TU/e technische universiteit eindhoven

TABASCO:
TAxonomy-BAsed Software COnstruction
+
A Keyword Pattern Matching Example

Loek Cleophas
l.g.w.a.cleophas@tue.nl

May 11, 2005

Software Construction Group
FASTAR Research
Espresso Research

http://www.win.tue.nl/soc
http://www.fastar.org
http://espresso.cs.up.ac.za

- Overview
- TABASCO
- Taxonomy Construction
  - Keyword Pattern Matching
- Toolkit Design & Implementation
  - SPARE Time
- DSL Design & Implementation
  - Fuel & SPARE Fuel
- Concluding Remarks
  - Project Ideas

References

Aim

To give you an overview of the TABASCO method and an example of its application

This includes:
taxonomy construction; toolkit construction; domain-specific languages; role of benchmarking

Motivation

Useful approach to software construction, in the form of domain-specific toolkits

Course assignment involves small scale application of the method

→ Tom Verhoeff and Bruce Watson will elaborate on some of this in later lectures

TAxonomy-BAsed Software COnstruction

- Long term SoC group research theme
- Method for constructing domain-specific toolkits based on taxonomies
- Aims
  - ease comparison and understanding
  - provide correctness arguments
  - ease implementation and choice
  - lead to new algorithms

TABASCO Steps

1. Selection of problem field
2. Literature survey
3. Taxonomy construction
4. Toolkit design
5. DSL design
6. Toolkit implementation
7. Benchmarking
8. DSL implementation

Application areas

Done:

- Garbage collection
- Attribute evaluation
- Single- and multiple keyword pattern matching
- Automata construction
- Deterministic automata minimization
- LZ compression algorithms
- Approximate pattern matching
- Graph representations

Upcoming:

- Minimal acyclic deterministic automata construction
- 2D pattern matching
- Tree parsing & pattern matching

All of these were/are Master's or PhD thesis subjects, your assignments will likely

- Involve a smaller domain and/or
- Deal with a subset of algorithms in a domain and/or
- Use a less formal domain model

1. **Selection of problem field / domain**
   - Must have sufficient algorithms / data structures
   - Preferably has practical applications
2. **Literature survey**
   - Search literature for algorithms
   - Look for commonalities and variation among them
   - Abstract away unnecessary details or presentation style

→ Note similarity to pattern analysis approach presented by prof. Watson in earlier lecture

3. Taxonomy construction

What is a taxonomy?

- Classification according to *algorithm & problem details*
- Algorithm details → algorithm structure
- Problem details → problem changes
- Tree form (actually, *directed acyclic graph*)
- Nodes are algorithms, branches represent details added
- Root is most simple solution
  → For KPM: 'the solution is to take the set of occurrences of the keywords in the text'
- Refinement by adding details (branches)
- Taxonomy forms a *domain model*

3. Taxonomy construction

- Given algorithms from literature, distill commonalities and variation
  - Commonalities result in common root path
  - Adding detail (branch) results in correctness preserving transformation
  - Formal derivation/proofs accompany detail (branch)
- Development often bottom-up
- Final presentation top-down
- Multiple paths to same solution possible

What Is Keyword Pattern Matching (KPM)?

> The problem of finding all occurrences of keywords from a given set as substrings in a given string (text)

Note: For simplicity, in this presentation we sometimes assume a *single* keyword instead of a *set* of keywords

Example:

Position											1
	0	1	2	3	4	5	6	7	8	9	0
String	*a*	*b*	*a*	*b*	*y*	*d*	*a*	*b*	*a*	*b*	*y*

Keyword *baby* occurs twice, ending at positions 4 and 10

Solutions & Taxonomies (I)

KPM has many solutions

Watson & Zwaan developed taxonomy and toolkit in early 90s; aims:

- obtain correctness arguments / proofs
- make algorithms easy to understand / compare
- find new algorithms
- create large collection of implementations
- compare running time performance

Solutions & Taxonomies (II)

Cleophas extended taxonomy and extended & revised toolkit in 2003

- Taxonomy extended;
  - new algorithms described in past decade
  - other algorithms gained attention in past decade
- Toolkit revised & extended
  - C++ language changes: booleans, templates
  - stable STL, providing basic data structures, iterators, ...
  - add algorithms from new taxonomy part to toolkit

Some Necessary 'Definitions'

- *prefixes* of a string are strings that appear as substrings at its beginning
- *suffixes* of a string are strings that appear as substrings at its end
- *factors* of a string are strings that appear as substrings

Note: This includes the string itself and the empty word ε.

