Methodology to Define Software in a Deterministic Manner

Fumio Negoro

The Institute of Computer Based Software Methodology and Technology

3-11-3 Takanawa, Minato-ku, Tokyo 108-0074, Japan

f-negoro@lyee.co.jp

Abstract

Our intelligence (science or a thoughts) is established on a basis of what is memorized. Since memory is always ambiguous, intelligence can be established only ambiguously. The reason why memory is ambiguous is that the way memory is formed is ambiguous. In our methodology, we consider that the cause of formation of memory is Intention and try to explore the structure of Intention. By doing this, we can get around the ambiguity. For this purpose, we have made hypothesis and made it into an axiomatic system and deduced this methodology for software development from the axioms.

Keywords: deterministic manner, intention, requirement, software structure

1 Introduction

This paper discusses a methodology to define computer-based software (herein after called "software") in a deterministic manner. In order to do so, we have created a "metaphysical model". A conventional way uses a model derived from empirical knowledge, which have generally been called a theoretical model. However, from our viewpoint we do not call such a model a theoretical model. That is, a model that is defined by theorems and thoughts derived from empirical knowledge is not theoretical. We regard a model that can be defined only by the theorems established in the metaphysical model as theoretical. Our methodology results in definitions of the process to capture Intention, not Requirement. Therefore, if you work based on our methodology, you will get one result that you have defined software to capture Intention. The process herein means a thinking process in the human mind to define the software. Requirement herein is objects that are represented by sentences. The sentences include mathematical and logical notations. Intention herein is defined as a cause in the human mind to create the sentence, which we cannot cognize at all. The metaphysical model that we have created is a model for capturing Intention, whereas the conventional models are created from Requirement. This tells the difference between our methodology and conventional ways. The metaphysical model is divided into two types. One is a model to establish an axiomatic system and then define Intention with the axiomatic system, and the other is a model to realize the Intention. The former is called Consciousness Model (CSM) and the latter is called Three-dimension-like Space Model (TDM). Discussion of the model for defining Intention axiomatically is very important although it is out of scope of this paper. TDM can be expressed in any programming language. This means that TDM can be processed on a computer. TDM expressed in a programming language is called Scenario Function (SF). The purpose of this paper is to discuss conceptually how SF is defined.

2 Relation between Requirement and Scenario Function

A relation between Requirement and Intention is replaced with that of Requirement and SF. In conventional ways, a program deduced from Requirement logically represents a dynamic state of Requirement. In this connection, from our viewpoint, the reason why productivity of software development and maintainablity cannot be improved lies in our instinctive way of thinking which forces us to deal with the dynamic state. Therefore, such issue of software cannot be resolved simply with engineering. Intention can be expressed as a static state with TDM. In other words, the axiomatic system makes TDM to express Intention as a static state. That is, SF is a program representing Intention in a static state, and the mode of SF is different before execution, i.e. a state defined by a human (user, SE, programmer, etc.) and during the execution on a computer. Such difference does not appear in conventional programs. In order to capture Intention, we need some kind of cognition. Such cognition exists as a basis of Requirement.

Let us discuss our concept of software for you to understand our methodology. Software consists of people (those who express Requirement, users of the software and developers of the software) the defined and the process logic. The defined is layout of screens and printouts, for example. The process logic consists of the logic to control the processing and user's logic. The user's logic is taken into software but nothing to do with software intrinsically. This point is very important to understand our methodology. The logic to control processing does not need to be defined when SF is defined by developers because SF is deduced from the axiomatic system for capturing Intention. For software to be developed in conventional ways, the logic to control processing is crucial and has to be defined by developers because the software need to be deduced from Requirement, not Intention. As a consequence, for conventional software development a software model has to be made on a basis of Requirement, and the model is nothing but the logic to control processing. As SF realizes a software model that can be applied universally, even though our methodology uses Requirement, what is expected to Requirement is different from that of conventional ways. The amount of information necessary for our methodology is smaller than that of conventional ways, moreover, the information to be used is much simpler than conventional ways.[1]

---

[1] See our website at http://www.lyee.co.jp.

