

On the Informatics Laws and Deductive Semantics of Software

Yingxu Wang, *Senior Member, IEEE*

Abstract – A fundamental finding in computer science is that software, an artifact of human creativity, is not constrained by the laws and properties known in the physical world. Thus, a natural question we have to ask is: what are the constraints that software obeys?

This paper attempts to demonstrate that software obeys the laws of informatics and mathematics. This paper explores a comprehensive set of informatics and semantic properties and laws of software as well as their mathematical models. In order to provide a rigorous mathematical treatment of both the abstract and concrete semantics of software, a new type of formal semantics known as the deductive semantics is developed. The deductive models of semantics, semantic function, and semantic environment at various composing levels of programs are formally described. The findings of this paper can be applied to perceive the basic characteristics of software and the development of fundamental theories that deal with the informatics and semantic properties of software.

Index Terms – Cognitive informatics, software engineering, metaphors on software, foundations, constraint laws of software, informatics properties, deductive semantics, semantic analysis.

I. INTRODUCTION

It is recognized that *matter*, *energy*, and *information* are the three essences of the natural and abstract worlds [17, 18]. In a modern society, information plays an increasingly important role because it is the only link between the physical (external) and the abstract (internal) worlds in human life. In cognitive informatics [17, 18], software is perceived as a type of instructive and behavioral information that describes a solution for the design and implementation of a computing system.

A fundamental finding in computer science and software engineering is that software, as a unique entity, is

not constrained by any law or principle known in the physical world [7, 11, 17]. Therefore, we have to ask the following question: *What kind of constraints does software obey?*

This paper attempts to demonstrate that software obeys the laws of informatics [18, 22], because software is a mathematical entity and a kind of instructive and behavioral information that we use to communicate with computers as the servers, to provide specified functionality for users of the computing system [17]. However, by answering the above question, another important question is developed that asks: *What are the laws of informatics that constrain software in computing and software engineering?*

In order to answer the above question, this paper explores the informatics and semantic properties and laws of software, as well as their mathematic models. The metaphors and basic characteristics of software are examined in Section II. A set of 19 informatics laws of software is derived in Sections III. Then, the semantic laws of software are developed on the basis of a novel formal semantics known as *deductive semantics* in Section IV, which rigorously interpret and analyze the meanings and behaviors of software.

II. FUNDAMENTAL CHARACTERISTICS OF SOFTWARE

In order to reveal the fundamental characteristics of software, this section explores basic perceptions or metaphors of software. In contrast to the conventional mathematics or product metaphors, an information metaphor is introduced that identifies software as instructive and behavioral information.

A. Metaphors of Software

The nature of software has been perceived quite differently in research and practice of computing and software engineering. The following metaphors have been reported in the literature:

- Software is a mathematical entity [6, 9]
- Software is a concrete product [10, 11, 15]
- Software is a set of behavioral instructions to computers and a coded solution to given problems [18, 22]

Manuscript received Jan. 6, 2004; revised Sept. 26, 2004/July 28, 2005. This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). This paper was recommended by Guest Editor W. Kinsner.

Yingxu Wang is with Theoretical and Empirical Software Engineering Research Center, University of Calgary, 2500 University Drive NW, Calgary, AB, T2N 1N4, Canada (e-mail: yingxu@ucalgary.ca).

Digital Object Identifier

The following subsections analyze the basic characteristics of software in terms of the *mathematics*, *product*, and *informatics* metaphors.

1) The Mathematical Metaphor of Software

The *mathematical metaphor* of software has been adopted by some computer scientists who perceive software as a stored programmed logic on computing hardware [1, 4] or a mathematical entity [9].

A general taxonomy of the usages of computational mathematics can be derived on the basis of their relations with natural languages. It is recognized that language is the means of thinking. Although natural languages can be rich, complex, and powerfully descriptive, they share the common and basic mechanisms [19, 22] in three functional categories known as ‘to be ($=$),’ ‘to have ($|<$),’ and ‘to do ($|>$)’ as shown in Table I.

TABLE I
FUNDAMENTAL ELEMENTS IN NATURAL LANGUAGES

Function	Category	Notation	Example
Identify objects and attributes	To be	$ =$	$A = B$ (A is B)
Describe relations and possession	To have	$ <$	$A < B$ (A has B)
Describe status and behaviors	To do	$ >$	$A > B$ (A does B)
	Indirect to do	$ >> \dots >$	$A >> B > C$ (A has B to do C)
Describe negative facts	Negative	\neg	$A \neg = B$ (A is not B) $A \neg < B$ (A has not B) $A \neg > B$ (A does not B)

All mathematical means and forms, in general, are an abstract description of these three categories of human or system behaviors and their common rules. Taking this view, mathematical logic may be perceived as the abstract means for describing ‘to be,’ set theory for describing ‘to have,’ and algebras, particularly the process algebra, for describing ‘to do.’ A descriptive mathematic means in the third category, known as Real-Time Process Algebra (RTPA), has been developed in [16] for a formal and rigorous treatment of human and software system architectures and behaviors [19, 21-23].

2) The Product Metaphor of Software

In the IT industry, software is perceived broadly as a concrete product [11, 15]. With the *product metaphor*, a number of manufacturing technologies and quality assurance principles were introduced into software engineering. However, the phenomenon that we are facing almost all the same problems in software engineering as we dealt with 40 years ago, indicates a failure of the manufacture-based and mass-production-oriented

metaphor, and related technologies in software development. Therefore, we have to rethink the nature of software and how we produce software in software engineering.

There are three generic engineering goals, known as *efficiency*, *productivity*, and *quality*. These generic goals can be described by a triangular Engineering Objective Model (EOM) as shown in Fig. 1 [22]. In the EOM model, each of the three generic goals obeys a basic constraint for engineering organization and practice in terms of *costs*, *time*, and *utility*, respectively. It is found, unfortunately, the three basic goals in engineering are contradictory. It is difficult to achieve them all at the same time within a given engineering context, or there is always a need for trade-offs between these goals.

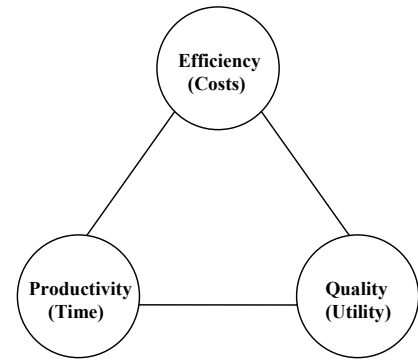


Fig. 1 The engineering objective model (EOM)

This observation on the EOM model as shown in Fig. 1 leads to the following theorem [22].

