Kind Parsing: An Adaptive Parsing Technique

Michal Žemlička and Pavel Šašek
Department of Software Engineering
Charles University, Faculty of Mathematics and Physics
Prague, Czech Republic
Email: zemlicka@ksi.mff.cuni.cz, pavel.sasek@email.cz

***Abstract*—Applications have many opportunities to be adaptive. One of them is the ability to control the input language(s). To achieve this objective we present an efficient semi-top-down parsing technique – kind parsing. It supports on-line (parse-time) as well as off-line extensions, restrictions or other adaptation of the accepted language. Kind parsers can be adapted quickly and the changes of the parser's structure tend to be only local. We suppose that such a technique can be very useful for adaptive system development.**

***Keywords*-extensible parsing, adaptive parsing, parse-time extension.**

I. Introduction

Development and maintenance of adaptive applications could become simpler if adaptive techniques were used. Do we have enough usable adaptive techniques? We suppose that it would come to useful if there were more techniques available. Therefore, this paper describes one of them.

In the case that our input data are in an XML-based format, we can use standard XML techniques. As XML is extensible from the beginning, the XML-parsers and other XML-tools should be able to cope with it somehow.

But what happens if the input is not XML-based? Hand-written parsers as well as parsers generated by usual parser generators are not (reasonably easily) extensible. They are built to handle just one language – the one specified in the beginning. So, should we give up adaptivity for non-XML inputs? No!

We should use parsers that are extensible or even modifiable. The parser extensibility or modifiability can be off-line (applicable only on parsers that do not parse at the moment) or on-line (applicable also during parsing). To support adaptability or self-adaptability it could be better if the parser could be extended or modified on-line.

There is a parsing technique called *kind parsing* [1], [2] allowing on-line changes of the parsers and being efficient (parsing in the linear time). We will shortly describe the parsing technique here and show that it is possible to use its structure also for on-line language restrictions. If we have both on-line extensibility and restrictability of the input language, we get an on-line modifiable (adaptable) parsing technique.

A. *Paper Structure*

It is reasonable to start projects with the specification of requirements. We collect requirements on parsers to be adaptive in Section II. The structure of Kind Parsers is introduced in Section III. Section IV presents how they work. Section V describes how Kind Parsers can be extended and when it is safe to do that. A discussion about communication of parsers with other parts is in Section VI. Issues related to an (on-line) parser restriction are covered by Section VII.

II. Requirements

We expect that readers are a bit familiar with the basics of the parsing theory presented in many nice books – let us mention at least a few of them: [3], [4], [5], [6].

To change the parser efficiently it is good if the changes are only local – i.e., it is not necessary to generate the entire parser again and only a few its parts are affected. If the changes are on-line, the computation, which has been done, must be preserved. On condition that we use a pushdown automaton, it involves at least the current state and all symbols on the stack. More precisely, it should be possible to get from the current parsing configuration to some final configuration or to other parser-changing configuration.

Therefore, we need a parser structure that is easily modifiable and computation-preserving. Such a requirement holds for structures like a tree or a forest. Well known parsing techniques like LR(k) [7], SLR(k) [8], LALR(k) cannot be used, as even a small change in the grammar may induce large changes in the parser. Other techniques like LL(k) [9], [10] and SLL(k) are of a very limited use: Their structure is a forest, but they are too restrictive in the form of the trees – they can fork only in their root nodes. Restrictions on the parser changes would be too strong, or productions (and the parser structure) would have to be adapted to productions with at most two symbols on their right-hand side (to support more changes). Such grammars are harder to read (understanding them requires more effort and thus using them may induce more practical troubles), so it is better if the system supports productions written ”naturally” (i.e., in the way designers think). Therefore, we do not want to be so restrictive in the grammar design.

Ch(k) grammars proposed by Nijholt and Soisalon-Soininen in [11] are quite promising in this sense. Productions for every nonterminal may create arbitrary trees. There are, however, two issues:

- re-computation of lookaheads is quite complex for $k > 1$,
- left recursion is not supported. (A production is called left recursive if it is possible to derive a string of symbols from its right-hand side, which starts with the symbol from its left-hand side. A grammar is called left recursive if it has left-recursive productions. Left recursion is often used to describe lists or arithmetic expressions. It is possible to describe them without left recursion [12] but it is less comfortable.)

