How to Speed Up Software Migration and Modernization: Successful Strategies Developed by Precisiating Expert Knowledge

Francisco Zapata[1], Octavio Lerma[2], Leobardo Valera[2], and Vladik Kreinovich[1,2]
[1]Department of Computer Science, [2]Computational Science Program
University of Texas at El Paso, 500 W. University, El Paso, Texas 79968, USA
Emails: fazg74@gmail.com, lolerma@episd.org, leobardovalera@gmail.com, vladik@utep.edu

***Abstract*—Computers are getting faster and faster; the operating systems are getting more sophisticated. Often, these improvements necessitate that we migrate existing software to the new platform. In an ideal world, the migrated software should run perfectly well on a new platform; however, in reality, when we try that, thousands of errors appear, errors that need correcting. As a result, software migration is usually a very time-consuming process. A natural way to speed up this process is to take into account that errors naturally fall into different categories, and often, a common correction can be applied to all error from a given category. To efficiently use this idea, it is desirable to estimate the number of errors of different types. In this paper, we show how imprecise expert knowledge about such errors can be used to produce very realistic estimates.**

I. Introduction: Software Migration and Modernization is Important but Difficult

Computers are ubiquitous. Computers are ubiquitous. In many aspects of our daily life, we rely on computer systems: computer systems record and maintain the student grades, computer systems handle our salaries, computer systems record and maintain our medical records, computer systems take care of records about the city streets, computer systems regulate where the planes fly, etc.

Most of these systems have been successfully used for years and decades – and this is what every user wants, to have a computer system that, once implemented, can effectively run for a long time, without a need for time- and effort-consuming maintenance.

Need for software migration and modernization. No matter how successful a computer system, the time comes when there is a need to modernize it.

The main reason for such a need comes from the fact that computer systems operate in a certain environment; they are designed:

- for a certain computer hardware – e.g., with support for operations with data pieces ("words") of certain length,
- for a certain operating system,
- for a certain programming language,
- for a certain computer interface, etc.

Eventually, the computer hardware is replaced by a new one. While all the efforts are made to make the new hardware compatible with the old code, there are limits to that: every hardware or software feature that makes possible the use of old software inevitably slows down the new system and thus makes its use not as convenient for new users. Every computer upgrade requires a trade-off to balance the interest of old and new users. As a result, after some time, not all the features of the old system are supported. In such situations, it is necessary to adjust the software so that it will work on a new system. This process is called *software migration* or, alternatively, *software modernization*, and the software that needs such a migration is called *legacy software*; see, e.g., [5], [6], [9], [10], [11].

Software migration and modernization is difficult. At first glance, software migration and modernization sounds like a reasonably simple task. Indeed, the main intellectual challenge of software design is usually when we have to invent new algorithms, new techniques – because the previous techniques cannot solve the practical problem; in software migration and modernization, these techniques have already been invented.

However, anyone who has ever tried to upgrade a legacy system knows that it is not as easy as it may sound at first glance. It may have been easier if every single operation from the legacy code was clearly explained and justified. This abundance of explanatory comments is what we all strive for, but the actual software is far from this ideal. There is a strong competition between software companies; whoever releases the product first has a great advantage, and whoever is last risks bankruptcy. In such an environment, there is no time to properly document all the code. Moreover, comments are sometimes obscured or even deleted on purpose, so that competitors would not learn about the ideas that make this code efficient.

In search for efficiency, many "tricks" are added by programmers that take into account specific hardware, specific operating system – and when the hardware and/or operating system changes, these tricks can slow the system down instead of making it run more efficiently. For example:

- some old image processing systems utilized the existence of hardware supported operations with long inputs;
- in newer RISC systems, with limited number of hardware supported operations, processing of large

inputs is no longer hardware supported, and thus, the resulting software becomes very slow.

People who need to migrate the legacy code do not know which parts of the code contain such tricks.

A typical legacy code is huge: the corresponding system has a million or more lines. As a result, when a user tries to run an old legacy code on a new system, the compiler will produce an astronomical number of error messages: 5,000 or even 10,000 is a typical number.

How migration and modernization are usually done. Usually, migration is done the hard way: a software developer looks into each and every error message, tries to understand what is the problem, and come up with a correction. This is a very slow and very expensive process: correcting each error can take hours, and the resulting salary expenses can run to millions of dollars.

There exist tools that try to automate this process by speeding up the correction of each individual error. These tools definitely help, they speed up the required time by a factor of two, three, or even ten, but still thousands of errors have to be handled individually.