A mentioned above, Requirement is expressed as sentences. The sentences are regarded as a set of words. One of the characteristics of our methodology is that the rule of grammatical sequence of the terms can be ignored. We form a set of words. This set is made into a unit for each screen and printout. A unit establishes a SF. Since Requirement consists of the plural units, Requirement is replaced with plural SF. Plural SF are linked each other with four rules. Linked SF become a unit of software, which can be defined in a diagram. This diagram is called Process Route Diagram (PRD). These four rules will be discussed later. A variety of parts of speech belong to the unit. Each word belonging to the unit becomes a one-variable proposition taking a word as its variable. The rule to define the proposition is axiomatically and deterministically applied to any word, and we call this rule Predicate Structure. Predicate Structure and the four rules mentioned above are deduced from the axiomatic system. Although the way they are deduced is an important issue, this issue is not discussed in this paper. The proposition, four rules and the elements to be mentioned later are generally called Vector.

In summary, SF consists of a set of Vectors, and the sequence of placement of Vectors in a set can be neglected. The four rules, other elements and the sequence of placement of Vectors are discussed later. If a word cannot be defined as Vector by Predicate Structure, the word should be eliminated from candidates for Vector. Sentences defining Requirement are handled as follows: The role of a noun is replaced with a proposition. The role of an intransitive verb is absorbed in a proposition. The role of a transitive verb is considered to be absorbed in the structure of TDM, and its role is generated autonomously when SF is executed on a computer. The role of other parts of speech is considered to be unable to be absorbed in software. Such words are, for example, “beautiful”, “lively”, and “shall”. The role of these words is entrusted to an individual. The role of an article is considered to be a part of a noun in software. The actions mentioned above, such as to replace, absorb, autonomously generate, and entrust, are done in a deterministic manner. The cognizable foundation on which our methodology simplifies Requirement is axiomatic establishment of the deterministic manner.

3 Concept of Scenario Function

In this section, we discuss SF as a formula.

$$SF = \Phi[\Phi 4(\{L4,j\},\{O4,r\alpha\},\{S4,r\beta\},R4) + \Phi 2(\{I2,r\alpha\},\{L2,i\},R2) + \Phi 3(\{L3,j\},\{R3,k\})]$$

$\Phi 4(\{L4,j\},\{O4,r\alpha\},\{S4,r\beta\},R4)$, $\Phi 2(\{I2,r\alpha\},\{L2,i\},R2)$ and $\Phi 3(\{L3,j\},\{R3,k\})$ are called Pallet and denoted as *W04*, *W02* and *W03* respectively. In the axiomatic system, we exist in both cognizable and incognizable spaces. Intention establishes in the incognizable space whereas Requirement establishes in the cognizable space. As Pallet represents cognizable space, Pallet can be defined. SF is defined based on the Requirement established in the cognizable space. Execution of SF means to establish the cognizable space defined by Pallet, and establishment of the space means that Intention has been established. According to the axiomatic system, establishment of Intention causes to establish cognition. This hypothetical story is reflected in Predicate Structure and TDM. The elements enclosed in the parentheses *( )* are the all kinds of Vectors. The role of Vectors differs from each other based on the role of Pallet. The rule of Predicate Structure is to complete the instructions composing a Vector in a programming language and to determine the sequence of placement of these instructions. That is, Predicate Structure deduced axiomalically is the rule to define Vectors deterministically. The number of kinds of Vectors is ten. That is, six of them are *L4,j*, *O4,r*α, *S4,r*β, *I2,r*α, *L2,i*, *L3,j* and the remaining four kinds are to link Pallets or SF. These four kinds of Vectors are referred to as the four rules in the above and herein after called Routing Vector. They are denoted as *R4*, *R2* and *R3,k*. Vectors denoted as *L4,j*, *L2,I* and *L3,j* are called Signification Vector and defined as one-variable (one-word) propositions. Vectors denoted as *O4,r*α, *S4,r*β, and *I2,r*α are generally called Action Vector and defined as plural-variable (plural-word) propositions.

