SSCS: A Smart Spell Checker System Implementation Using Adaptive Software Architecture

Deepak Seth[1] and Mieczyslaw M. Kokar[2]

[1] Northeastern University, Boston, MA 02115, USA,
seth@coe.neu.edu
[2] Northeastern University, Boston, MA 02115, USA,
kokar@coe.neu.edu,
WWW home page: http://www.coe.neu.edu/~kokar

Abstract. The subject of this paper is a Smart Spell Checker System (SSCS) that can adapt to a particular user by using the user's feedback for adjusting its behavior. The result of the adjustment is manifested in a different ordering of the suggestions to the user on how a particular spelling mistake should be corrected. The SSCS uses the Adaptive Software Architecture (ASA). The ASA consists of a hierarchy of layers, each containing a number of components called*Knowledge Sources*. The layers are connected by a software bus called *Domain*. External elements include *User* and *Initiator(s)*. Initiators supply input data to the system. The system also includes an *Evaluator* that generates feedback. Each Knowledge Source is responsible for generating suggestions for correcting a specific type of error. Feedback is propagated to Knowledge Sources after the user makes a selection of the correction. In response to feedback, Knowledge Sources adjust their algorithms. In this paper we present the results of the evaluation of the adaptability of the SSCS.

1 Introduction

Spell checking applications are not only common in today's marketplace, but have reached a plateau; it is often difficult to distinguish between one application and the other because they offer almost the same features and functionalities. In general, spell checking applications present valid suggestions to the user based on each mistake they encounter in the user's document. The user then either makes a selection from a list of suggestions or chooses to ignore the suggestions and accepts the current word as valid. Regardless of how often this is done, the spell checking application will perform its task independent of the types of mistakes most commonly made by that particular user.

Most spell checkers available today tend to be inflexible because their interfaces present suggestions in alphabetical order. They are tailored to the needs of the general population rather than to the mistakes of a particular individual. There is a need for adaptive interfaces that can anticipate and adapt to the specific spelling mistakes of any user [10]. A self-adapting interface [9] monitors

the user's activity and attempts to adjust its behavior automatically to be more compatible with the particular user. By monitoring the user's activity, enough useful information about the user can be elucidated and an accurate model of the user can be generated [8]. An adaptive spell checker monitors the user's mistakes and is able to determine the types of spelling mistakes the user makes. Based on these mistakes, it presents suggestions to the user based on how frequently a particular type of mistake has occurred in the past, instead of simply displaying them in alphabetical order. We believe that an adaptive spell checker makes a good case study for self-adapting software.

In many cases, there are many different ways to correct an erroneous word. The spell checker needs to decide which correction to propose to the user. For the efficiency of the spell checking process, it is important that the right suggestion is presented as a default suggestion. In such a case, the user needs only to confirm the default suggestion and proceed with the next error. Otherwise, the user needs to scroll through a list of suggestions and pick one as the right one. Even worse, often the right suggestion is not on the list and thus the user needs to type the full word again. Our Smart Spell Checker System (SSCS) attempts to adapt to the user by learning most typical mistakes made by that particular user and incorporate this knowledge into the process of selecting the default suggestion that is put on the top of the list of suggestions.

The goal of the research presented in this paper was to implement and analyze an intelligent spell checker that adapts to the user's mistakes using a new paradigm in software architecture - self-adaptive software (cf.[2–4, 8]). This architecture is designed upon the structure of an adaptive controller [2, 5, 6] that uses the feedback mechanism to induce adaptability of the system. Towards this aim, we implemented a modular and flexible program, the SSCS, which incorporates some of the ideas of adaptive control systems [1]. The modularity of the architecture, called the Adaptive Software Architecture (ASA), allows easy additions of new components into the system, such as additional knowledge sources or domains to the overall system.

To demonstrate the adaptability feature of the SSCS, we evaluated the system against a non-adaptive system. The results of these comparisons were then mapped against a theoretical scenario. Finally, we analyzed the results and drew conclusions based on this study and proposed directions for future research.