The tree/forest structure and support for handling left recursion are met by kind grammars and kind parsers [13], [2]. We will describe their structure, show the local character of changes, and discuss the limitation of input language restrictions and modifications.

III. Structure of Kind Parser

We will explain the structure and behaviour of kind parsers via the "transformation" of a kind grammar to a kind parser and a few examples. In our examples a grammar G_{SE} = (N,Σ,P,S) of a simple arithmetic expression will be used.

Example 3.1 (Simple arithmetic expression):

$\mathrm{N} = \{S, E, T, F\}$

$\Sigma = \{id, num, (,), +, *\}$

$$\mathrm{P} = \left\{ \begin{array}{l} S \to E, \\ E \to E + T, E \to T, \\ T \to T * F, T \to F, \\ F \to id, F \to num, F \to (E) \end{array} \right\}$$

$\mathrm{S} = S$

In one case we will need a set of productions describing a simple command.

Example 3.2 (Productions of a simple command):

cmd → **begin** cmds **end**
cmd → id **:=** E
cmd → id (E)
cmd → **read** id
cmd → **write** E
cmd → **write** str
cmd → **while** cond **do** cmd

Let us start with dividing the productions into groups followed by their simple changes and redrawings. First, the productions are grouped according to the nonterminal on their left-hand side and according to their left recursiveness (Table I).

Then a special symbol is put to the end of each production that denotes the end of a production. These special symbols will be used to handle productions that are prefixes of some other productions and moreover to simplify the parsing process. In many parsing systems the end of a production may start *semantic actions* – activities related to a production or a parsing point. For top-down or semi-top-down parsing

Table I
Productions defining a simple arithmetic expression grouped according to the nonterminal on their left-hand side and recursiveness

Productions	
without left recursion	with left recursion
$S \to E$	
$E \to T$	$E \to E + T$
$T \to F$	$T \to T * F$
$F \to id$ $F \to num$ $F \to (E)$	

it is reasonable to allow semantic actions on arbitrary (or close to arbitrary) places within productions (Table II).

Table II
Special-symbol-terminated productions defining a simple arithmetic expression grouped according to the nonterminal on their left-hand side and recursiveness

Productions	
without left recursion	with left recursion
$S \to E \ \#S_1$	
$E \to T \ \#E_1$	$E \to E + T \ \#E_2$
$T \to F \ \#T_1$	$T \to T * F \ \#T_2$
$F \to id \ \#F_1$ $F \to num \ \#F_2$ $F \to (E) \ \#F_3$	

Then we can isolate the left-hand-side nonterminals from the productions. In the left-recursive groups the leading nonterminals can be omitted.

Table III
Modified productions

Nonterminal	right-hand sides of productions having	
	no left recursion	direct left recursion
S	$E \ \#S_1$	
E	$T \ \#E_1$	$+T \ \#E_2$
T	$F \ \#T_1$	$*F \ \#T_2$
F	$id \ \#F_1$ $num \ \#F_2$ $(E) \ \#F_3$	

After that, the productions are redrawn from the textual form to graphs with edges labelled by symbols (Figure 1).

Next, the productions are reordered so that all productions with the common prefix are together. The common prefixes can be joined to make a single path (Figure 2).

The tree structure is better visible on the example of a simple command (Figure 3).

Finally, the nodes of the trees can be decorated with lookaheads to simplify a traversal along the forest (Figure 4).

IV. Parsing Process

The work of the kind parser can be described as an input-driven traversal through the production forest – regardless of the representation of the kind parser (which can be in the