A sign + in SF means that Pallets are controlled in the order of *W04*, *W02* and *W03* when they are executed. Φ*4,* Φ*2* and Φ*3* are not defined with Predicate Structure, i.e. not defined axiomatically, but defined deterministically by Vectors belonging to *W04*, *W02* and *W03*. Φ*4,* Φ*2* and Φ*3* are called Pallet Function and control execution of elements enclosed in the parentheses *( )* on a computer. Routing Vectors are as follows: *R3,k* are categorized into four kinds, i.e., the role of them are categorized as Continuous, Recurrence, Duplex and Multiplex links. The role of *R4* and *R2* is the same as that of *R3* Continuous link. The role of Continuous link of *R4* is to hand over the control of execution from *W04* to *W02*. This role is denoted as + in the formula. The role of Continuous link of *R2* is to hand over the control of execution from *W02* to *W03*. This role is also denoted as + in the formula. The role of Continuous link of *R3,1* is to hand over the control of execution from *W03* to *W04* of the next SF. The role of Recurrence link of *R3,2* is to hand over the control of execution from *W03* to *W04* of the same SF. The role of Duplex link of *R3,3* is to hand over the control of execution from *W03* to *W03* of the preceding SF adjoined to the current SF. The role of Multiplex link of *R3,4* is to hand over the control of execution from *W03* to W04 defined as its preceding SF. As the result of execution of Routing Vector, Pallet is executed iteratively in itself or linked with other Pallet. Control of iteration is done by Pallet Function based on the information produced when Routing Vectors are executed. Control of linking is done by Φ based on the information produced when Routing Vectors are executed. Φ is called Tense Control Function. Φ is expressed in a programming language although it is not defined with Predicate Structure, i.e., not defined axiomatically but deterministically by the characteristics of SF and the execution environment.

The role of Signification Vector when they are executed is to produce a relation to establish Intention. When Signification Vector becomes TRUE, the relation is established by the axiomatic system. If not TRUE, the Vector autonomously iterates until it gets TRUE. For this purpose, iteration and linking of Pallets are done. If it turns out that the Vector cannot become TRUE, the Vector requests a person to play its role. This logic to control processing is autonomously created by SF. Being TRUE means that a value is made in the memory area for the 4th box of a proposition. The structure of a proposition (Predicate Structure) and the 4th area are shown in Figure 1. The role of Action Vector is to control the relation between the defined and CPU memory. *O4,r*α is

to hand over data from CPU memory to the defined. *I2,r*α is to get data from the defined to CPU memory. *S4,r*β is to manage the CPU memory. *r*α and *r*β are units of data when Action Vectors are executed where α and β express that the structure of the units, or a set of words, is different from each other.

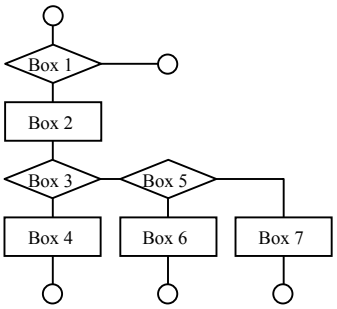

Figure 1. Predicate Structure

4 Relation between SF and Requirement

SF possesses two characteristics. One appears when SF is defined by a human and the other is created when SF is executed on a computer, and these characteristics are different. On the other hand, such difference does not appear in conventional programs. For example, a control structure of conventional programs is realized as defined by a human, whereas the control structure of SF is not necessary to be defined by a human as it is autonomously realized on a computer. Such difference between conventional ways and SF influences the degree of complexity of human work of dealing with Requirement. The structure of SF is determined by Φ, Φ4, Φ2 and Φ3, which are deduced from the axiomatic system, but not Requirement as conventional ways. Definition of Vector is simplified as Vectors are logically independent of each other owing to Predicate Structure and the structure of SF.

Label	ID	I/O attribute	Defined to which word belongs	Data type	Length	Decimal places	Array-column	Array-row	User's logic	Meta-expression
Company Code	A0010	O	FW01	S9	3	0	Not used	Not used	If FR02 has not been read, set A0010 of FR01 to A0010 of FW01.	IF CRX_FR02_RKSTS = '3' SET FW01.A0010 = FR01.A0010

Figure 2. Definition of Words Expressed in Requirement

File name	Logical ID	Physical ID	How to use file by Input or output	Access conditions
Monthly sales record by classification	F03	HKF0D0	O	Structure of the key for adding a record: F01.Company Code AND F01.Branch Code AND F01.first 6 digits of Transaction Date AND F01.Classification Code AND F01.Fiscal Year

Figure 3. Definition of the Defined

When defining SF, developers do not need to think about how SF is executed on a computer, i.e. the logic to control processing, whereas in conventional ways they have to think about how programs to be executed on a computer. As complexity of Requirement is replaced with that of the logic to control processing, in the conventional world we encounter the issue of complexity. Since the logic to control processing is deeply related to our thought, this issue is not limited to software. It is considered that by using SF we are free from this issue. Such effect of SF is considered to be brought by the establishment of the hypothetical axiomatic system to capture Intention.