Resulting problem: need to speed up migration and modernization. Since migration and modernization of legacy software is a ubiquitous problem, it is desirable to come up with ways to speed up this process.

In this paper, we propose such an idea, and we show how expert knowledge can help in implementing this idea.

II. Our Main Idea: Description and Challenges

Main idea. Our main idea is based on the fact that modern compilers do not simply indicate that there is an error, they usually provide a reasonably understandable description of the type of an error. For example:

- it may be that a program is trying to divide by zero,
- it may be that a program is trying to access an element of an array with a computed index which is beyond the original bounds, etc.

Some of these types of error appear in numerous places in the software. Our experience shows that in many such places, these errors are caused by the same problem in the code. So, instead of trying to "rack our brains" over each individual error, a better idea is to look at all the errors of the given type, and come up with a solution that would automatically eliminate the vast majority of these errors.

This idea is actually natural. Judging by the current practice, this idea sounds innovative in software migration. However, if one looks at it from the general viewpoint, one can see that this from the viewpoint of a general algorithmic development, this is a very general idea.

This is how most algorithms originated; let us give a few historical examples. In many case, people wanted to know what will happen if we merge two groups of objects together. If we have a group of 20 sheep and we merge it with a group of 12 sheep, how many sheep will we have? While sheep herders were solving this type of problems, cow herders were solving similar problems about cows: what if we merge a group of 20 cows and a group of 12 cows, how many cows will there be total? Later on they discovered that it is possible to find an algorithm that would add objects no matter whether they are sheep, cows, or plates.

Similarly in our case, instead of dealing with individual errors, we try to come up with a general approach that would enable us to handle all (or at last almost all) errors of a given type.

This idea only works if we have sufficiently many errors of a given type. Of course, this idea saves time only if we have enough errors of a given type. For example, if we only have two or three errors of some type, it is probably faster to eliminate these few errors one by one than to try to come up with a general solution that would include all these errors.

How many errors of different type there are? Need for an expert knowledge. To successfully implement this approach, we therefore need to be able to predict how many errors of different type we will encounter.

To the best of our knowledge, there are currently no well-justified software models that can predict these numbers. What we do have is many system developers who have an experience in migrating and modernizing software. It is therefore desirable to utilize their experience.

Since experts usually describe their experience not in precise terms, but by using imprecise ("fuzzy") words from natural language, it is reasonable to use the known precisiation techniques to transform this expert knowledge into precise terms – in particular, techniques developed in fuzzy logic; see, e.g., [2], [8], [12].

III. How Many Errors of Different Types There Are? An Automated Answer to This Question

How to find possible types of errors: general idea. In order to apply the proposed approach, we need to first know the possible types of errors.

The type of each error is usually determined by the compiler. So, all we have to do is to run the compiler on the migrated software, and record all error messages that are generated. Extracting all such error messages is a straightforward text processing problems, which can be easily done by many existing text processing programs.

How to find possible types of errors: example. For example, under Unix, we can catch all error messages in a single file. This can be done, e.g., by the following straightforward Unix command:

```
gmake -f sourcefile.mak > compile.log
```

This command sends all the error messages – which by default would otherwise go to the screen – into the specially created file compile.log.

In C compilers, error messages usually start with keywords like `error` or `warning`. The first line of each error message

contains this keyword followed by the type of the corresponding error. For example, this line can contain the following statement:

```
warning #2940-D: missing return statement
   at end of non-void function
```

As a result, the list of possible types of errors can be extracted from the `complile.log` file by using the following Unix command:

```
grep " warning " complile.log |
  sort -u list > categories.txt
```

The `grep` command finds all the lines that contain the word `warning` in the file `compile.log`. The resulting list of lines is then sorted *uniquely* (i.e., duplicate lines are avoided), and the sorted list is placed into the new file `categories.txt`.

How many errors of each type do we have? Now that we know the most frequent types of errors we can use the standard text processing programs to count how many errors of each type we have.

Example. For example, we can upload the file with error messages into a spreadsheet program (e.g., into Excel) and use its counting features.

IV. Expert Knowledge about Software Migration and Modernization and Its Precisiation

What we are trying to describe. Once we know how many errors of different types are there, a reasonable idea is to start with the errors of the most frequent type. Once we have learned how to deal with these errors, we should concentrate on errors of the second most frequent type, etc. After a few iterations, when all frequently repeated errors are eliminated, we reach a stage on which for each remaining type, there are so few errors of this type that it is easier to deal with these errors one by one.