2 Adaptive Software Architecture

The basic structure of the ASA is shown in Figure 1. The main components of this architecture are Knowledge Sources (KSs), Domains, Initiator and Evaluator. Knowledge Sources are the main working modules of an application. Domains act as software buses (also known as "blackboards" [7]) and connect the Knowledge Sources, Initiators and Evaluators. KSs take data from the Domain below, but the results are placed on either the Domain below or above, depending on a particular system configuration (design).

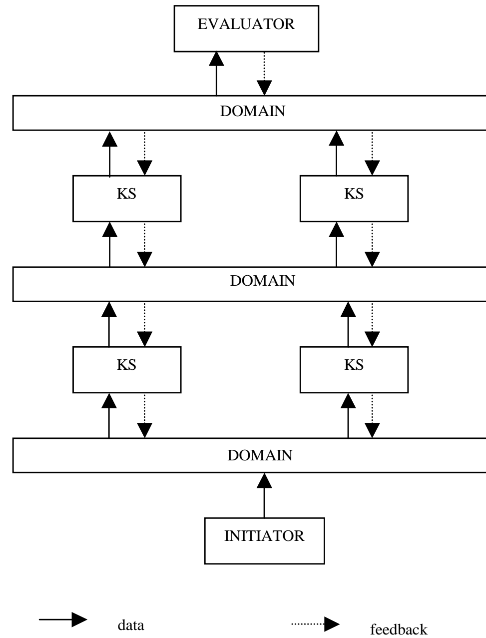

Fig. 1. Adaptive Software Architecture (ASA)

The input data are placed on the first Domain by the Initiator (the source of data) and consequently become visible to the KSs that are connected to the Domain. Each KS is equipped with a mechanism (knowledge) for making decisions on whether to process the data or not. The algorithms for data processing, decision-making, score generating and updating are also implemented in the KSs. After the first layer of KSs process the data placed on the first Domain, the results are placed either on the same Domain or on the next Domain (above), depending on the configuration of the system. If the new data are placed on the lower domain the same processing starts over. When the processing ceases, the second layer of KSs invoke their algorithms - first to make decisions on whether to process the data or not, and then their processing algorithms, if the decision is to process. When the processing reaches the highest Domain, the Evaluator assesses the quality of each of the results and gives feedback on whether the result was good or not. This feedback is then backpropagated through Domains and

KSs so that it reaches only those KSs that participated in the processing, i.e., those KSs that made the decision to process. In response to the received feedback, the KSs adjust their processing algorithms so that the next time around the processing is different, tuned towards higher feedback scores.

3 Experimental Scenario

To investigate the adaptability of the system we use the application of spell checking. The input to the system is a text file containing a list of words. The goal of the system is to recognize erroneous words and then to attempt to correct the identified erroneous words and present suggestions to the user on what word should be used instead. The suggestions presented to the user are based on the mistakes made most often in the document.

The application, called here SSCS (Smart Spell Checking System) has been implemented as an instance of the Adaptive Software Architecture (cf. [2–4]) shown in Figure 1. The structure of the SSCS is shown in Figure 2. It consists of three domains (Input, Error and Evaluation) and nine KSs. Both the detection and correction of erroneous words are implemented within the KSs. The detection of erroneous words is implemented using a Dictionary and a User Defined Dictionary. The User Defined Dictionary contains words that the user wishes to be considered, in addition to standard dictionary entries. The following Knowledge Sources were used for correcting erroneous words:

- Left-Right Character Shifter
- Character Doubler
- End Character Appender
- Character Remover
- Subsequent Character Switcher

Each KS reflects one kind of typing mistake that has its source in the domain, i.e., in typing. The Left-Right Character Shifter corrects mistakes that arise due to misplacement of the hand on the keyboard. For example the UNIX command "cd" can be erroneously typed as "vf" or "xs". The Left-Right Character Shifter then replaces the characters with their neighbors on the keyboard and makes such a suggestion to the user. The Character Doubler assumes that a character was typed once instead of twice and corrects words by repeating characters that are potentially missing. For example, the erroneous word "siting" would be corrected as "sitting". The End Character Appender corrects erroneous words that have the end character missing. For example, the erroneous word "facto" would be corrected as "factor". The Character Remover corrects an erroneous word by removing a single character to produce a correct word. For example, "networjk" would be corrected as "network". The Subsequent Character Switcher swaps two consecutive characters in an erroneous word to generate a correct word. For example, "hta" would be corrected as "hat".