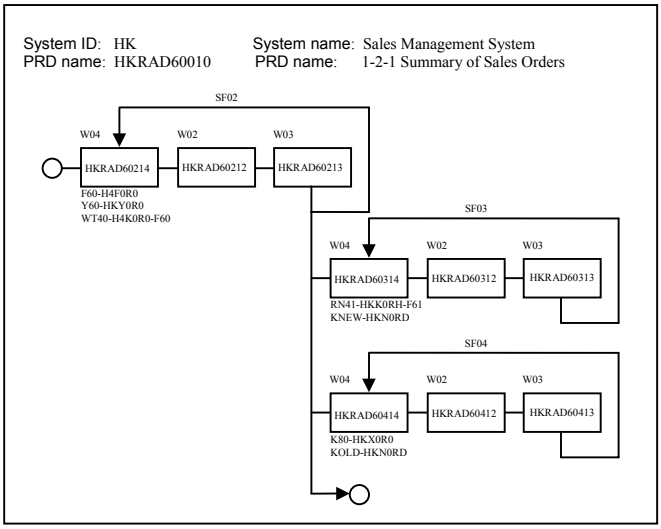

Figure 4. Definition of PRD

Let us explain concretely what are needed as Requirement for our methodology. They are categorized into three as follows:

1. Definition of words expressed in Requirement
2. Definition of the defined
3. Definition of PRD

Once definition of words expressed in Requirement is done, definition of the defined is axiomatically done. Then, PRD can be defined axiomatically. What is done axiomatically here can be replaced with logic to establish SF automatically. Examples of the above 1, 2, and 3 are shown in Figures 2, 3, and 4[2]. When SF is to be established automatically, a tool for it is used. The logic used for the tool is not

[2] These are extracts from actual projects.

complicated as this tool is also on a basis of the axiomatic system. Requirement for our methodology is basically the definition of words expressed in Requirement.