Example: For the word *baby*,

- the prefixes are baby, bab, ba, b, ε
- the suffixes are baby, aby, by, y, ε
- the factors are baby, bab, aby, ba, ab, by, b (occurs as a factor twice), a, y, ε

Categories of Pattern Matching - Some Observations and Remarks (I)

Forward, Naive

- each character of text read at least once
- uses trie recognizing prefixes of keywords, or no automaton at all

Forward, Prefix-based

- each character of text read exactly once
- uses automaton which is extension of above trie
- some use bit-parallelism

Categories of Pattern Matching - Some Observations and Remarks (II)

Backward, Suffix-based

- not all characters of text have to be read
- uses reverse suffix automaton / reverse trie: recognizes keywords' reverse suffixes
- shifts of more than 1 position possible

Backward, Factor-based

- uses factor automaton: larger automaton, recognizes keywords' reverse factors
- reads more characters than suffix-based
- larger shifts than suffix-based
- some use bit-parallelism

Categories of Pattern Matching - Some Observations and Remarks (III)

Backward, Factor Oracle-based

- uses factor oracle: recognizes at least keywords' reverse factors, possibly more
- reads more characters than factor-based
- same shifts as with factor-based
- easier to construct than factor automaton
- smaller than factor automaton

Categories of Pattern Matching - "Famous" Algorithms

Forward, Naive

Forward, Prefix-based
Aho-Corasick,
Knuth-Morris-Pratt,
Shift-And

Backward, Suffix-based
Commentz-Walter,
Boyer-Moore,
Boyer-Moore-Horspool

Backward, Factor-based
Backward DAWG Matching,
Backward Nondeterm. DAWG Matching

Backward, Factor oracle-based
Backward Oracle Matching

(*Filtration-based*
Karp-Rabin and others)

Taxonomy Algorithms (I)

(S is text string, P is keyword set)

{ Algorithm () (*Root Algorithm*) }
$O := (\bigcup\ l, v, r : lvr = S \wedge v \in P : \{(l, v, r)\})$
$\{\ R : O = (\bigcup\ l, v, r : lvr = S \wedge v \in P : \{(l, v, r)\})\ \}$

{ Algorithm (P) }
$O := \varnothing;$
for $(u, r) : ur = S \rightarrow$
    $O := O \cup (\bigcup\ l, v : lv = u \wedge v \in P : \{(l, v, r)\})$
rof$\{\ R\ \}$

{ Algorithm (P,S) }
$O := \varnothing;$
for $(u, r) : ur = S \rightarrow$
    for $(l, v) : lv = u \rightarrow$
        as $v \in P \rightarrow O := O \cup \{(l, v, r)\}$ **sa**
    rof
rof$\{\ R\ \}$

{ Algorithm (P+,S) }
$u, r := \varepsilon, S;$
if $\varepsilon \in P \rightarrow O := \{(\varepsilon, \varepsilon, S)\} \parallel \varepsilon \notin P \rightarrow O := \varnothing$ **fi**
do $r \neq \varepsilon \rightarrow$
    $u, r := u(r\uparrow 1), r\downarrow 1;$
    for $(l, v) : lv = u \rightarrow$
        as $v \in P \rightarrow O := O \cup \{(l, v, r)\}$ **sa**
    rof
od$\{\ R\ \}$

Taxonomy Algorithms (II)

{ Algorithm (P+, S+) }
$u, r := \varepsilon, S;$
if $\varepsilon \in P \to O := \{(\varepsilon, \varepsilon, S)\} \,[\!]\, \varepsilon \notin P \to O := \varnothing$ **fi**
do $r \neq \varepsilon \to$
    $u, r := u(r\!\uparrow\!1), r\!\downarrow\!1;$
    $l, v := u, \varepsilon;$
    as $\varepsilon \in P \to O := O \cup \{(u, \varepsilon, r)\}$ **sa**
    do $l \neq \varepsilon \to$
        $l, v := l\!\downarrow\!1, (l\!\uparrow\!1)v;$
        as $v \in P \to O := O \cup \{(l, v, r)\}$ **sa**
    od
od{ R }