Each KS has associated with it a probability given by the equation:

$$P(t) = c/N$$

where c is the number of times that the correct suggestion was at the top of the list after t attempts, and N is the number of valid suggestions presented to the user. This value is a measure of the degree of success in generating the correct words selected by the user. Every time a user selects a word generated by a particular KS, the probability factor of that KS is incremented (the count c of correct words incremented by one). This value is used in determining the order of suggestions presented to the user. Suggestions generated by KSs that have a higher probability factor will be ahead of suggestions generated by other KSs.

The suggestions generated by the KSs are presented to the Evaluator. The Evaluator compares these words against a dictionary and displays the valid suggestions to the user. The user chooses a suggestion or selects Ignore. Based on the user's choice, feedback is generated and is passed back to the KSs.

4 SSCS Operation

The Initiator reads a text file and places the extracted words from the text file, one at a time, on the Input Domain. The extracted words are examined by both the Dictionary and the User-Defined Dictionary. If a word is present in the Dictionary or the User-Defined Dictionary, it is interpreted as a valid word and the Input Domain asks for the next word. If a word in not present in either dictionary, it is forwarded to the Error Domain.

All of the Knowledge Sources (Left-Right Character Shifter, Character Doubler, End Character Appender, Character Remover and Subsequent Character Switcher) attempt to correct the erroneous word based on their algorithms. For example, for the given erroneous word, the Knowledge Sources' algorithms will generate the words shown in Table 1. Note that these are not the suggestions that the user will see since only the words that pass the dictionary comparison test will be displayed.

Table 1. An example of generated suggestions

Knowledge Source	Erroneous Word	Generated Suggestions
Left-Right Character Shifter	jod	his, pkf
Character Appender	hai	haia, haib, haic, haid, haie, haif, haig, haih haij, haik, hail, haim, haio, haip, haiq, hair hais, hait, haiu, haiv, haiw, haix, haiy, haiz
Character Doubler	ben	bben, been, benn
Character Remover	caree	aree, cree, caee, care
Subsequent Character Switcher	hta	tha, hat

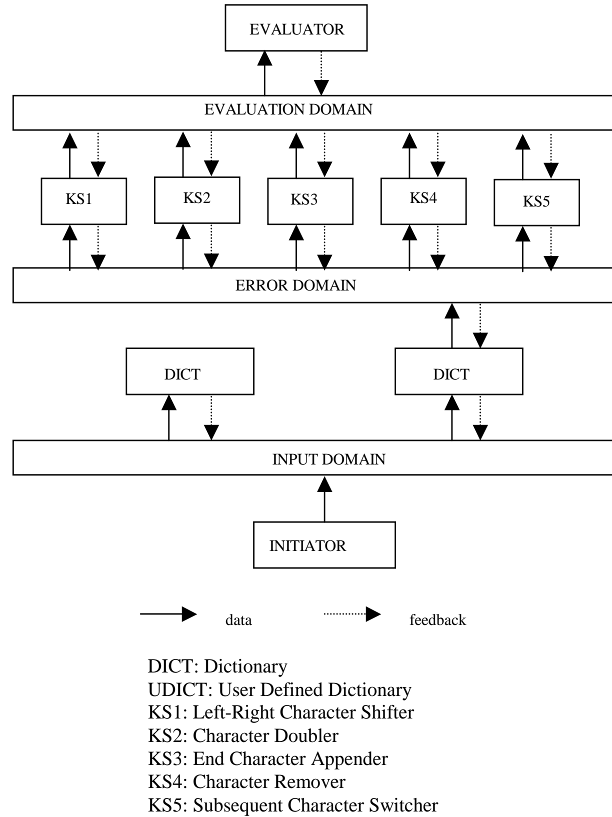

Fig. 2. Smart Spell Checker System (SSCS)

Each Knowledge Source generates the above words as suggestions and forwards them to the Evaluation Domain. Along with these suggestions, it also forwards its probability value. The Evaluation Domain receives this information, concatenates the suggestions and probability of all KSs and passes them to the Evaluator. The Evaluator receives these suggestions and compares each suggestion against the words in the dictionary. The suggestions that exist in the dictionary are displayed to the user. The order in which they appear is based on the probability associated with the particular KSs that generated the suggestions. Therefore, words associated with errors that appear more frequently will be above those that are less frequent.