From this viewpoint, what we would like to describe is how many errors there are of different types. We would like to know the number of errors n_1 of the most frequent type, the number of errors n_2 of the second most frequent type, etc. In general, we want to know the numbers n_1, n_2, ..., for which

$$n_1 \geq n_2 \geq \ldots \geq n_{k-1} \geq n_k \geq n_{k+1} \geq \ldots$$

Available expert knowledge. We know that for every level k, the number of errors n_{k+1} of the next level is somewhat smaller than the number of errors n_k of a given type.

Similarly, if we compare the number of errors n_k of a given type and the number of errors n_{k+2} of the level $k+2$, then we can also say that n_{k+2} is smaller than n_k – and that the difference between n_k and n_{k+2} is larger than the difference between n_k and n_{k+1}. We can make similar statements about the relation between n_k and n_{k+3}, etc.

How can we precisiate this idea – first approximation: idea. Let us start with the rule that n_{k+1} is somewhat smaller than n_k. By using the usual fuzzy control methodology:

- We first formulate what it means for two given values of n_k and n_{k+1} to be consistent with this rule. In fuzzy logic, this is obtained by describing, for every two possible values n_k and n_{k+1}, the degree to which the pair of n_{k+1} and n_k is consistent with this rule.
- For a given n_k, we then apply a defuzzification procedure and get a single estimate for n_{k+1} as a function of n_k:

$$n_{k+1} = f(n_k). \tag{1}$$

Comments.

- The exact form of the function $f(n)$ function depends on which membership function we used to describe the imprecise term "somewhat smaller" and on what defuzzification procedure we use.
- In principle, in addition to using the standard fuzzy methodology, we can use any other appropriate techniques to precisiate the dependence of n_{k+1} and n_k. For example, we can use interval-valued fuzzy techniques which often lead to a more accurate description of expert knowledge; see, e.g., [3], [4], [7].

Which function $f(n)$ should we select? To select an appropriate function $f(n)$, let us take into account that in many cases, a software package that needs migration consists of two (or more) parts. Because of this, we can estimate the number of errors of type $k+1$ in two different ways:

- We can use the overall number $n_k = n_k^{(1)} + n_k^{(2)}$ of k-th type errors in both parts to predict the overall number n_{k+1} of the $(k+1)$-th type errors. In this case, we get the following estimate:

$$n_{k+1} \approx f(n_k) = f(n_k^{(1)} + n_k^{(2)}).$$

- Alternatively, we can start with the numbers of errors $n_k^{(1)}$ and $n_k^{(2)}$ in each part, predict the values $n_{k+1}^{(1)}$ and $n_{k+1}^{(2)}$, and then add up these predictions. As a result, we get the following estimate:

$$n_{k+1} \approx f(n_k^{(1)}) + f(n_k^{(2)}).$$

It is reasonable to require that these two approaches lead to the same estimate, i.e., that we have

$$f(n_k^{(1)} + n_k^{(2)}) = f(n_k^{(1)}) + f(n_k^{(2)})$$

for all possible values of $n_k^{(1)}$ and $n_k^{(2)}$. In other words, for any two natural numbers a and b, we should have

$$f(a+b) = f(a) + f(b). \tag{2}$$

For $a = b = 0$, the formula (2) implies that $f(0) = 2f(0)$ and thus, $f(0) = 0$. For any other integer a, this formula implies that

$$f(a) = f(1) + \ldots + f(1) \quad (a \text{ times}),$$

i.e., that

$$f(a) = c \cdot a, \tag{3}$$

where we denoted $c \stackrel{\text{def}}{=} f(1)$. Thus, the most appropriate function $f(n)$ is a linear function $f(n) = c \cdot n$.

Resulting dependence n_k. Substituting the linear function $f(n) = c \cdot n$ into the equation (1), we conclude that $n_{k+1} = c \cdot n_k$. Thus, $n_2 = c \cdot n_1$, $n_3 = c \cdot n_2 = c^2 \cdot n_1$, etc. One can easily see that for a general k, we get

$$n_k = n_1 \cdot c^{k-1},$$

i.e., that $n_k = \dfrac{n_1}{c} \cdot c^k$ and thus,

$$n_k = A \cdot \exp(-b \cdot k), \tag{4}$$

where $A \stackrel{\text{def}}{=} \dfrac{n_1}{c}$ and $b \stackrel{\text{def}}{=} -\ln(c)$.