Theorem 1. The law of conservation of basic engineering constraints states that the three constraints of the basic objectives of engineering, known as *time* (T), *costs* (C), and *utility* (U) are conservative in a given engineering context, i.e.:

$$\begin{aligned}
 f_t(T^{-1}) + f_c(C^{-1}) + f_u(U) \\
 &= k \frac{U}{T \cdot C} \\
 &\equiv \delta
 \end{aligned} \tag{1}$$

where k and δ are constants. \square

Theorem 1 explains that the three basic engineering constraints can not be achieved to their maximum at the same time in a given engineering context. That is, any pair of constraints among the three may be achieved in the sacrifice or trade-off of the remainder. For example, the reduction of time (T) and expectations of better result (U) will increase costs (C).

Because software engineering is a branch of engineering disciplines, it obeys the generic engineering rules of Theorem 1. There are a number of goals proposed in software engineering [11, 15], such as to improve customer satisfaction, ensure quality, reduce time to

market, decrease costs and effort, improve process capability, enhance reliability/dependability/code stability, provide better services, minimize defects, increase project estimation accuracy, and obtain better maintainability. However, productivity, efficiency, and quality as identified in EOM are the most fundamental categories of goals that dominate the individual ones in software engineering.

In the EOM model, productivity is the principal objective and major purpose of any engineering discipline. The improvement of productivity is the key to achieve other engineering goals by technical innovation, i.e. by increasing δ as described in Eq. 1. For example, the automatic exchanger revolutions in the telecommunication industry in the 1940's and 1990's show how enhancement of productivity by technical innovation may increase δ . Therefore, it is inevitable that software engineering should set its paramount goal on the improvement of productivity in software development by intelligent tools rather than human labor.

3) The Informatics Metaphor of Software

Information is the third essence in modeling the natural world in addition to matter and energy. In cognitive informatics [17, 18] it is recognized that human beings live in a dual world. One aspect of it is the physical or the concrete world; the other is the abstract or the perceived world. We use *matter* and *energy* to model the former, and *information* to the latter. An information-matter-energy (IME) model, as shown in Fig. 2, is developed to describe the generic view towards the physical and information worlds [18].

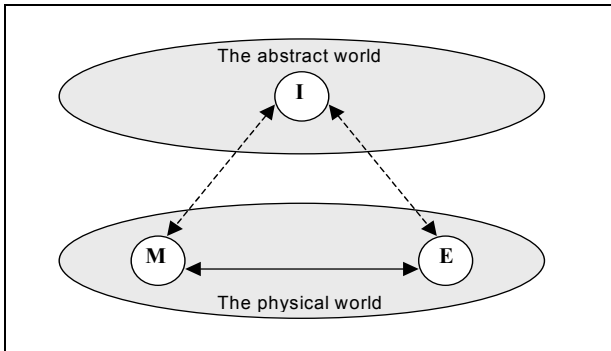


Fig. 2 The IME model of the world view

Models of the natural world have been well studied in physics and other natural sciences. Relationships between matter and energy have been investigated and revealed by Albert Einstein and other scientists. However, the modeling of the abstract world is still a fundamental issue yet to be explored in informatics, computing, software science, cognitive science, and life sciences. Especially, the relationships between I-M-E and their transformations are perceived as one of the fundamental questions in

cognitive informatics. It is believed that any breakthrough in this area will profoundly facilitate the development of next generation technologies in informatics, computing, software, and cognitive sciences.

Definition 1. *Information* in cognitive informatics is defined as abstract artifacts and their relations that can be modeled, processed, and stored by human brains.

The *content of information* in cognitive informatics is measured by the cost of code to abstractly represent a given size of message M in a digital system based on a constant k [18], i.e.:

$$\begin{aligned} I_k &= f: M \rightarrow S_k \\ &= \log_k M \end{aligned} \quad (2)$$

where I_k is the content of information in a k -based digital system, and S_k the measurement scale based on k . The unit of I_k is the number of k -based digits.

Eq. 2 is a generic information size measurement. When a binary digital representation system is adopted, i.e. $k = b = 2$, it becomes the most practical one:

$$\begin{aligned} I_b &= f: M \rightarrow S_b \\ &= \log_2 M \quad [\text{bit}] \end{aligned} \quad (3)$$

where the unit of information, I_b , is a *bit*. Note that the bit here is concrete and deterministic, and it no longer possesses a property as a value of weighted probability as that in the classical informatics [18].

Software, in daily life, is simply meant as anything flexible and without a physical dimension. In cognitive informatics, software can be defined as follows:

Definition 2. *Software* is a kind of coded and instructive information that describes the algebraic process logic of software system architectures and behaviors in computing.

This definition indicates a new way to explain the laws and properties that govern the behavior of software. That is, the informatics metaphor provides a new approach to study the nature and basic properties of software in software engineering. Definition 2 also indicates that the current philosophy and methodology using manufacturing quality control principles in organizing and managing software engineering as mass manufacturing processes are perhaps fundamentally mismatching. Therefore, the processes and techniques widely used in the publishing and journalism sectors are worth to be intensively studied and adopted in software engineering on the basis of the informatics metaphor of software.

B. Software as Instructive and Behavioral Information

A software system can be perceived as a virtual agent of human beings created to do something repeatable, to extend human capability, reachability, and/or memory capacity. The author found that both human and software behaviors can be described by a three-dimensional representative model comprising of *action*, *time*, and *space*. For software system behaviors, the three

dimensions are known as *mathematical operations*, *event/process timing*, and *memory manipulation* [16, 19].

For explaining the information nature of software, let us consider when one needs a software system in particular, and a computing solution in general.

Theorem 2. The need for software is necessarily and sufficiently determined by the following three conditions:

(a) The *repeatability*: Software is required when one needs to do something for more than once.

(b) The *flexibility* or *programmability*: Software is required when one needs to repeatedly do something not exactly the same.

(c) The *run-time determinability*: Software is required when one needs to flexibly do something by a series of choices on the basis of varying sequences of events determinable only at run-time. \square

Theorem 2 describes that the above three situations, namely repeatability, flexibility, and run-time determinability, form the necessary and sufficient conditions that warrant the requirement for a software solution in computing [22].

Repeatability is one of the most premier needs for a software solution, but it is not the only sufficient condition for demanding a software system, because repeatability may also be implemented by wired logic or hardware. Therefore, the flexibility and run-time determinability, particularly the latter, are necessary and sufficient for the usage of software. The third situation may be also considered as the *non-determinism* at compile-time or design-time. This property of software is the fundamental issue in computation that constitutes the complexity of programming [4, 22].