{ Algorithm (P+, S+, GS, EGC, SSD) }
$u, r := \varepsilon, S;$
if $\varepsilon \in P \to O := \{(\varepsilon, \varepsilon, S)\} \,[\!]\, \varepsilon \notin P \to O := \varnothing$ **fi**
$l, v := \varepsilon, \varepsilon;$
do $r \neq \varepsilon \to$
    $u, r := u(r\!\uparrow\!k(l,v,r)), r\!\downarrow\!k(l,v,r);$
    $l, v := u, \varepsilon;$
    $q := \delta_{R,\mathbf{f}}(q_0, l\!\uparrow\!1);$
    as $\varepsilon \in P \to O := O \cup \{(u, \varepsilon, r)\}$ **sa**
    { invariant: $q = \delta^*_{R,\mathbf{f}}(q_0, ((l\!\uparrow\!1)v)^R)$ }
    do $l \neq \varepsilon$ **cand** $q \neq \perp \to$
        $l, v := l\!\downarrow\!1, (l\!\uparrow\!1)v;$
        $q := \delta_{R,\mathbf{f}}(q, l\!\uparrow\!1);$
        as $v \in P \to O := O \cup \{(l, v, r)\}$ **sa**
    od
od{ R }

4. Toolkit design

- Made more straightforward by taxonomy
- Taxonomy makes commonalities and variation explicit, showing grouping of algorithms
- Design based on standard techniques for handling commonalities and variation, such as those in Coplien's Multi-paradigm Design for C++ book
- Still requires some creativity and experience
- Future work: apply this step to other taxonomies to learn more about guidelines to get from taxonomy to toolkit design

SPARE TIME: a KPM toolkit (I)

- follows structure of taxonomy closely
- shallow class hierarchy, four concrete classes
  - empty pattern matcher class PM, with children PMSingle and PMMulti
  - single-keyword PMKMP and PMBM derived from PMSingle
  - multiple-keyword PMAC and PMCW derived from PMMulti
- corresponds to distinction in taxonomy between forward and backward reading algorithms, plus difference between multiple and single keyword algorithms

SPARE Time: a KPM toolkit (II)

- many template classes used:
  - PMAC, takes automaton as argument
  - PMBM, takes shifter, skip loop class as arguments
  - PMCW, takes automaton, shifter class as arguments
  - automaton classes for PMAC, PMCW
  - shifter classes for PMBM, PMCW
  - skip loop classes for PMBM
- pattern matchers have same interface

SPARE TIME: some remarks

- Taxonomy-basedness leads to clear structure
- Structure follows from taxonomy and standard techniques (use of patterns, multi-paradigm design)
- Use of formal techniques in deriving algorithms and creating a taxonomy helped in comparing and implementing

SPARE TIME: future work

- Still some well-known algorithms to add
- Generalize suffix-based shifters to work with other automata
- Toolkit tuning & benchmarking—done
- Development of 'little language' to simplify toolkit use—done
- Extensions to related fields: approximate & regular expression pattern matching

5. DSL design

- Toolkits to be used by people who may know little about toolkit implementation language
- Toolkit users may be domain experts, or average users without much knowledge about the domain
- Develop DSL or 'little language' to cater to users
- Usually focuses on performance and problem details, not on algorithm details
- Users write DSL expression, get toolkit components
- Mapping needed from DSL expression to toolkit components

6. Toolkit implementation

Relatively easy, due to availability of algorithm presentations in taxonomy, and of the toolkit design.

Actual implementation depends on implementation language chosen

7. Benchmarking

Necessary to fill in DSL requirements-components mapping, since a DSL usually allows one to specify performance requirements (running time, memory usage)

Confidence in results due to similar style and implementation of algorithms

8. DSL implementation

We distinguish three forms of DSL implementations:

- Manual selection
- Explicit DSL, possibly using generator to create (composition of) toolkit component
- DSL as output of DSL generator, which uses a configuration file specifying the DSL

Manual selection

Textual guideline on how to use toolkit, or pseudo-code

```
if performance independent of
keyword set is required then
  AC-OPT
else
  if multiple-keyword sets
  are used then
    if fewer than thirteen
    keywords and the shortest
    keyword length is at least
    four then
      CW-NORM
    else
      choose an AC algorithm
...
```

Explicit DSL

Possibly using *generator*

Example: sorting algorithm DSL

Grammar:

```
Sorter     sorter[ArraySize, UniqueRange, Randomness,
           Stable, Auxiliaries, Order]
ArraySize     arraySmall | arrayLarge
UniqueRange     rangeSmall | rangeLarge
Randomness     random | nearSorted | nearReversed | unknown
Stable         stable | nonStable
Auxiliaries     useAuxiliaries | noAuxiliaries
Order          n_log_n | n_n
```

DSL implemented using C++ template metaprogramming, quite error prone

Using it:

```
#include "miloDSL.h"

typedef SORTER_GENERATOR<sorter<arraySmall<>, whatever,
 whatever, whatever, whatever, n_log_n<>>>::RET aSorterType;

aSorterType mySorter;
mySorter::sort( myArray, myArray+1000 );
```

Can use Perl, Python, Ruby to implement instead...