The KSs receive notification in the form of a feedback from the Evaluator whether any word from their suggestion was selected or not. All inputs to and

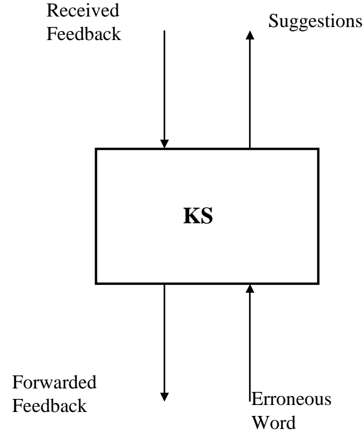

Fig. 3. Inputs and Outputs of a Knowledge Source

outputs from a KS, including feedback propagation, are shown in Figure 3. After receiving feedback, KSs update their probability values according to the following formula:

$$P(t+1) = c \pm \frac{1}{N}$$

If the suggestion is accepted by the user, the probability increases, otherwise it decreases. Note that the selection of the default suggestion is based on the relative value of the probability with respect to the probabilities of other Knowledge Sources, and not just on the absolute value.

5 Measuring Adaptability

To assess the adaptability of the SSCS, we employed a "black box" approach by sending an identical input file to two systems and comparing the outputs; the first system is the SSCS, with the adaptability mechanism, and the second system is without any adaptability mechanism. We ran the same set of tests independently on the two systems using the same input words and selecting words in the same order for both the adaptive and non-adaptive systems. Based on their respective outputs, we then calculated the probability that the correct

word was presented at the top of the list. For measurement purposes, we made the following assumptions:

– User errors can be processed by at least one of the Knowledge Sources
– Cases where user-defined words are processed were not included, since most spell checkers have a similar capability and, therefore, adaptation does not present any differentiation.

The main idea of our tests is presented in Table 2. One of our goals was to obtain results in terms of the number of suggestions that are spread over a range of numbers. This is difficult to achieve when using a randomly selected text file. We also wanted to be able to test all of the Knowledge Sources in action. Again, in a randomly selected file this is not necessarily the case. For these reasons, and keeping in mind that our real goal was to evaluate the adaptability mechanism of our architecture, we decided to generate a text file with artificially constructed words that satisfy the requirements of our test. Similarly, we pre-selected the user's choice of correct words. The test cases produced 9, 12, 15 and 18 suggestions, as shown in Table 2.

Table 2. Examples of test cases (artificial words)

Input Text	Number of Suggestions	Suggested words
bc	9	vx, nv, bbc, bcc, bca, bcz, c, b, cb
bcd	12	vxs, nvf, bbcd, bccd, bcdd, bcda, bcdz, cd, bd, bc, cbd, bdc
bcde	15	vxsw, nvfr, bbcde, bccde, bcdde, bcdee, bcdea, bcdez, cde bde, bce, bcd, cbde, bdce, bced
bcdef	18	vxswd, nvfrg, bbcdef, bccdef, bcddef, bcdeef, bcdeff, bcdefa bcdefz, cdef, bdef, bcef, bcdf, bcde, cbdef, bdcef, bcedf, bcdfe

A sample of output displays of the SSCS with an input word "bc" that generates 9 suggestions is shown in Table 3. The "*" indicates which word was selected from the list of suggestions.