In conventional engineering disciplines, the common approach moves from abstract to concrete, and the final product is the physical realization of a design blueprint. In software engineering, however, the approach is reversed. The final software product is the virtualization and abstraction of a set of original real-world requirements by binary streams. The only tangible part of a software system is its storage media or its run-time behaviors. This is probably the most unique and interesting feature of software engineering.

According to cognitive informatics [18], information is any property or attribute of entities in the natural world that can be abstracted, digitally represented, and mentally processed. For software engineering to become a matured engineering discipline like others, it must establish its own laws and theories, which are perceived to be relied on cognitive informatics.

III. INFORMATICS LAWS OF SOFTWARE

The informatics laws that constrain software can be explored by investigating the properties and basic principles of cognitive informatics. The following

subsections describe fundamental properties and laws of information that software and software behaviors obeys. Some of the properties may not be necessarily complicated, but are profound in describing the axiomatic theory of software engineering.

Property 1. Abstraction: Information is abstract artifacts that are elicited from physical entities in the natural world or are created for representing relations between these entities or the entities and abstract mental objects. Information can be attributes, status, characteristics, structures, and dynamic processes of real-world entities, as well as relations between them. New information may be derived based on existing information and their relations in the abstract world [18].

Therefore, although it can be recorded, transformed, and communicated, information is the product of the brain and it exists in the abstract world. The IME model presented in Section II and the object-attribute-relation (OAR) model of internal information representation in the brain developed in [20] provide a generic view about the abstractive property of information and its relationship with the real-world entities.

Property 2. Generality: According to Property 1 (abstraction) and the OAR model [20], it can be derived that sources of information are widely general. Information can be elicited from objects, attributes, and their relations. Any physical entity in the universe is the source of information, and any abstract artifact (object) is the crystallization of information. Therefore, information is formed by the combination between physical entities, abstract objects, and relations between them, i.e.:

- Abstraction of physical entities and their attributes
- Relations between physical entities
- Relations between physical entities and abstract objects
- Relations between abstract objects

Hence, in a certain extent, cognitive informatics studies the sources and initiation of information, as well as the creation and perception of information by human cognitive processes.

It is realized in cognitive informatics that *relations* are information too. Although the number of the physical entities of the real-world is limited, their potential relations may result in an explosive exponential combination, i.e.:

$$C_n^k = \frac{n!}{k!(n-k)!} \quad (4)$$

where n is the total number of objects, and k is the number of average connections between the objects.

The human brain consists of about 100 billion (100×10^9) neurons, and each of them has hundreds to thousands of synapses (relations) connecting to various subsets of related neurons [20]. Thus, according to Eq. 4, the upper limit of potential connections of the neural clusters in the brain is in the order of:

$$\begin{aligned}
 C_n^k &= \frac{n!}{k!(n-k)!} \\
 &= \frac{10^{11}!}{10^3! \cdot (10^{11} - 10^3)!} \\
 &\approx 10^{8,432} \tag{5}
 \end{aligned}$$

This result is an astonishing finding [20, 22] that shows the capacity and complexity of the brain is on a magnitude we might only find in the universe. However, it indeed also interestingly exists in our internal world – the brain.

Property 3. Cumulativeness: The natural world is conservative. According to the natural law of conservation, matters and energies can neither be reproduced nor destroyed (while they may be transformed). However, information is not conservative but cumulative, because information may be created, destroyed, and reproduced. The *cumulativeness of information* is the most significant attribute of information that mankind relies on in evolution.

Property 4. Dependency on Cognition: Information should be recognized and consumed by human brains or other intelligent systems by a cognitive process before it can be effectively retained, retrieved, and used. According to the OAR model [20], information is represented internally by its relations with existing information and knowledge in the brain. Without cognition and comprehension, there is no information and knowledge, as well as no access and retrieval of them.

Property 5. Three-Dimensional Behavior Space: Information can be modeled by four independent factors known as the subject (O), behavior (B), space (S), and time (T). That is, information (I) can be determined by a 4-tuple:

$$I = (O, B, S, T) \tag{6.1}$$

where behavior B is a set of observable events, actions, and outcomes.

When the subject O is obvious, the information related to O , I_O , can be simplified as a triple as shown below:

$$I_O = (B, S, T) \tag{6.2}$$

Therefore, software as instructive information, I_s , can be modeled in a 3-dimensional behavioral space Ω , i.e.:

$$I_s = \Omega = (B, S, T) \tag{6.3}$$

where B is a finite set of operations.

Property 6. Sharability: Information can be shared and reused by multiple users without loss in quantity and without degradation in quality. Information may be amplified or multiplied by broadcasting. The lossless reuse of existing information will usually result in the creation of new information.

Property 7. Dimensionless: Related to Property 1 (abstraction), information has no physical size and dimension. No matter how large or small of the physical entities, their conceptual abstraction is only one unit. Their abstract representations or the cognitive visual objects

occupy a similar sight frame; only the resolutions may be varying [17].

Property 8. Weightless: A direct corollary based on Property 7 (dimensionless) is that the weight of information, W_i , is always zero, i.e.:

$$W_i \equiv 0 \tag{7}$$

This explains why an empty or full hard disk has the same weight; a blank or recorded tape has no difference in weight; and a memory chip storing all 0s, all 1s, or any combinations of them has the same weight. This property of information can also explain why one can afford to do a PhD without feeling any change of the weight of the brain, rather than the changes of its internal configurations.

Property 9. Transformability between I-M-E: According to the IME model [18], the three essences of the world are predicated to be transformable between each other as shown in Fig. 3.

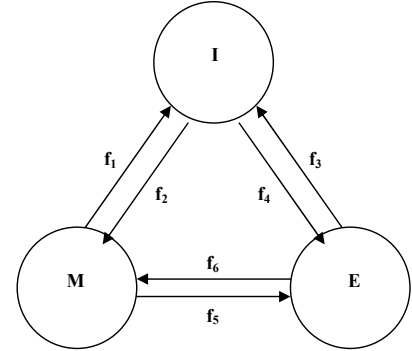


Fig. 3 The transformability between I-M-E

There are six possible relations between the three essences in the natural and information worlds. These relations can be described by the following generic functions f_1 to f_6 :

$$I = f_1(M) \tag{8.1}$$

$$M = f_2(I) \stackrel{?}{=} f_1^{-1}(I) \tag{8.2}$$

$$I = f_3(E) \tag{8.3}$$

$$E = f_4(I) \stackrel{?}{=} f_3^{-1}(I) \tag{8.4}$$

$$E = f_5(M) \tag{8.5}$$

$$M = f_6(E) \stackrel{?}{=} f_5^{-1}(E) \tag{8.6}$$

where a question mark denotes an uncertainty if there exists such a reverse function.