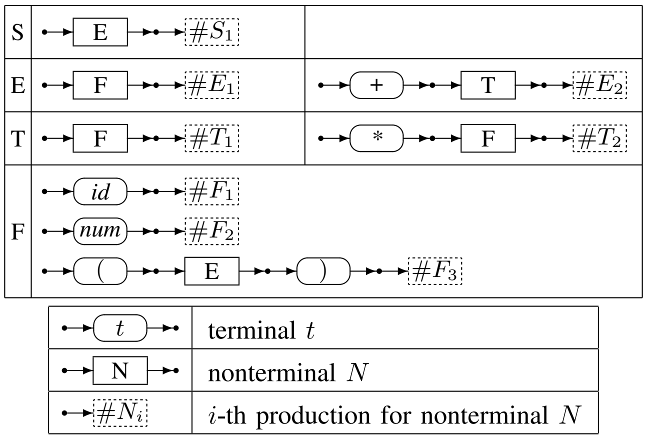

Figure 1. Productions drawn as graph

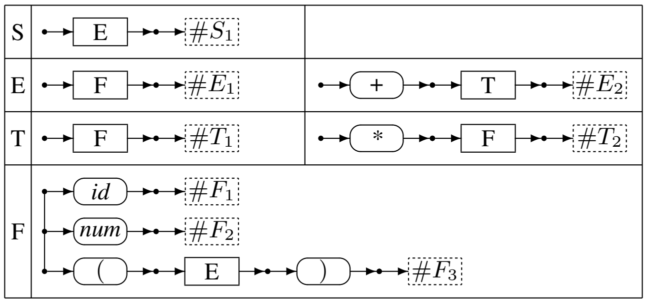

Figure 2. Productions drawn as forest

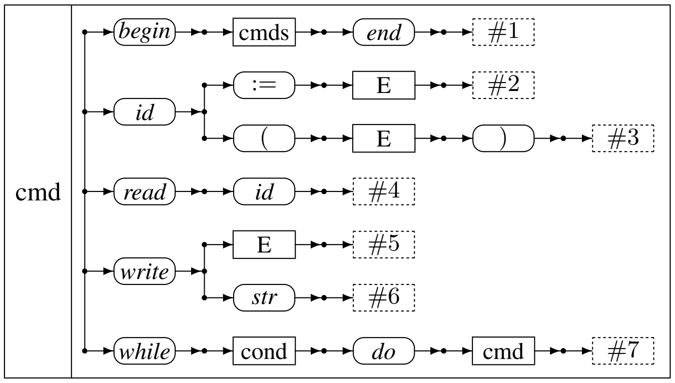

Figure 3. Simple command basic production tree

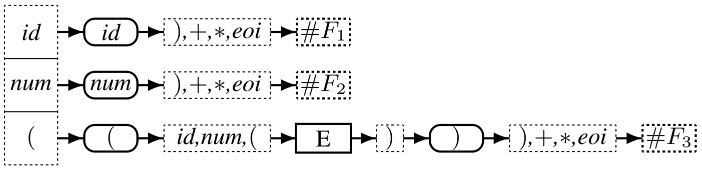

Figure 4. Production tree for nonterminal F with precomputed lookaheads

form of, e.g., a forest, a pushdown automaton, or a set of procedures forming a recursive descent parser).

1) It starts in the root of the non-recursive production tree that is associated with the starting symbol and the pushdown is in its initial setting (e.g., it contains the initial stack symbol only).
2) The lookahead is compared with the lookahead decoration of the node.
3) If there is a path corresponding to the lookahead, the corresponding edge is entered; if no such edge exists, the parsing ends with rejection of the input (or error handling is started).
4) The label type of the current edge is checked:
    a) **terminal**: The input is read and matched with the label – if they do not correspond, the input is rejected and a syntax error is reported. If full lookahead handling is used, the check is redundant. However, in case of lazy lookahead handling (lookaheads are computed for branching nodes only), it can be a necessity.
    b) **nonterminal**: The current position in the forest is put on the stack, the root of the non-recursive production tree is entered.
    c) **production**: The lookahead is checked whether it conforms with the lookahead of the root of the corresponding recursive tree. If it does, the parsing continues here. If not, the Follow set of the given nonterminal (covering only non-recursive cases) is checked. If the current lookahead does not fit, the input is rejected or a syntax error handling procedure is started. If there is a stored position on the stack, it is retrieved and the parsing continues from that parsing position. If the entire input has been read, it is accepted, otherwise only the read part of the input conforms to the input language.

The parsing process of a kind pushdown automaton is described in [14] or [2].

V. Parser Extension

A kind parser extension is quite simple: New productions are inserted into production trees in such a way that new paths are created there. It holds that all existing nodes and edges of the trees are preserved, only new edges and nodes are added. If new nonterminals are introduced, new production trees occur. Every reference to a node or an edge from the parsing stack existing before the extension is available and usable also after the extension. Every input acceptable without the extension is accepted the same way also after the extension. Hence on-line extensions of kind parsers are safe.