DSL as output of DSL generator

Uses configuration file specifying the DSL

Example: Fuel tool

Takes XML file specifying DSL

```
<fuel>
 <copyright name="SPARE Fuel" version="0.1" author="Jan Ouwens"
            date="2004" notes="DSL for SPARE Time"/>

 <delimiter open="[*" close="*]"/>

 <codefiles>
  <file src="sparefuel.src" target="sparefuel.h"/>
 </codefiles>

 <algorithms>
  <alg id="CWNFS" name="CW-[NFS, Factoracle]"
                  classname="PMCW<CWShiftNFS, RFactoracle>">
   <param>cw/cws.hpp</param>
  </alg>
  ...
 </algorithms>

 <rules>
  ...
  <rule name="alphabet" value="English" default="CWNFS">
   ...
   <rule name="memory" value="high" default="CWNFS">
    ...
    <rule name="length" value="3" default="ACOpt">
     <rule name="setsize" value="1" default="CWNFS"/>
     <rule name="setsize" value="2" default="CWBMH"/>
     <rule name="setsize" value="3..20" default="ACOpt"/>
    </rule>
    ...
   </rule>
  </rule>
 </rules>
</fuel>
```

Generates component when called with XML file and DSL expression

Call using example XML:

```
./fuel.rb sparefuel alphabet=English memory=high \
    length=3 setsize=1
```

Generates following include file for user:

```
/*
  SPARE Time include file.
  Generated by SPARE Fuel, version 0.1.
  Chosen algorithm: CW-[NFS, Factoracle]
  Parameters: alphabet=English, memory=high,
              length=3, setsize=1
*/

#ifndef SPAREFUEL_H
#define SPAREFUEL_H

#include "cw/cws.hpp"
typedef PMCW<CWShiftNFS, RFactoracle> PMAlg;

#endif
```

Reminder: TABASCO Steps

1. Selection of problem field
2. Literature survey
3. Taxonomy construction
4. Toolkit design
5. DSL design
6. Toolkit implementation
7. Benchmarking
8. DSL implementation

→ Apply (some of) it yourselves in your assignments

TABASCO and domain engineering

- TABASCO can be considered a domain engineering method for a restricted form of domains
- More on TABASCO as domain engineering in paper by Cleophas & Watson, and on domain engineering in Generative Programming book by Czarnecki and Eisenecker
- Instead of taxonomy, other domain models can be used, e.g. feature models
- Prof. Watson likely to discuss feature models in a future lecture

Project ideas

- Pattern matching in compressed text
- Sorting algorithms
- Compression algorithms
- String parsing
- ... other ideas

Tom Verhoeff will present some ideas in upcoming lecture as well

Other ideas are possible as well

Talk to Tom Verhoeff or Bruce Watson to get approval for an idea you have

References (I)

L.G.W.A. Cleophas & B.W. Watson, TABASCO: *a Taxonomy based Domain Engineering Method*, submitted to SAICSIT 2005, May 2005.

L.G.W.A. Cleophas & B.W. Watson, *Taxonomy-based software construction of* SPARE TIME: *a case study*, IEE Proceedings—Software, 152(1):29-37, February 2005.

K. Czarnecki & U.W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

L.G.W.A. Cleophas, B.W. Watson & G. Zwaan, *A new taxonomy of sublinear keyword pattern matching algorithms*, CS-Report 04-07, TU/e, March 2004.

References (II)

B.W. Watson & L.G.W.A. Cleophas, *SPARE Parts: a C++ toolkit for string pattern recognition,* Software—Practice & Experience, 34(7):697-710, June 2004.

L.G.W.A. Cleophas, *Towards* SPARE TIME - *A New Taxonomy and Toolkit of Keyword Pattern Matching Algorithms,* MSc thesis, TU/e, August 2003.

B.W. Watson & G. Zwaan, *A taxonomy of sublinear multiple keyword pattern matching algorithms,* Science of Computer Programming 27(2):85-118, September 1996.

B.W. Watson, *Taxonomies and Toolkits of Regular Language Algorithms,* PhD thesis, Technische Universiteit Eindhoven, 1995.

References (III)

http://www.fastar.org

http://www.win.tue.nl/soc

http://espresso.cs.up.ac.za

Next lecture

Either May 18 or May 25, check following websites next week

http://www.win.tue.nl/~watson/2IS20/

http://www.win.tue.nl/~wstomv/edu/soc/