Albert Einstein has revealed Functions f_5 and f_6 , the relationship between matter (m) and energy (E), in the following form: $E = mc^2$. It is a great curiosity to explore what the remaining relationships and forms of transformation between I-M-E will be. In a certain extent, cognitive informatics is the science to seek possible solutions for f_1 to f_4 . A clue to explore the relations and transformability is believed in the understanding of the natural intelligence and information processing mechanisms in cognitive informatics.

Property 10. Multiple Representation Forms: Related to Property 1 (abstraction) and Property 2 (generality), it is observed that information can be represented in multiple forms, such as analogue (audio, visual), abstract (written languages and notation systems), and digital.

In the above classification, *digitalization* in information representation is the most generic and fundamental approach. The cognitive foundation of digitalization is that information is represented discretely or granularly in the brain with the basic unit as individual neurons. Therefore, the discrete representability is the foundation of information representation, storage, and processing. It is also the foundation of modern digital multimedia information engineering.

Property 11. Multiple Carrying Media: Parallel to Property 10 (multiple representation forms), information can be carried by various media, as listed in the following, and their combinations: electronic, electrical, magnetic, optical, mechanic, hydraulic, written, oral, and signs.

It is noteworthy that a certain medium may carry one or more forms of information. Correspondingly, a given form of information may be carried by different media.

Property 12. Multiple Transmission Forms: In addition to that information may be represented in multiple forms (Property 10) and carried by various media (Property 11), its transmission can be conducted in multiple forms as well. The following is the possible transmission forms of information:

- a) Information *passing*: 1 - to - 1
- b) Information *broadcasting*: 1 - to - n
- c) Information *gathering*: n - to - 1
- d) Information *networking*: n - to - m (9)

where l represents a single information source/receiver, n and m indicate multiple ones, and n and m can be the same.

The fast development of the Internet indicates that the fourth form of information transmission, *information networking*, is the highest form of communications.

Property 13. Dependency on Media: Information can not exist without a storage medium. The types of media may be organic, physical, chemical, or the combinations of them as described in Property 11. Therefore, in some extent, information may be perceived as a change of status of the storage medium.

Property 14. Dependency on Energy: All information processing tasks, such as acquisition, storage, retain, retrieve, and refresh, consume certain energy. There is no system that may process information without consuming energy. Therefore, in some extent, information may also be perceived as a change of status of energy on a given medium.

Property 15. Wearless and Time Dependency: The logic of formal information, such as special notation systems, mathematics, and philosophies as described at the abstract cognitive Levels 3 – 5 in Table II, does not wear

out. Once the logic of a specific piece of information is true, it remains so for eternity.

However, the timeliness of informal information is much shorter, i.e. such kind of information may be out of date quickly.

Property 16. Conservation of Information and Thermal Entropy: The law of thermodynamics that deals with the natural tendency of heat can be described as follows [3].

Theorem 3. *The second law of thermodynamics* states that:

- (a) Entropy of the universe ΔH_u does not change when a reversible process occurs, i.e.: $\Delta H_u = 0$;
- (b) Entropy of the universe ΔH_u increases when an irreversible process occurs, i.e.: $\Delta H_u > 0$. □

Because entropy can be interpreted in terms of disorder, when an irreversible process occurs and the entropy of the universe increases, the energy available for doing work decreases.

There are two types of entropies: the *thermal* and *information* entropy. The former exists in a physical system; the latter exists in an animate system, such as the brain and a human society. The second law of thermodynamics asserts that the thermal entropy in a closed physical system is conservative. Hence, an extended form of the second law of thermodynamics can be stated that in a hybrid system, the sum of the information entropy H_i and the thermal entropy H_t is a constant, i.e.:

$$k_t H_t + k_i H_i = \varepsilon \quad (10)$$

where k_t , k_i , and ε are constants for a given system.

Eq. 10 indicates that for the deduction of the thermal entropy of a system, the information entropy has to be increased. Therefore, the information entropy is also perceived as the *negative entropy*. In a physical system, entropy can be reduced by input of *energy* in order to maintain the order of the system. In neural and social systems, the order and the state of organization can be increased by inputting information.

Property 17. Quality Attributes of Informatics: On the basis of the conventional *product-based* metaphor [15], the quality of software is perceived as a collection of attributes, such as usability, availability, reliability, portability, and maintainability. Quality software is commonly considered as the software that contains fewer bugs. However, the concept of quality itself has never been properly defined in software engineering.

To model the quality of software and information, a set of *informatics-based* quality attributes is introduced such as *completeness*, *correctness*, *consistency*, *properly represented* (no mis-interpretation), *clearness* (no ambiguity), *feasibility* (can be implemented in technical and economical terms), and *verifiability* (attributes specified can be measured).

From this new angle, *software quality* can be defined as the achievement of the above inherent attributes for software architectures, static behaviors, and dynamic behaviors.

Contrasting the above two approaches towards software quality, it can be seen that the former is a set of external quality attributes, and the latter is a set of internal ones. The *internal quality attributes* should be focused and controlled first in software engineering. Otherwise, it would be too late to examine the external quality attributes, because this may only be carried out after a software system has already been built.

Property 18. *Susceptible to Distortion:* Unlike physical entities, information is more fragile and vulnerable to distortion, decay, and destruction. Therefore, information should be treated more carefully, and fault-tolerant and security techniques should be always adopted in dealing with information distortion.

Property 19. *Scarcity:* The principle of *information scarcity* states that information when needs is always inadequate, constrained by its availability, awareness, and/or the cost and complexity to thoroughly search, acquire, and comprehend it.

Theorem 4. The principle of *universal constraints* states that both the natural world and the perceived abstract world are constrained by certain known or yet to know restrictions and laws, due to the limitations of natural resources and/or human cognitive capability.

According to the principle of universal constraints, any theory, method, or technology has its own limitations and constraints. In a certain extent, science and engineering are the searching of the maximum extent of general relations between entities, phenomena, and behaviors under a set of constraints.

Theorem 4 will be found useful in a number of disciplines, such as the law of conservation of basic engineering constraints, the principle of generic constraints in systems theory, the principle of bounded rationality in decision theories, and the principle of resource scarcity in economics.