The first nine suggestions were generated by the following Knowledge Sources:

"vx" and "nv"	Left/Right Character Shifter
"bbc" and "bcc"	Character Doubler
"bca" and "bcz"	End Character Appender
"c" and "b"	Character Remover
"cb"	Subsequent Character Switcher

At the first display, the user picks "bbc", which is generated by the Character Doubler. When the next time suggestions are presented to the user in the second display, suggested words generated by the Character Doubler appear before the

Table 3. Examples of displays

	First Display	Second Display	Third Display	Fourth Display
1:	nv	bbc	nv	c
2:	vx	bcc	vx	b
3:	bbc*	bca	bbc	nv
4:	bcc	bcz	bcc	vx
5:	bca	c	bca	bbc
6:	bcz	b	bcz	bcc
7:	c	nv	c*	bca
8:	b	vx*	b	bcz
9:	cb	cb	cb	cb
10:	Ignore All	Ignore All	Ignore All	Ignore All

words generated by the other KSs. The same behavior can be observed in the third and the fourth displays.

The kind of test cases as presented in this paper was used to measure the adaptability of the SSCS, and not to test its functionality. For the adaptability test cases, we fed the same word into the initiator four times, and selected a word generated by a different KS every time. Normally, this would not occur in real interactions with a spell checker, but this test case was chosen to emphasize the demonstration of the adaptability mechanism.

The results of our tests are summarized in Figure 4. This figure shows a reference curve (marked with diamonds) obtained under the assumption that a word from a list of suggestions appears on the top of the list of suggestions (as default) randomly, i.e., the probability is

$$P(N) = 1/N$$

This kind of behavior was actually observed in a system without adaptability. In Figure 4, this behavior is represented by the triangle symbols. The SSCS, on the other hand, exhibits a different behavior. The probability of a correct suggestion being on the top of the list of suggestions presented by the system to the user only slightly decreases with the number of generated suggestions. This is a desirable behavior, since it means that more knowledge sources can be added to the system without putting more burden on the user who otherwise would need to sift through a huge number of generated suggestions.

6 Conclusions and Future Research

The goal of the experiments presented in this paper was to demonstrate the usability of the idea of Adaptive Software Architecture [2–4] in constructing adaptive human-computer interfaces. As a case study, we selected the application of spell checking. The goal of the system was to pick the right word for a

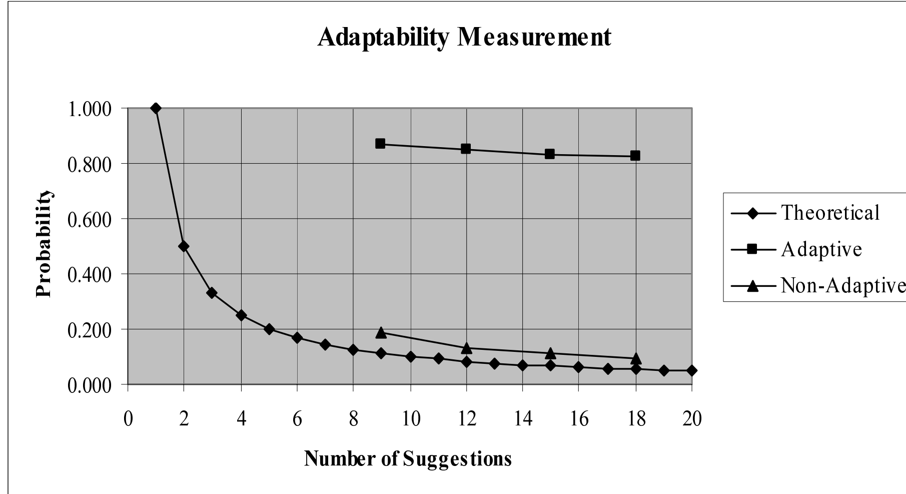

Fig. 4. Adaptability: Experimental Results

replacement of an erroneous word typed by the user and to adapt to the mistakes made by a particular user.

To achieve this goal, we mapped the ASA to the spell checking application, a Smart Spell Checking System (SSCS). Towards this goal, we had to select an instance of the architecture and populate particular components (Knowledge Sources) with domain specific knowledge. We implemented the system and tested it on various text files.