How accurate is this estimate? To check how accurate is this estimate, we compared it with the actual number of errors of different types obtained when migrating a health-related C-based software package from a 32-bit to a 64-bit architecture.

In the following table, the number of errors of type k =ab is stored:

- in the a-th column (which is marked ax),
- in its b-th row (which is marked marked xb).

For example, the number of errors of type $k = 23$ is stored:

- in the 2-nd column (which is marked 2x),
- in its 3-rd row (which is marked x3).

	0x	1x	2x	3x	4x	5x	6x	7x
x0	–	308	95	47	13	5	2	1
x1	7682	301	91	38	13	4	2	1
x2	4757	266	85	34	12	4	2	1
x3	3574	261	81	34	12	4	2	1
x4	2473	241	76	30	11	3	2	1
x5	2157	240	69	24	9	3	2	1
x6	956	236	58	21	8	3	2	1
x7	769	171	57	19	8	3	1	1
x8	565	156	50	17	8	2	1	1
x9	436	98	47	17	6	2	1	–

One can easily see that for $k \le 9$, we indeed have $n_{k+1} \approx c \cdot n_k$, with $c \approx 0.65$-0.75. Thus, the above simple rule described the most frequent errors reasonably accurately.

However, starting with $k = 10$, the ratio n_{k+1}/n_k becomes much closer to 1. Thus, the one-rule estimate is no longer a good estimate.

Let us use two rules: an idea. We have just mentioned that if we only use one expert rule, we do not get a very good estimate for n_k. A natural idea is this to use two rules:

- in addition to the rule that n_{k+1} is somewhat smaller than n_k,
- let us also use the rule that n_{k+2} is more noticeably smaller than n_k.

In this case, once we know n_k and n_{k+1}, we can use the standard fuzzy methodology (or any other appropriate methodology) and get an estimate

$$n_{k+2} = f(n_k, n_{k+1}).$$

Which function $f(n_k, n_{k+1})$ should we use? Similarly to the one-rule case, once we take into account that the software package consists of two parts, we can estimate the number of errors of type $k+2$ in two different ways:

- We can use the overall numbers $n_k = n_k^{(1)} + n_k^{(2)}$ and $n_{k+1} = n_{k+1}^{(1)} + n_{k+1}^{(2)}$ of k-th and $(k+1)$-th type type errors in both parts to predict the overall number n_{k+2} of the $(k+2)$-th type errors. In this case, we get the following estimate:

  $$n_{k+2} \approx f(n_k, n_{k+1}) = f(n_k^{(1)} + n_k^{(2)}, n_{k+1}^{(1)} + n_{k+1}^{(2)}).$$

- Alternatively, we can start with the numbers of errors $n_k^{(p)}$ and $n_{k+1}^{(p)}$ in each part p, predict the values $n_{k+2}^{(1)}$ and $n_{k+2}^{(2)}$, and then add up these predictions. As a result, we get the following estimate:

  $$n_{k+2} \approx f(n_k^{(1)}, n_{k+1}^{(1)}) + f(n_k^{(2)}, n_{k+1}^{(2)}).$$

It is reasonable to require that these two approaches lead to the same estimate, i.e., that we have

$$f(n_k^{(1)} + n_k^{(2)}, n_{k+1}^{(1)} + n_{k+1}^{(2)}) = f(n_k^{(1)}, n_{k+1}^{(1)}) + f(n_k^{(2)}, n_{k+1}^{(2)})$$

for all possible values of $n_k^{(1)}$, $n_{k+1}^{(1)}$, $n_k^{(2)}$, and $n_{k+1}^{(2)}$. In other words, for any two four numbers $a \ge a'$ and $b \ge b'$, we should have

$$f(a + b, a' + b') = f(a, a') + f(b, b'). \tag{5}$$

Let us solve the corresponding functional equation. We want to find the value $f(x, y)$ for all $x \ge y$. By taking $a = a' = y$, $b = x - y$, and $b' = 0$, we conclude that

$$f(x, y) = f(y, y) + f(x - y, 0). \tag{6}$$

From the same formula (5), we can now conclude that

$$f(y, y) = f(1, 1) + \ldots + f(1, 1) \quad (y \text{ times}),$$

i.e., that

$$f(y, y) = c_1 \cdot y \tag{7}$$

for a real number $c_1 \stackrel{\text{def}}{=} f(1, 1)$.

Similarly, from the property (5), we conclude that

$$f(z, 0) = f(1, 0) + \ldots + f(1, 0) \quad (z \text{ times}),$$

i.e., that

$$f(z, 0) = c_2 \cdot z, \tag{8}$$

where $c_2 \stackrel{\text{def}}{=} f(1, 0)$.