Source codes of the Vectors in COBOL appear in Figures 5 and 6, where a set of items in *italic* represents user's Requirement. As *R2* is the same as *R4*, *R2* is not presented in the Figures. Since boxes 5, 6 and 7 of *O4*, *I2*, *R4*, *R3* and *S4* are not important in terms of the discussion, they are omitted due to the limitation of the space.

```
L4-FORMID-j1 SECTION.
*BOX1
   IF  j1 OF W03-FORMID NOT = "j1"
   OR j1 OF W04-FORMID NOT = LOW-VALUE
      CONTINUE
   ELSE
*BOX2: USER'S LOGIC
      COMPUTE j1 OF WORK4-FORMID = i1 OF W04-FORMID
                                 + j2 OF W04-FORMID
*BOX3
      IF j1 OF  WORK4-FORMID  NOT = LOW-VALUE
*BOX4
         MOVE j1 OF  WORK4-FORMID TO j1 OF W04-FORMID
      ELSE
*BOX5
         IF STATUS-CHAGE-FLG OF  W04-FORMID = LOW-VALUE
*BOX6
            MOVE "1" TO j1-FALSE            OF  W04-FORMID
         ELSE
*BOX7
            MOVE "1" TO j1-REEXECUTE   OF  W04-FORMID
         END-IF
      END-IF
   END-IF
EXIT.
```

```
L2-FORMID-i1 SECTION.
*BOX1
   IF i1 OF W02-FORMID NOT = LOW-VALUE
      CONTINUE
   ELSE
*BOX2: USER'S LOGIC
      MOVE i1 OF READ-WFL-FORMID TO i1 OF WORK2-FORMID
*BOX3
      IF i1 OF  WORK2-FORMID IS NUMERIC
*BOX4
         MOVE i1 OF WORK2-FORMID TO i1      OF W02-FORMID
         MOVE i1 OF WORK2-FORMID TO i1-BD OF W04-FORMID
      ELSE
*BOX5
         IF STATUS-CHAGE-FLG OF W02-FORMID = LOW-VALUE
*BOX6
            MOVE "1" TO i1-FALSE          OF W02-FORMID
         ELSE
*BOX7
            MOVE "1" TO i1-REEXECUTE OF W02-FORMID
         END-IF
      END-IF
   END-IF
EXIT.
```

```
L3-FORMID-j1 SECTION.
*BOX1
   IF j1 OF W03-FORMID NOT = LOW-VALUE
      CONTINUE
   ELSE
*BOX2: USER'S LOGIC
      IF i1 OF W02-FORMID NOT = LOW-VALUE
         MOVE "j1" TO j1 OF WORK3-FORMID
      ELSE
*BOX3
         IF j1 OF WORK3-FORMID NOT = LOW-VALUE
*BOX4
            MOVE j1 OF WORK3-FORMID TO j1 OF W03-FORMID
         ELSE
*BOX5
            IF STATUS-CHAGE-FLG OF W03-FORMID = LOW-VALUE
*BOX6
               MOVE "1" TO j1-FALSE          OF W03-FORMID
            ELSE
*BOX7
               MOVE "1" TO j1-REEXECUTE OF W03-FORMID
            END-IF
         END-IF
      END-IF
   END-IF
EXIT.
```

```
O4-FORMID SECTION.
*BOX1
   IF    j1 OF W04-FORMID = j1-PRV OF W04-FORMID
   AND j2 OF W04-FORMID = j2-PRV OF W04-FORMID
      :
   AND jm OF W04-FORMID = jm-PRV OF W04-FORMID
*BOX2
      MOVE j1  OF W04-FORMID TO j1 OF WFL-FORMID
         :
      MOVE jm OF W04-FORMID TO jm OF WFL-FORMID
      WRITE  FORMID
*BOX3
      IF WRITE-STS OF CONTROL-BOX = LOW=VALUE
*BOX4
         MOVE LOW=VALUE TO j1 OF W04-FORMID
            :
         MOVE LOW=VALUE TO jm OF W04-FORMID
      END-IF
   END-IF.
EXIT.
```

```
I2-FORMID SECTION.
*BOX1
   IF KEY  OF CONTROL-BOX NOT = LOW=VALUE
*BOX2
      READ FORMID TO WFL-FORMID
*BOX3
      IF READ-STS OF CONTROL-BOX = LOW=VALUE
*BOX4
         MOVE "1" TO READ-FALSE OF CONTROL-BOX
      END-IF
   END-IF
EXIT.
```

```
R4-FORMID4 SECTION. (CONTINUOUS LINK)
*BOX1
   IF    j1 OF W04-FORMID = j1-PRV OF W04-FORMID
   AND j2 OF W04-FORMID = j2-PRV OF W04-FORMID
      :
   AND jm OF W04-FORMID = jm-PRV OF W04-FORMID
*BOX2
      MOVE "FORMID-W02PALLET"
           TO NEXT-PALLETID OF WORK4-FORMID
*BOX3
      IF NEXT-PALLETID OF WORK4-FORMID NOT = LOW=VALUE
*BOX4
         MOVE NEXT-PALLETID OF WORK4-FORMID
              TO NEXT-PALLETID OF CTRL
      END-IF
   END-IF
EXIT.
```

Figure 5. Source Codes of Vectors (COBOL) - 1

```
R3-FORMID  SECTION. (RECURRENCE LINK)
*BOX1
   IF     j1  OF  W03-FORMID  =  j1-PRV  OF  W03-FORMID
   AND  j2  OF  W03-FORMID  =  j2-PRV  OF  W03-FORMID
      :
   AND jm  OF  W03-FORMID  =  jm-PRV  OF  W03-FORMID
*BOX2
      MOVE  "SELF-FORMID-W04PALLET"
                TO  NEXT-PALLETID OF WORK3-FORMID
*BOX3
      IF NEXT-PALLETID OF WORK3-FORMID  NOT = LOW=VALUE
*BOX4
         MOVE NEXT-PALLETID OF WORK3-FORMID
            TO NEXT-PALLETID OF CTRL
      END-IF
   END-IF
EXIT.
```

```
S4-FORMID  SECTION.
*BOX1
   IF     j1  OF  W04-FORMID  =  j1 -PRV  OF  W04-FORMID
   AND  j2  OF  W04-FORMID  =  j2 -PRV  OF  W04-FORMID
      :
   AND  jm  OF  W04-FORMID  =  jm -PRV  OF W04-FORMID
*BOX2 (not used)
*BOX3 (not used)
*BOX4 (not used)
      MOVE  LOW-VALUE  TO  i1 OF W02-FORMID
      MOVE  LOW-VALUE  TO  i2 OF W02-FORMID
         :
      MOVE  LOW-VALUE  TO  in OF W02-FORMID
      MOVE  LOW-VALUE  TO  j1 OF W03-FORMID
      MOVE  LOW-VALUE  TO  j2 OF W03-FORMID
         :
      MOVE  LOW-VALUE  TO  jm  OF  W03-FORMID
   END-IF
EXIT.
```

Figure 6. Source Codes of Vectors (COBOL) - 2

There is a logical relation between SF and conventional programs. This relation establishes the logic that converts the structure from one to the other between them. This is possible because SF is defined deterministically. The fact that SF is deterministically defined inspires us to improve its availability and serviceability.