On-line language extensions can be of different extent (their validity may have a different span):

- permanent – from the extension forever;
- temporal – from a given point to another given point

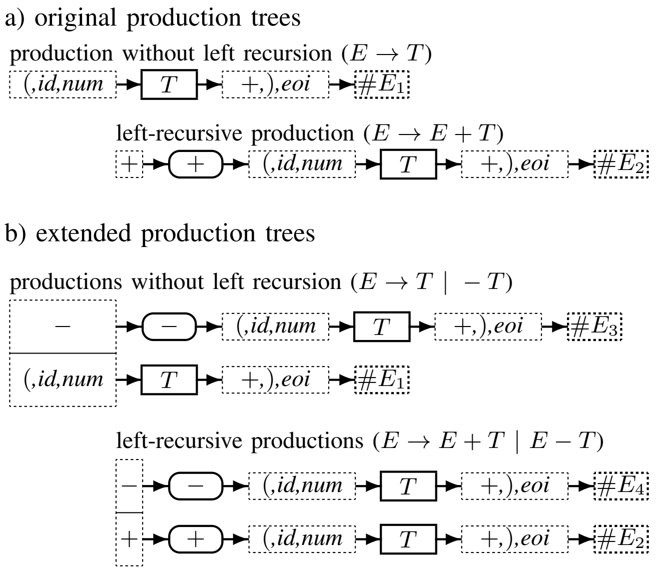
a) original production trees
production without left recursion ($E \to T$)
(,id,num
T
+,),eoi
#E_1
left-recursive production ($E \to E + T$)
+
+
(,id,num
T
+,),eoi
#E_2
b) extended production trees
productions without left recursion ($E \to T \mid -T$)
−
−
(,id,num
T
+,),eoi
#E_3
(,id,num
T
+,),eoi
#E_1
left-recursive productions ($E \to E + T \mid E - T$)
−
−
(,id,num
T
+,),eoi
#E_4
+
+
(,id,num
T
+,),eoi
#E_2

Figure 5. An extension of production trees for nonterminal E with productions $E \to -T$ and $E \to E - T$

(the validity can be, e.g., limited to a language construct);

- semi-permanent – from the extension to a parser restart.

Similarly, extension definitions may have different origin: They are either pre-defined in the language definition, or even part of the input text that is being parsed (or derived from it). A request to switch to the extended language can also be part of the input text or it is an asynchronous event generated by the application that uses the parser. The same holds for parsed language restrictions and modifications discussed below.

The design of extensible grammars may differ from the usual grammar design. It can be advantageous for later extensions to handle also grammars that are not reduced (according to, e.g., [15, p. 273] a context-free grammar G is said to be *reduced* if it contains neither inaccessible nor useless nonterminals; simply said, all symbols must be reachable from the starting symbol and it must be possible to rewrite all nonterminals to a terminal string) – see Example 5.1.

Example 5.1 (Possible use of a non-reduced grammar): Let us have a language $L = \{a^n cb^n ea^m cb^m | m, n \geq 0\} \cup \{a^n db^n ea^m db^m | m, n \geq 0\}$. The language can be generated by the semantic grammar

$$G = \left(\begin{array}{lcl} N & = & \{S, A, B\} \\ \Sigma & = & \{a, b, c, d, e\}, \\ \Phi & = & \{\sigma, \tau\}, \\ S, & & \\ P & = & \left\{ \begin{array}{l} S \to AeB \\ A \to aAb | c\sigma | d\tau \\ B \to aBb \end{array} \right\} \end{array} \right),$$

where semantic action σ adds production $B \to c$ and semantic action τ adds production $B \to d$.

Extensions also give us an elegant and clear way to express some grammar constructs or even a possibility to express context conditions at the level of the syntax analysis and still use a parser based on lookahead automata and context-free grammars.

VI. Output

There are several types of output that parsers can produce. The simplest one has mainly theoretical value – parsers just return a logical value stating whether the input word belongs to the accepted language or not.

Practical parsers produce an output that can be further used. One option is a parsing (derivation) tree or information usable for its construction – the sequence of used productions (e.g., the sequence of visited end-of-production edges of our structure).

Another possible output type is a sequence of extra symbols – semantic actions. Usually each semantic action corresponds to some activity of the application using the parser. A special case of such activity can be an extension, a reduction, or even a general modification of the parser.