Substituting the expression (7) and (8) into the formula (6), we conclude that

$$f(x,y) = c_1 \cdot y + c_2 \cdot (x - y) = c_2 \cdot x + (c_1 - c_2) \cdot y.$$

In other words, we conclude that $f(x,y)$ is a linear function of x and y. Thus, we have

$$n_{k+2} = a \cdot n_k + b \cdot n_{k+1} \tag{9}$$

for some constants $a \stackrel{\text{def}}{=} c_1 - c_2$ and $b \stackrel{\text{def}}{=} c_1$.

Resulting dependence n_k. Let us use the difference equation (9) to find the dependence of n_k on k. A general solution to a difference equation with constant coefficients is well known (see, e.g., [1]). In general, this solution is a linear combination of the expressions ρ^k, where ρ is a solution (real or complex) of the polynomial equation that is obtained when we plug in ρ_k into the corresponding difference equation. If the equation has a double or triple solution, then we can also consider the terms $k \cdot \rho^k$, $k^2 \cdot \rho^k$, etc.

In our case, substituting $n_k = \rho^k$ into the equation (9) and dividing both sides of the resulting equality by ρ^k, we conclude that

$$\rho^2 = a + b \cdot \rho. \tag{10}$$

This is a quadratic equation, and a quadratic equation either has two different real roots, or a single double real root, or it has complex conjugate roots.

For complex-conjugate roots ρ and ρ^*, the corresponding dependence has the following form:

$$n_k = A_1 \cdot \rho^k + A_2 \cdot (\rho^*)^k = A_1 \cdot (z \cdot k) + A_2 \cdot (z^* \cdot k) =$$

$$\text{const} \cdot \exp(p \cdot k) \cdot \cos(q \cdot k) + \text{const} \cdot \exp(p \cdot k) \cdot \sin(q \cdot k),$$

where $p + q \cdot \mathrm{i} = \ln(a + b \cdot \mathrm{i})$. This dependence contains trigonometric terms and is, thus, oscillating – and we want to a dependence for which always $n_k \geq n_{k+1}$.

So, in our case, the case of complex roots can be excluded, and we are left with situations in which we either have two different real roots, or one double real root. So, we have either

$$n_k = A_1 \cdot \rho_1^k + A_2 \cdot \rho_2^k, \tag{11}$$

or

$$n_k = A_1 \cdot \rho_1^k + A_2 \cdot k \cdot \rho_1^k, \tag{12}$$

for some values c_i and ρ_i. In other words, we have either

$$n_k = A_1 \cdot \exp(-b_1 \cdot k) + A_2 \cdot \exp(-b_2 \cdot k), \tag{13}$$

or

$$n_k = A_1 \cdot \exp(-b_1 \cdot k) + A_2 \cdot k \cdot \exp(-b_1 \cdot k), \tag{14}$$

where $b_i \stackrel{\text{def}}{=} -\ln(\rho_i)$.

With this new model, we get a much better fit with the data. Which of the models (13) and (14) is the best fit for the above data? One can see that the degenerate model (14) is close to exponential and thus, is not a good fit for the above experimental data.

So, we need to consider a general model (13). In this case, the values b_i are different. Thus, without losing generality, we can assume that $b_1 < b_2$. So, the desired estimate n_k is the sum of two terms:

- a slower-decreasing term $A_1 \cdot \exp(-b_1 \cdot k)$, and
- a faster-decreasing term $A_2 \cdot \exp(-b_2 \cdot k)$.

Under this assumption, what is the relation between the values A_1 and A_2?

If $A_1 > A_2$, then:

- for $k = 1$, the first term is larger, and
- since the second term decreases faster, the first term dominates for all k.

In this case, the expression (13) is close to an exponential function $A_1 \cdot \exp(-b_1 \cdot k)$, and we already know that an exponential function is not a good description of n_k.

Thus, to fit the empirical data, we must use models with $A_1 < A_2$. In this case:

- for small k, the second – feaster-decreasing – term $A_2 \cdot \exp(-b_2 \cdot k)$ dominates;
- however, since the second term decreases exponentially faster than the first one, for larger k, the first – slower-decreasing – term $A_1 \cdot \exp(-b_1 \cdot k)$ dominates.

Thus:

- for small k, we have $n_k \approx A_2 \cdot \exp(-b_2 \cdot k)$;
- for larger k, we have $n_k \approx A_1 \cdot \exp(-b_1 \cdot k)$.