4.1 Productivity

Let us discuss productivity from a viewpoint of development and maintenance. As it is clear from the above example, Vector is defined with the information of a minimum Requirement (word information). Looking at the structure of Vector closely, the number of lines of source codes of any Vector of the same kind is the same, and the role of instructions defined in the 1st box of a Vector is to verify that the Vector can be executed. The role of instructions defined in the 3rd box of a Vector is to check the validity of the result of execution of the Vector. Vector autonomously performs verification and validation when it is executed, but those who define Vector do not need to be aware of it because the instruction in the 1st box is defined axiomatically and that of the 3rd box is defined axiomalically as well as deterministically. The instruction in the 2nd box is defined with user's logic except logic to control processing. The instructions in the 4th, 5th, 6th and 7th boxes are defined axiomatically. Therefore, verification and validation of a set of Vectors, i.e. that of SF as a whole, is always guaranteed by Vector, and this means that correctness of SF is guaranteed. In conventional ways, there is no guarantee of the correctness of the program as a whole even if a part of the program is correct. In other words, it has not been possible to make a part that makes the whole correct. However, Vector makes it possible. In conventional ways, tests of a program as well as combined programs as a whole are required, whereas SF does not need to be tested in such a way. In summary, Requirement for our methodology is to be simplified, and SF has a variety of characteristics to expedite mental work. These improve the productivity of human work of defining SF.

4.2 Reliability

Reliability of software of our methodology is obvious owing to the characteristics of the structure of SF as mentioned.

4.3 Availability

Any Requirement that can be expressed in a natural language can be replaced with SF as mentioned. Since our cognition is expressed in a natural language, our cognition is to be replaced with SF.

4.4 Serviceability

Serviceability of software is assured by maintenance in conventional ways. This is considered to be characteristics of software. Therefore, in order to improve serviceability, maintainability should be high. However, in conventional ways, improvement of serviceability has not been done sufficiently. Figure 7 shows correspondence of processes of software development in our methodology and conventional ways. Programming in conventional ways is completely automated in our methodology, and testing in conventional ways is replaced with a part of our tool. Validity of Requirement is replaced with a function of our tool as semantics analysis. This logic, which might be considered complex, is replaced with a relation among word for which Vectors are established. This means that maintenance and development of software can be done in a same manner, and they are done in much shorter time than in conventional ways.

5 Execution Performance of SF

When programs are developed in our methodology and a conventional way for the same Requirement, the number of lines of source codes of SF is three to five times as much as that of conventional programs. When a conventional program is automatically converted to SF, the number of lines of the source codes increases to 10 to 50 times. However, the important point is that this does not mean that the number of lines of SF to be executed is greater than that of the conventional program to be executed. In conventional programs, instructions to be executed are defined, whereas in SF, all the instructions are defined and only the necessary instructions among them are executed based on the execution conditions. As a consequence, the CPU time is almost the same as conventional ways. SF can be automatically converted into the structure of conventional programs.

