required for any kind of word, including compoundings. We have also used the same transducer structure to encode information on derivation and creation history. The source data were also the 100,000 German lexemes.

- Creation history.
  Given the identifier of a lexeme, here Sperrung ('closing'), it retrieves the base and the WM word formation rule if the input lexeme is complex.

```
IN "sperrung" (Cat N)(Gender F)
OUT
"sperren" Derivation V-To-N Suffixing No-Umlaut V-No-Det-Prefix
```

- Generation history.
  The same data can be used in the reverse order, i.e. from the base lexeme to the derived ones. In the following example the transducer generates all lexeme identifiers of lexemes formed by word formation rules applied to a given input lexeme, in this case kind ('child').

```
IN "kind" (Cat N)(Gender N)
OUT
"enkelkind" (Cat N)(Gender N)
"wunderkind" (Cat N)(Gender N)
"schulkind" (Cat N)(Gender N)
etc.
```

3.2 Simple FSA

A further category of tools includes FSAs. They are not bi-directional as the transducers, but they can still be used for different purposes:

- Structuring into formatives.

```
IN "gegangen"
OUT "ge + gang + en"
```

- Spelling-checker (yes/no answer).

```
IN "ging"
OUT yes
```

- Generative spelling-checker.
  It is similar to the former spelling-checker, in the sense that it does not give strings or features as output, but only yes or no, depending on successful recognition. The difference is that this FSA has been generated taking existing word forms from Word Manager database into account as well as formatives which may result from the application of productive Word Manager wordformation rules. As opposed to common spellcheckers, which only check a text against a word list, it could be called a generative spellchecker ([7]), because it also tries to generate the word as possible word.

```
IN "krankenversicherungssystem"
OUT possible word
```

3.3 Others

Some more finite-state tools are foreseen. The most important of them is a special transducer able to recognize particular sentences. The sentences will be acquired through an existing product, Phrase Manager ([6]), which will generate the data used to produce the independent transducer.

4 Conclusion

In this paper, various finite-state tools have been described which are based on a general common framework. The significance of these tools does not reside primarily in their individual functionalities. Although each of them is useful and fast, their principal interest lies in the fact that they are produced with so little effort on the basis of an existing object-oriented framework. The framework represents an abstract finite-state element, which can be easily customized to produce new kinds of concrete finite-state tools. We think that the use of a customizable framework, as well as the use of a source database for the generation of the input, is a further step towards the optimal reuse of expensive tasks like defining some tens of thousands of entries which will usually not focus on a single application.

References

1. Domenig M. and ten Hacken P. 1992. Word Manager: A System for Morphological Dictionaries. Olms Verlag, Hildesheim.
2. Gamma E., Helm R., Johnson R., Vlissides J. 1995. Design Patterns. Addison Wesley.
3. ten Hacken P, Bopp S., Domenig M., Holz D., Hsiung A, Pedrazzini S. 1994 A Knowledge Acquisition and Management System for Morphological Dictionaries. In *Proceedings of Coling-94, International Conference on Computer Linguistics*, Kyoto.
4. Johnson R.E and Foote B. 1988. Designing reusable classes. Journal of Object-Oriented Programming. June/July 1988.
5. Karttunen Lauri. 1993. Finite-State Lexicon Compiler. Xerox Corporation Palo Alto Research Center. Technical Report [P93-00077].
6. Pedrazzini Sandro. 1994. Phrase Manager: A System for Phrasal and Idiomatic Dictionaries. Olms Verlag, Hildesheim.
7. Pedrazzini Sandro. 1997. Word Games. In *Proceedings of the Fifth International Symposium on Social Communication*, Santiago de Cuba, Editorial Academia, La Habana.