To evaluate the adaptability mechanism of the SSCS we developed a number of artificial test cases. The test cases consisted of artificial words, i.e., words that are normally not legal English words. We also generated annotations indicating which word was selected by the user as the right word. We ran this kind of tests and collected quantitative results that allowed us to develop characteristic curves of the SSCS, as shown in the paper. We ran the same experiments on the SSCS with the adaptive mechanism switched off. Additionally, we developed a theoretical performance curve of a non-adaptive system. We assumed that such a system would make its decisions randomly and thus the probability of right selection would depend only on the number of suggestions. This proved to be the behavior of our system without the adaptability mechanism.

Our adaptive system, on the other hand, performed much better and picked the right corrected word in more than 80% of the cases. Moreover, this kind of behavior was observed even with larger numbers of suggestions to pick from. The non-adaptive system's performance degraded significantly with the increased number of possible suggestions. This is a very encouraging result, since it indicates that the system can scale up to more Knowledge Sources. This feature is

very important since the five Knowledge Sources developed in this experiment would not be sufficient for a real spell checker.

While the above implementation and results proved the possibility of building an adaptive spell checker, it is clear that more research surrounding the architecture and the adaptability measurement is needed. A more accurate measure of the SSCS's adaptability is to run the system using real words and using a full-fledged dictionary. The data points obtained from such a study would represent a more conclusive result as to how adaptable the system is.

It is clear that for a real spell checker more knowledge sources would need to be added. A larger number of KSs would produce more valid suggestions to the system and consequently would result in better coverage of spelling errors. At the same time the convergence to the real error distribution of a given user would become slower. The question of what is the convergence rate would have to be answered based on results collected using real human subjects rather than artificially generated text.

In order to make a decision on whether to use such an architecture in a real system one would need to first decide what should be the usability measure and then conduct more intensive studies to assess the impact of this kind of adaptability on the selected usability metric. One candidate for such a measure could be the time spent for correcting one error. This metric would tell what is the value added in terms of the productivity of people using this kind of spell checker. However, the goal of the study presented in this paper was merely to provide an indication of the adaptability of the proposed architecture. The spell checking application was just a case study to achieve this goal. Human factor studies were not within the scope of this research.

Acknowledgments

The authors would like to thank the members of the Software Engineering Project class for their work on the Adaptive Software Architecture: Bob Bamberg, Prasad Bandaru, Jianqing Huang, Amir Kompany, Tamnun Mursalin, Madhavi Narla, Firozur Rahman and Charles Tan.

References

1. K. J. Åström. *Adaptive Control*. Addison-Wesley, Reading, MA, 1989.
2. Y. A. Eracar. Raacr: A reconfigurable architecture for adapting to changes in the requirements. Master's thesis, Northeastern University, Boston, MA, 1996.
3. Y. A. Eracar and M. M. Kokar. An architecture for software that adapts to changes in requirements. *Journal of Systems and Software*, 50:209–219, 2000.
4. M. M. Kokar, K. Baclawski, and Y. Eracar. Control theory-based foundations of self-controlling software. *IEEE Intelligent Systems*, May/June 1999:37–45, 1999.
5. M. M. Kokar, K. M. Passino, K. Baclawski, and J. E. Smith. Mapping an application to a control architecture: Specification of the problem. *Lecture Notes in Computer Science*, 1936:75–89, 2001.

6. I. Mareels and J. W. Polderman. *Adaptive Systems: An Introduction*. Birkhauser, Boston, MA, 1996.
7. H. P. Nii. Blackboard systems. *AI Magazine*, (7(4)):82 – 107, 1986.
8. P. Robertson, H. Shrobe, and R. Laddaga. *Self-Adaptive Software. Lecture Notes in Computer Science, Volume 1936*. Springer-Verlag, 2001.
9. M. Schneider-Hufschmidt, T. Kuhme, and U. Malinowski. *Adaptive User Interfaces: Principles and Practices*. North-Holland, Amsterdam, The Netherlands, 1993.
10. K. P. Vaubel and C. F. Gettys. Inferring user expertise for adaptive interfaces. *Human Computer Interaction*, 5:95–117, 1990.