IV. DEDUCTIVE SEMANTICS OF SOFTWARE

The semantic properties of software are formal interpretations of software behaviors and constraint laws. Basic semantics of a programming language can be described by its *behavioral equivalency* to another language, for example, a natural language or a target language. Semantics can also be described by a set of predefined executable functions in machine languages. The third approach to specify the semantics of a programming language is by mathematical models known as the formal semantics, such as the *operational* [10, 14], *denotational* [2, 12, 13], *axiomatic* [6, 8], and *algebraic* [5, 14] semantics.

Definition 3. The *semantics* of a program in a given programming language is the logical consequences of an execution of the program that results in the changes of values of a finite set of variables in the underlying computing environment.

The mathematical models of the target machines and the semantic environments in conventional semantics seem to be inadequate to deal with the semantics of complex programming requirements, and to express some important instructions, complex control structures, and the real-time environments at run-time. In order to provide a rigorous mathematical treatment of both the abstract and concrete semantics of software, a new type of formal semantics known as *deductive semantics* is developed in this section that provides a systematic semantic deduction methodology.

A. The Mathematical Model of Programs

Prior to developing a deductive semantic model of software, necessary notations and mathematical models of program and its supporting platform need to be studied. This section introduces the big-R notation and the mathematical notion on partial differential of sets. Then, a hierarchical and compositional model of programs and an abstract logical model of system platforms are developed.

1) Mathematical Preparations

In order to develop the mathematical models of semantics and semantic environments, some special notations and operators are introduced to establish the fundamental expressive power required in semantic denotations and analyses. One of the important notations is the big-R notation for describing both repetitive behaviors and recurring architectures [16, 21]. The big-R notation is introduced first in the Real-Time Process Algebra (RTPA) [16], intending to provide a unified and expressive mathematical treatment for iterations and recursions in computing.

Definition 4. The *big-R notation* is a mathematical operator that is used to denote: (a) a set of *repetitive* behaviors, or (b) a finite set of recurring architectural constructs of computing, in the following forms:

$$(a) \prod_{\text{exp} \mathbf{BL} = \mathbf{T}}^{\mathbf{F}} P \quad (11.1)$$

$$(b) \prod_{i \in \mathbf{N}}^n P(i) \quad (11.2)$$

where \mathbf{BL} and \mathbf{N} are the type suffixes of Boolean and integer variables, respectively, as defined in RTPA. Other useful type suffixes that will appear in this paper are string (\mathbf{S}), pointer (\mathbf{P}), hexadecimal (\mathbf{H}), time (\mathbf{TM}), interrupt (\mathbf{C}), run-time type (\mathbf{RT}), system type (\mathbf{ST}), and the Boolean constants (\mathbf{T}) and (\mathbf{F}) [16].

Example 1. The architecture of a two-dimensional array with $n \times m$ integer elements, A_{nm} , can be denoted by the big-R notation as follows:

$$A_{nm} = \mathbf{RR}_{\substack{i=0 \\ j=0}}^{n-1 \ m-1} A[i, j] \mathbf{N} \quad (12)$$

Because the big-R notation provides a powerful and expressive means for denoting iterative and recursive behaviors and architectures of systems or human beings, it is a general mathematical calculus for system modeling in terms of repetitive ‘to do’ and recurrent ‘to be,’ respectively [16, 19, 21]. From this point of view, Σ and Π are special cases of the big-R for repetitively doing additions and multiplications, respectively.

In deductive semantics, another operator introduced is the *partial differential of sets* that is used to facilitate the instantiation of abstract semantics by concrete ones.

Definition 5. A *partial differential* of a set U on a subset X with elements $x, x \in X \subseteq U$, denoted by $\partial U / \partial x$, is a selection of interested elements from U as specified in X , i.e.:

$$\frac{\partial U}{\partial x} = X, \quad x \in X \subseteq U \quad (13)$$

The partial differential operation of sets can be easily extended to double or multiple partial differentials. For example:

$$\begin{aligned} \frac{\partial^2}{\partial x \partial y} U &= \frac{\partial U}{\partial x} \times \frac{\partial U}{\partial y} \\ &= X \times Y, \quad x \in X \subseteq U \wedge y \in Y \subseteq U \end{aligned} \quad (14)$$

and

$$\begin{aligned} \frac{\partial^3}{\partial x \partial y \partial z} U &= \frac{\partial U}{\partial x} \times \frac{\partial U}{\partial y} \times \frac{\partial U}{\partial z} \\ &= X \times Y \times Z, \quad x \in X \subseteq U \wedge y \in Y \subseteq U \wedge z \in Z \subseteq U \end{aligned} \quad (15)$$

where \times is a Cartesian product.

2) The Compositional Model of Programs

The semantics of a program in a given language can be described and analyzed at various composition levels, such as those of *statement*, *process*, and *system* from the bottom-up, according to the hierarchical architecture of the program. It is noteworthy that a statement is the *minimum unit* of semantics at the most fundamental level of programming. If the semantics of all fundamental instructions (known as the *meta processes* in RTPA) [16, 22] and their relational composition rules (known as the *process relations* in RTPA) in a given language can be defined, semantics of the process and program at the higher levels can be derived via mathematical deduction. This is the foundation of program composition, which will be formally described in Section IV.B.2.

Definition 6. A *statement* p in a program is an instantiation of a meta instruction of a programming language that executes a basic unit of coherent function and leads to a predictable behavior.

A set of 16 meta instructions in computing, as shown in Table II, has been identified and elicited in RTPA known as the meta processes [16]. Although, existing programming languages may implement a larger set of instructions, the additional ones are logical combinations of these 16 essential meta processes.

TABLE II
RTPA META PROCESSES

No.	Meta Process	Notation	Syntax
1	Assignment	:=	yRT := xRT
2	Read	>	Mem(ptrP)RT > xRT
3	Write	<	xRT < Mem(ptrP)RT
4	Input	>	Port(ptrP)RT > xRT
5	Output	<	xRT < Port(ptrP)RT
6	Addressing	⇒	idS ⇒ Mem(ptrP)RT
7	Memory allocation	←	idS ← Mem(ptrP)RT
8	Memory release	←	idS ← Mem(⊥)RT
9	Timing	@	@:TM @ \$tTM TM = yy:MM.dd hh:mm:ss.ms yy:MM.dd:hh:mm:ss.ms
10	Duration	△	@t _n TM △ \$t _n TM + ΔnN
11	Increase	↑	↑(nRT)
12	Decrease	↓	↓(nRT)
13	Exception detection	!	!(@eS)
14	Skip	∅	∅
15	Stop	⊠	⊠
16	System	§	§(SysIDS)

A process at the component level is composed by individual statements with given rules of composition.