In effect, we here have *two* exponential models:

- the first model works for small k, while
- the second model works for large k.

This double-exponential model indeed describes the above data reasonably accurately:

- for $k \leq 9$, as we have mentioned, the data is a good fit with an an exponential model for which $\rho = n_{k+1}/n_k \approx 0.65$–$0.75$;
- for $k \geq 10$, the data is a good fit with another exponential model, for which $\rho^{10} \approx 2$–3.

Practical consequences. Since for small k, the dependence n_k rapidly decreases with k, the values n_k corresponding to small k constitute the vast majority of all the errors. In the above example, 85 percent of errors are of the first 10 types. Thus, once we learn to repair errors of these type, the remaining number of un-corrected errors decreases by a factor of seven. This observation has indeed led to a significant speed-up of software migration and modernization.

V. Conclusion

In many practical situations, we need to migrate legacy software to a new hardware and system environment. Usually, if we simply run the existing software packages in the new environment, we encounter thousands of difficult-to-correct errors. As a result, software migration is very time-consuming.

A reasonable way to speed up this process is to take into account that errors can be naturally classified into categories, and often all the errors of the same category can be corrected by a single correction.

Coming up with such a joint correction is also somewhat time-consuming; the corresponding additional time pays off only if we have sufficiently many errors of this category. So, to plan when to use this idea, it is desirable to be able to estimate the number of errors n_k of different categories k. In this paper, we show that an appropriate use of expert knowledge leads to a double-exponential model (13) that is in good accordance with the observations.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation grants HRD-0734825 and HRD-1242122 (Cyber-ShARE Center of Excellence) and DUE-0926721.

The authors are greatly thankful to the anonymous referees for valuable suggestions.

REFERENCES

[1] Th. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 2009.

[2] G. J. Klir and B. Yuan, *Fuzzy Sets and Fuzzy Logic: Theory and Applications*, Prentice Hall, Upper Saddle River, New Jersey, 1995.

[3] J. M. Mendel, *Uncertain Rule-Based Fuzzy Logic Systems: Introduction and New Directions*, Prentice-Hall, Upper Saddle River, New Jersey, 2001.

[4] J. M. Mendel and D. Wu, *Perceptual Computing: Aiding People in Making Subjective Judgments*, IEEE Press and Wiley, Piscataway, New Jersey, 2010.

[5] A. Menychtas, K. Konstanteli, J. Alonso, L. Orue-Echevarria, J. Gorronogoitia, G. Kousiouris, C. Santzaridou, H. Bruneliere, B. Pellens, P. Stuer, O. Strauss, T. Senkova, T. Varvarigou, "Software modernization and cloudification using the ARTIST migration methodology and framework", *Scalable Computing: Practice and Experience*, 2014, Vol. 15, No. 2, pp. 131–152.

[6] A. Menychtas, C. Santzaridou, G. Kousiouris, T. Varvarigou, J. Gorronogoitia, O. Strauss, T. Senkova, L. Orue-Echevarria, J. Alonso, H. Bruneliere, B. Pellens, P. Stuer, "ARTIST methodology and framework: a novel approach for the migration of legacy software on the cloud", *Proceedings of the 15th IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing SYNASC'2013*, Timisoara, Romania, September 23–26, 2013, pp. 424–431.

[7] H. T. Nguyen, V. Kreinovich, and Q. Zuo, "Interval-valued degrees of belief: applications of interval computations to expert systems and intelligent control", *International Journal of Uncertainty, Fuzziness, and Knowledge-Based Systems (IJUFKS)*, 1997, Vol. 5, No. 3, pp. 317–358.

[8] H. T. Nguyen and E. A. Walker, *A First Course in Fuzzy Logic*, Chapman and Hall/CRC, Boca Raton, Florida, 2006.

[9] I. Sahin and F. Zahedi, "Policy analysis for warranty, maintenance, and upgrade of software systems", *Journal of Software Maintenance: Research and Practice*, 2001, Vol. 13, pp. 469–493.

[10] R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*, Addison-Wesley, Boston, Massachusetts, 2003.

[11] I. Warren and J. Ransom, "Renaissance: a method to support software system evolution", *Proceedings of the 26th Annual International Computer Software and Applications Conference COMPSAC'2002*, Oxford, UK, August 26–29, 2002, pp. 415–420.

[12] L. A. Zadeh, "Fuzzy sets", *Information and Control*, 1965, Vol. 8, pp. 338–353.