Parsers can be extended to be transducers. They can transform a message in the input language to another message in the output language. In such a case another type of symbols called *output symbols* or *output terminals* are used. This kind of the parser output can correspond to SAX events known from XML processing.

VII. Parsed Language Restriction

Having an extensible tool is good. Having an opportunity to remove unnecessary constructs gives some further advantages.

In contrast to a language extension (which is safe for kind parsing), a language restriction requires to be done with some care. For an off-line language restriction usual checks performed for parser construction are sufficient. If we want to reduce the language on-line, we must be more careful, as the already parsed part of the input produced some results that should be preserved for the rest of the parsing process.

It can happen (if we are not careful enough) that a too strong restriction (removal of some parts of the parsing forest) may affect already made computations or even the opportunity to finish the computation itself successfully.

Therefore, some rules must be set, which would guarantee that even after a parser restriction the computation can be finished for a proper input (and the input accepted). We can consider these safety levels:

1) no checking;
2) postponed checking – non-correctness is detected during lookahead recomputation or even during parsing (in case of lazy lookahead handling or when a deleted symbol is popped from the stack);

3) a sequence of steps leading (for proper input) to the next syntax change or to an accepting situation must be preserved; (We use a situation instead of a configuration, as in the parsing theory a configuration means a triple (state, stack, unread part of input));
4) a sequence of steps leading (for proper input) to the final configuration must be preserved;

We can also check whether the resulting grammar is reduced or not.

The real complexity of checks of resulting grammars depends not only on the grammar size, but also on the current parser stack depth (all the symbols on the stack should be checked). The number of different stack symbols is limited by the grammar, so we can bind the complexity of the checks to the grammar size.

A tool, built according to the theory summarised in this paper, is available in [16] or [17]. It is an implementation of a modifiable kind parser that manages to extend or modify the input language at parse time, both permanently and temporarily. As a demonstration of its power, see Example 7.1. It makes possible to start a program in one language, then switch to another, go back to the first one, and so on. The whole source can be processed in a single pass.

Example 7.1 (Two languages in one source):

```
paslike

var A : array[1..10] of integer;
    n : integer;

{Quicksort procedure}
procedure QS (start, stop : integer);

var P, pom : integer;
    i, j : integer;

begin

  if  start < stop  then
    begin

      P:=(A[start] + A[stop]) div 2;
      i:=start;
      j:=stop;

      while i <= j do
        begin
          clike
            /* find items to change */
            while (A[i] < P) ++i;
            while (A[j] > P) --j;
            if (i <= j){
                pom=A[i]; A[i]=A[j]; A[j]=pom;
                ++i;
                --j;
             }
          finish
        end;

      QS(start,j);
      QS(i, stop)

    end;
end;

clike

int main(int argc,char **argv) {
        fill(A);
        QS(1, 10);
        present(A);
        return 0;
}

finish

finish
```

VIII. Conclusion

Kind parsing presented in this paper can be – as an adaptive technology – a very useful tool for adaptive system development. It is an efficient and easy to understand parsing technique that supports on-line and off-line adaptation (extensions, reductions, general changes) of the accepted language.

The on-line changes can be induced by information contained in the input that is being parsed (processed as a semantic action) or even by an asynchronous parser changing event generated by the application using the parser, i.e., the application can change the behaviour of the underlying parser processing a data stream.

The theory introduced in this paper was practically verified by implementing tools.

We suppose that such adaptive parsers can be used at least in these cases:

- parsing/compiling extensible languages,
- (adaptive) front-end gates [18],
- input parts of adaptive systems controlled from outside (other applications or adaministrators),
- input parts of (self-)adaptive systems controlled by the input text/data,
- input parts of self-adaptive systems controlled by the application logic.

The only restriction is that the cooperating parts of the system influenced by the adaptation must be adapted correspondingly.

Acknowledgment

This paper was partially supported by the Czech Science Foundation by the grant number 201/09/0983.

References

[1] M. Žemlička and J. Král, “Run-time extensible (semi-)top-down parser,” in *Proceedings of the 2nd International Workshop on Text, Speech and Dialogue (TSD-99)*, ser. LNAI, V. Matoušek, P. Mautner, J. Ocelíková, and P. Sojka, Eds., vol. 1692. Berlin: Springer, Sep. 13–17 1999, pp.