Definition 7. A *process* P is a composed component in a program that forms a logical combination of n meta statements p_i and p_j , $1 \leq i < n$, $1 < j \leq n$, according to certain composing relations r_{ij} , i.e.:

$$\begin{aligned} P &= \mathbf{R}_{i=1}^{n-1} (p_i r_{ij} p_j), \quad j = i+1 \\ &= (\dots(((p_1) r_{12} p_2) r_{23} p_3) \dots r_{n-1,n} p_n) \end{aligned} \quad (16)$$

where r_{ij} is a set of relations or composing rules.

A comprehensive set of those composing rules has been identified and elicited in RTPA known as process relations [16, 22] as provided in Table III.

Definition 8. A *program* \wp is a composition of a finite set of k processes at the component level according to certain process dispatching rules, i.e.:

$$\wp = \mathbf{R}_{i=1}^k (@e_i \hookrightarrow P_i) \quad (17)$$

where \hookrightarrow denotes a process dispatch according to a predesignated event $@e_i$, which is an external, a system timing, or interrupt event [16].

TABLE III
RTPA PROCESS RELATIONS

No.	Process Relation	Notation	Syntax
1	Sequence	\rightarrow	$P \rightarrow Q$
2	Branch	\parallel	$? \text{exp}_{\mathbf{BL}} = \mathbf{T} \rightarrow P$ $? \sim \rightarrow Q$
3	Switch	\parallel \vdots	$? \text{exp}_{\mathbf{RT}} \rightarrow P_i$ $? \sim \rightarrow \emptyset$
4	While-loop	$\overset{\mathbf{F}}{\underset{\text{exp}_{\mathbf{BL}} = \mathbf{T}}{R}}$	$\overset{\mathbf{F}}{R} (P)$ $\text{exp}_{\mathbf{BL}} = \mathbf{T}$
5	Repeat-loop	$\overset{\mathbf{F}}{\underset{\text{exp}_{\mathbf{BL}} = \mathbf{T}}{R}}$	$P \rightarrow \overset{\mathbf{F}}{R} (P)$ $\text{exp}_{\mathbf{BL}} = \mathbf{T}$
6	For-loop	$\overset{n}{\underset{i=1}{R}}$	$\overset{n}{R} (P(i))$ $i=1$
7	Function call	\hookrightarrow	$P \hookrightarrow F$
8	Recursion	\cup	$P \cup P$
9	Parallel	\parallel	$P \parallel Q$
10	Concurrence	$\overset{\parallel}{\parallel}$	$P \overset{\parallel}{\parallel} Q$
11	Interleave	\parallel	$P \parallel Q$
12	Pipeline	\gg	$P \gg Q$
13	Time-driven dispatch	$@_t \mathbf{TM} \hookrightarrow P_i$	$@_t \mathbf{TM} \hookrightarrow P_i$
14	Event-driven dispatch	$@_e \mathbf{S} \hookrightarrow P_i$	$@_e \mathbf{S} \hookrightarrow P_i$
15	Interrupt	\ddagger	$P \ddagger Q$
16	Jump	\rightarrow	$P \rightarrow Q$

According to Definitions 7 and 8, a program can be reduced to the composition of a finite set of processes at the component level. Then, each of the processes can be further reduced to the composition of a finite set of statements at the bottom level.

B. The Deductive Semantic Model of Software

Deduction is a reasoning process that discovers new knowledge or derives a specific conclusion based on generic premises such as abstract rules or principles. This subsection develops the mathematical models of deductive semantics and elicits the fundamental properties of software semantics.

Definition 9. *Deductive semantics* is a formal software semantics that deduces the semantics of a program in a given programming language from a generic abstract semantic function to the concrete semantics, which are embodied onto the changes of status of a finite set of variables constituting the semantic environment of computing.

Deductive semantics perceives that the carriers of software semantics are a finite set of variables declared in a given program. Therefore, program semantics can be reduced onto the changes of values of these variables.

1) The Mathematic Model of Deductive Semantics

The semantic environment constituting the behaviors of software is inherently a three dimensional structure as described in Property 5 in Section III.

Definition 10. The *behavioral space* Ω of a program executed on a certain machine is a finite set of variables operated in a three-dimensional state space determined by a 3-tuple, i.e.:

$$\Omega = (B, S, T) = B \times S \times T \quad (18)$$

where B is a finite set of operations, S is a finite set of memory locations or their logical representations by identifiers of variables, and T is a finite set of discrete steps of program execution.

According to Definition 10, the variables of a program S play an important role in semantic analysis, because they are the *objects* of software behaviors and the *carriers* of program semantics. On the basis of the definition of software behavioral space, the semantic environment of software can be introduced as follows.

Definition 11. The *semantic environment* Θ of a program on a certain target machine is its run-time behavioral space Ω projected onto the Cartesian plane determined by T and S , i.e.:

$$\Theta = \frac{\partial^2 \Omega}{\partial t \partial s} = \frac{\partial^2}{\partial t \partial s} (B \times T \times S) = T \times S \quad (19)$$

As indicated in Definition 11, the semantic environment of a program is dynamic, because following each execution of a statement in the program, Θ , particularly the set of variables S and their values V , may be changed.

In semantic analysis, the changed part of the semantic environment Θ is particularly interesting, because it is the embodiment of software semantics. A generic semantic function is developed below, which can be used to derive a specific and concrete semantic function for a given statement, process, or program by mathematical deduction.

Definition 12. A *semantic function* of a program \wp , $f_\wp(\wp)$, is a finite set of values V determined by a Cartesian product on a finite set of variables S and a finite set of executing steps T , i.e.:

$$f_\wp(\wp) = f_\wp : T \times S \rightarrow V \quad (20)$$

$$= v_\wp(t, s), t \in T \wedge s \in S \wedge v_\wp \in V$$

$$= \begin{pmatrix} \mathbf{s}_1 & \mathbf{s}_2 & \cdots & \mathbf{s}_m \\ \mathbf{t}_0 & \perp & \perp & \cdots & \perp \\ \mathbf{t}_1 & v_{11} & v_{12} & \cdots & v_{1m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{t}_n & v_{n1} & v_{n1} & \cdots & v_{nm} \end{pmatrix}$$

where $T = \{t_0, t_1, \dots, t_n\}$, $S = \{s_1, s_2, \dots, s_m\}$, and V is a set of values $v_\wp(t_i, s_j)$, $0 \leq i \leq n$, and $1 \leq j \leq m$.

In Eq. 20, all values of $v_\wp(t_i, s_j)$ at t_0 is undefined for a program as denoted by the bottom symbol \perp , i.e. $v_\wp(0, s_j) = \perp$, $1 \leq j \leq m$. However, for a statement or a

process, it is usually true that $v_p(0, s_j) \neq \perp$ dependent on the context of previous statement(s) or the initialization of the system.