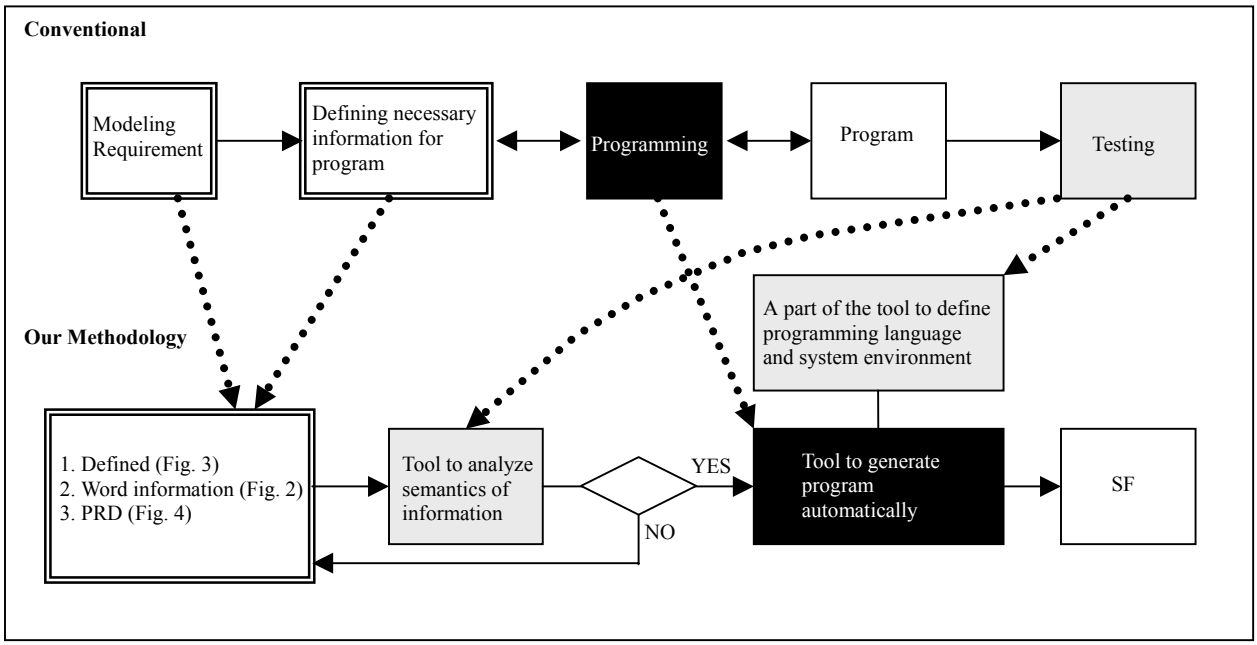

Figure 7. Correspondence between Conventional Way and our Methodology

6 Conclusion

Emergence of software for computer systems is mostly attributed to the progress of development technology of hardware, but not to the development of software engineering. This is clear from the evidence that since the early days of software development, software has been developed in a way as if they have tried to build a high-rise building with lumber. Consequently, a large number of trials have been proposed, but none of them has contributed to improving efficiency of the intrinsic quality of the software development process. Moreover, functionality as a system, which is acquired as a result of software development, is advertised while the issue of software development itself is always forgotten. Although a lot of contradictions due to deficiency of development technology appear one after another and are left behind, a worldwide tendency to excluding an opportunity to confront and solve the problems has been created.

Software is a means of capturing the real world with phenomena. Whereas software is established based on Intention of its users and developers as a physical structure, the connection between Software and the Intention is stronger than that between a physical structure and the Intention. This is the intrinsic quality of software and indicates that software is existence with the characteristics that we should capture it in the metaphysical world. In short, a viewpoint (theory) that establishes such a world is indispensable. That is, the intrinsic quality of software requires a viewpoint attained in the metaphysical world beyond the abstract concepts created by a traditional knowledge system. Therefore, problems arisen from software development technologies cannot be solved without overcoming this issue.

Our methodology is a theory constructed on a basis of the above reflection. This is not a mere theory to show some concepts although traditional theories are often to be so. This methodology is nothing but a theory to specify the precise work process of software development.

References

HAMID, I.A. and NEGORO, F. (2001): New Innovation on Software Implementation Methodology for the 21st Century -What Software Science can Bring to Natural Language Processing-. *Proc. SIC 2001 World Multiconference on Systemics, Cybernetics and Informatics,* Orlando FL, USA, **XIV**: 487-489, IIIS (International Institute of Informatics and Systemics) & IEEE Computer Society.

NEGORO, F. (2001): Intent Operationalisation for Source Code Generation. *Proc. SIC 2001 World Multiconference on Systemics, Cybernetics and Informatics,* Orlando FL, USA, **XIV**: 496-503, IIIS (International Institute of Informatics and Systemics) & IEEE Computer Society.

NEGORO, F. (2000): Principle of Lyee Software. *Proc. 2000 International Conference on Information Society in the 21st Century*, Aizu-Wakamatsu, Japan, 441-446, The University of Aizu, Information Processing Society of Japan & IEEE Japan Council.

This paper appears in the Proceedings for ICII2001 in October 2001, in Beijing, China.