121–126. [Online]. Available: http://dx.doi.org/10.1007/3-540-48239-3_22 [Last visited: August 31, 2010]

[2] M. Žemlička, "Introduction to kind parsing," Ph.D. dissertation, Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic, Jul. 2006. [Online]. Available: http://www.ksi.mff.cuni.cz/~zemlicka/pdf/PrinciplesOfKindParsing.pdf [Last visited: August 31, 2010]

[3] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation and Compiling*. Englewood Cliffs, N.J.: Prentice-Hall, 1972, vol. I.: Parsing.

[4] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, ser. World Students Series Edition. Addison-Wesley, 1986, vol. 10194.

[5] A. Meduna, *Automata and Languages: Theory and Applications*. London: Springer, 2000.

[6] D. Grune and C. J. H. Jacobs, *Parsing Techniques*, 2nd ed., ser. Monographs in Computer Science. New York, USA: Springer, 2008. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-68954-8 [Last visited: August 31, 2010]

[7] D. E. Knuth, "On the translation of languages from left to right," *Information and Control*, vol. 8, no. 6, pp. 607–639, 1965. [Online]. Available: http://dx.doi.org/10.1016/S0019-9958(65)90426-2 [Last visited: August 31, 2010]

[8] F. DeRemer, "Simple LR(k) grammars." *Commun. ACM*, vol. 14, no. 7, pp. 453–460, 1971. [Online]. Available: http://doi.acm.org/10.1145/362619.362625 [Last visited: August 31, 2010]

[9] P. M. Lewis, II and R. E. Stearns, "Syntax directed transduction," in *Switching and Automata Theory*. IEEE, 1966, pp. 21–35. [Online]. Available: http://dx.doi.org/10.1109/SWAT.1966.26 [Last visited: August 31, 2010]

[10] D. J. Rosenkrantz and R. E. Stearns, "Properties of deterministic top-down grammars," *Information and Control*, vol. 17, no. 3, pp. 226–256, 1970. [Online]. Available: http://dx.doi.org/10.1016/S0019-9958(70)90446-8 [Last visited: August 31, 2010]

[11] A. Nijholt and E. Soisalon-Soininen, "Ch(k) grammars: A characterization of LL(k) languages." in *MFCS*, ser. Lecture Notes in Computer Science, J. Becvár, Ed., vol. 74. Springer, 1979, pp. 390–397. [Online]. Available: http://dx.doi.org/10.1007/3-540-09526-8_38 [Last visited: August 31, 2010]

[12] E. Soisalon-Soininen and E. Ukkonen, "A method for transforming grammars into $LL(k)$ form," *Acta Informatica*, vol. 12, pp. 339–369, 1970. [Online]. Available: http://dx.doi.org/10.1007/BF00268320 [Last visited: August 31, 2010]

[13] M. Žemlička and J. Král, "Run-time extensible deterministic top-down parsing," *Grammars*, vol. 2, no. 3, pp. 283–293, 1999. [Online]. Available: http://dx.doi.org/10.1023/A:1009914518458 [Last visited: August 31, 2010]

[14] J. Král and M. Žemlička, "Semi-top-down syntax analysis." in *Grammars and Automata for String Processing*, ser. Topics in Computer Mathematics, C. Martín-Vide and V. Mitrana, Eds., vol. 9. London: Taylor and Francis, 2003, pp. 77–90.

[15] R. Wilhelm and D. Maurer, *Compiler Design*, ser. International Computer Science. Wokingham, England: Addison-Wesley, 1995, vol. 42290.

[16] P. Šašek, "Rozšiřování syntaxe za běhu (in Czech: Run-time syntax extensions)," Master's thesis, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic, 2007.

[17] P. Šašek, ExTra - Extensible Transducer. [Online]. Available: http://www.ksi.mff.cuni.cz/~zemlicka/projects/download/ExTra/ [Last visited: August 31, 2010]

[18] J. Král and M. Žemlička, "Software architecture for evolving environment," in *Software Technology and Engineering Practice*, K. Kontogiannis, Y. Zou, and M. D. Penta, Eds. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 49–58. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/STEP.2005.25 [Last visited: August 31, 2010]