2) Properties of Software Semantics

Observing the formal definitions and mathematical models of deductive semantics developed in previous subsections, a number of common properties of software semantics may be elicited, which are useful for explaining the fundamental characteristics of software semantics.

One of the most interesting characteristics of program semantics is its invariance against different executing speeds as described in the following theorem.

Theorem 5. The semantics of a program are invariant with the changes of executing speed, as long as any absolute time constraint is met. \square

Theorem 5 asserts that different executing speeds or simulation paces will not alter the semantics of a software system. This explains why a programmer may simulate the run-time behaviors of a given program executing at a speed of up to 10^9 times faster than that of human beings. It also explains why computers with different system clocks may correctly run the same program and perform the same behavior.

The meta instructions shared by all programming languages can be classified into three categories: (a) *Internal operations*, such as memory manipulation and assignments; (b) *External operations*, such as input/output, event handling, and human-machine interactions; and (c) *Basic control structures* (BCS's) [16, 22], such as jump, branch, and iteration constructs as shown in Table III.

Theorem 6. A program is *compossible* in a given language *iff* both sufficient sets of meta instructions and BCS's are rigorously defined in the language. \square

Theorem 6 indicates that the necessary and sufficient conditions of program *compositionality* in a given language are that all the meta instructions (Table II) and fundamental BCS's (Table III) must be implemented in the language. In case of non real-time programming languages, the requirement for the four special BCS's, BCS's #10 through #13, may be waived. However, it is helpful to be aware of the whole set of software compositional rules for both ordinary and real-time software systems.

It is noteworthy that some of the BCS's as shown in Table IV are used to be treated as basic instructions rather than compositional rules. According to Theorem 5, there is a need to distinguish the semantic roles of statements (the *minimum semantic unit* of a language) and BCS's (the compositional rules of the language).

C. Formal Description of Software Behaviors by Deductive Semantics

The *behavior* of a computational statement is a set of observable actions or changes of status of objects operated

by the statement. According to the architectural model of programs as described in Section IV.A.2, the semantics of a program in a given language can be described and analyzed at various composition levels, such as those of *statement*, *process*, and *system* from the bottom-up.

Definition 13. The *semantics of a statement* p , $\theta(p)$, on a given semantic environment Θ is a double partial differential of the semantic function, $f_\theta(p) = f_p : T \times S \rightarrow V = v_p(t, s), t \in T \wedge s \in S \wedge v_p \in V$, on the sets of variables S and executing steps T , i.e.:

$$\begin{aligned} \theta(p) &= \frac{\partial^2}{\partial t \partial S} f_\theta(p) = \frac{\partial^2}{\partial t \partial S} v_p(t, s) \\ &= \underset{\#T(p)}{\mathbf{R}} \underset{\#S(p)}{\mathbf{R}} v_p(t_i, s_j) \\ &= \underset{i=0}{\mathbf{R}} \underset{j=1}{\mathbf{R}} v_p(t_i, s_j) \\ &= \begin{pmatrix} & \mathbf{s}_1 & \mathbf{s}_2 & \cdots & \mathbf{s}_m \\ \mathbf{t}_0 & v_{01} & v_{02} & \cdots & v_{0m} \\ (\mathbf{t}_0, \mathbf{t}_1] & v_{11} & v_{12} & \cdots & v_{1m} \end{pmatrix} \end{aligned} \quad (21)$$

where t denotes the discrete time immediately before and after the execution of p during $(t_0, t_1]$, and $\#$ is the *cardinal calculus* that counts the number of elements in a given set, i.e. $n = \#T(p)$ and $m = \#S(p)$.

In Definition 13, the first partial differential selects all related variable $S(p)$ of the statement p from Θ . The second partial differential selects a set of discrete steps of p 's execution $T(p)$ from Θ . According to Definition 13, the semantics of a statement can be reduced onto a semantic function that results in a 2-D matrix with the changes of values for all variables over time with program execution.

On the basis of Definitions 12 and 13, semantics of individual statements can be analyzed using Eq.21 via a deductive process.

Example 2. Analyze the semantics of Statement 3 in the following program entitled *sum*.

```
void sum;
{
    (0) int x, y, z;
    (1) x = 8;
    (2) y = 2;
    (3) z := x + y;
}
```

According to Definition 13, the semantics of Statement 3 are as follows:

$$\begin{aligned} \theta(p_3) &= \frac{\partial^2}{\partial t \partial S} f_\theta(p_3) = \frac{\partial^2}{\partial t \partial S} v_{p_3}(t, s) \\ &= \underset{3}{\mathbf{R}} \underset{\#S(p_3)}{\mathbf{R}} v_{p_3}(t_i, s_j) \\ &= \underset{i=2}{\mathbf{R}} \underset{j=1}{\mathbf{R}} v_{p_3}(t_i, s_j) \end{aligned}$$

$$= \begin{pmatrix} & \mathbf{x} & \mathbf{y} & \mathbf{z} \\ \mathbf{t}_2 & 8 & 2 & \perp \\ (\mathbf{t}_2, \mathbf{t}_3) & 8 & 2 & 10 \end{pmatrix} \quad (22)$$

This example shows how the concrete semantics of a statement can be derived on the basis of the generic and abstract function of deductive semantics.

According to Definitions 7 and 8, a program or a process is composed by individual statements with given rules of compositions. Therefore, the definitions and mathematical models of semantics at the statement level can be extended onto the higher levels of program hierarchy.

Definition 14. The *semantics of a process* P , $\theta(P)$, on a given semantic environment Θ is a double partial differential of the semantic function $f_\theta(P)$ on the sets of variables S and executing steps T , i.e.:

$$\begin{aligned} \theta(P) &= \frac{\partial^2}{\partial t \partial s} f_\theta(P) \\ &= \mathbf{R}_{k=1}^{n-1} \left\{ \left[\frac{\partial^2}{\partial t \partial s} f_\theta(P_k) \right] r_{kl} \left[\frac{\partial^2}{\partial t \partial s} f_\theta(P_l) \right] \right\}, l = k + 1 \\ &= \mathbf{R}_{k=1}^{n-1} \left\{ \left[\mathbf{R}_{i=0}^{\#T(P_k)} \mathbf{R}_{j=1}^{\#S(P_k)} v_{P_k}(t_i, s_j) \right] r_{kl} \left[\mathbf{R}_{i=0}^{\#T(P_l)} \mathbf{R}_{j=1}^{\#S(P_l)} v_{P_l}(t_i, s_j) \right] \right\} \\ &= \begin{pmatrix} \mathbf{V}_{P_1} & & & \mathbf{V}_G \\ & \mathbf{V}_{P_2} & & \mathbf{V}_G \\ & & \ddots & \vdots \\ & & & \mathbf{V}_{P_{n-1}} & \mathbf{V}_G \end{pmatrix} \quad (23) \end{aligned}$$

where \mathbf{V}_{P_k} , $1 \leq k \leq n-1$, is a set of values of local variables that belongs to processes P_k , and \mathbf{V}_G is a finite set of values of global variables.

Example 3. The *semantics of the sequential relations* of processes in RTPA, $\theta(P \rightarrow Q)$, is a double partial differential of the semantic function $f_\theta(P \rightarrow Q)$ on the sets of variables S and executing times T , i.e.:

$$\begin{aligned} \theta(P \rightarrow Q) &= \frac{\partial^2}{\partial t \partial s} f_\theta(P \rightarrow Q) \\ &= \frac{\partial^2}{\partial t \partial s} f_\theta(P) \rightarrow \frac{\partial^2}{\partial t \partial s} f_\theta(Q) \\ &= \mathbf{R}_{i=0}^{\#T(P)} \mathbf{R}_{j=1}^{\#S(P)} v_{P_i}(t_i, s_j) \rightarrow \mathbf{R}_{i=0}^{\#T(Q)} \mathbf{R}_{j=1}^{\#S(Q)} v_{Q_j}(t_i, s_j) \\ &= \mathbf{R}_{i=0}^{\#T(P \rightarrow Q)} \mathbf{R}_{j=1}^{\#S(P \rightarrow Q)} v(t_i, s_j) \\ &= \begin{pmatrix} & \mathbf{s}_P & \mathbf{s}_Q & \mathbf{s}_{PQ} \\ \mathbf{t}_0 & \perp & \perp & \perp \\ (\mathbf{t}_0, \mathbf{t}_1) & V_{1P} & - & V_{1PQ} \\ (\mathbf{t}_1, \mathbf{t}_2) & - & V_{2Q} & V_{2PQ} \end{pmatrix} \\ &= \begin{pmatrix} \mathbf{V}_P & \mathbf{V}_{PQ} \\ & \mathbf{V}_Q & \mathbf{V}_{PQ} \end{pmatrix} \quad (24) \end{aligned}$$

where $P \square Q$ indicates a concatenation of these two processes over time, and in the simplified notation of the matrix, $V_P = v(t_P, s_P)$, $0 \leq t_P \leq n_P$, $1 \leq s_P \leq m_P$; $V_Q = v(t_Q, s_Q)$, $0 \leq t_Q \leq n_Q$, $1 \leq s_Q \leq m_Q$; and $V_{PQ} = v(t_{PQ}, s_{PQ})$, $0 \leq t_{PQ} \leq n_{PQ}$, $1 \leq s_{PQ} \leq m_{PQ}$.

A semantic diagram of the sequential process relation as defined in Eq. 24 is illustrated in Fig. 4 on the semantic environment Θ , where S_{PQ} is the set of shared or global variables of statements P and Q .

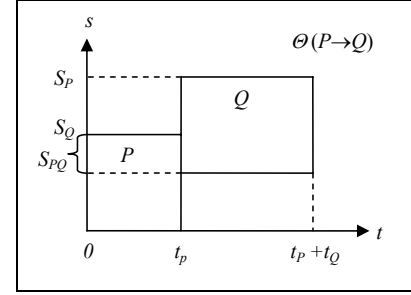


Fig. 4 The semantic diagram of sequential processes

Definition 14 can be applied to derive the deductive semantics of any complex process built by the compositional rules as listed in Table IV. Then, the semantics of a program can be deduced to the combination of semantics of a set of processes, each of which can be further deduced to the composition of all statements' semantics as described below.

Definition 15. The *semantics of a program* \wp , $\theta(\wp)$, on a given semantic environment Θ , is a combination of the semantic functions of all processes $\theta(P_k)$, $1 \leq k \leq n$, i.e.:

$$\begin{aligned} \theta(\wp) &= \mathbf{R}_{k=1}^{\#K(\wp)} \frac{\partial^2}{\partial t \partial s} f_\theta(\wp) \\ &= \mathbf{R}_{k=1}^{\#K(\wp)} \theta(P_k) \\ &= \mathbf{R}_{k=1}^{\#K(\wp)} \left[\mathbf{R}_{i=0}^{\#T(P_k)} \mathbf{R}_{j=1}^{\#S(P_k)} v_{P_k}(t_i, s_j) \right] \quad (25) \end{aligned}$$

where $\#K(\wp)$ is the number of processes or components in the program.

It is noteworthy that Eq. 25 will usually result in a very large matrix of semantic space, which can be quantitatively predicated as follows.

Definition 16. The semantic space of a program $S_\theta(\wp)$ is a product of $\#S(\wp)$ variables and $\#T(\wp)$ executing steps, i.e.:

$$\begin{aligned} S_\theta(\wp) &= \#S(\wp) \bullet \#T(\wp) \\ &= \sum_{k=1}^{\#K(\wp)} \#S(\wp_k) \bullet \sum_{k=1}^{\#K(\wp)} \#T(\wp_k) \quad (26) \end{aligned}$$

The semantic space of programs provides a useful measure of software complexity. Due to the tremendous

size of the semantic space, both program composition and comprehension are innately a hard problem in terms of complexity and cognitive difficulty.

Deductive semantics and laws of software developed in this section can greatly simplify the description and analysis of the semantics of complex software systems in various programming and specification languages. Deductive semantics can be used to define both abstract and concrete semantics of large-scale software systems, facilitate software comprehension and recognition, support tool development, enable semantics-based software testing and verification, and explore the semantic complexity of software systems.

V. CONCLUSIONS

The product-based metaphor for software development has dominated the methodologies of software engineering in the last four decades. However, the failure in explaining the nature of software and in ensuring its quality indicates the need for new theories and practices. A new informatics metaphor has been proposed to explain the fundamental characteristics of software. A comprehensive set of informatics, cognitive properties and laws has been identified in this paper, which answers an age-long fundamental question on what constrains software. A new approach towards formal semantics known as deductive semantics has been developed on the basis of the compositional model of program, and the mathematical model of software semantics. It has been demonstrated that the semantic properties and laws of software can be used to greatly simplify semantic analyses in software engineering on the basis of deductive semantics.

The findings of this paper will serve as a coherent part of foundations in software engineering and computer science. A wide range of new software notations, processes, quality principles, verification techniques, and organizational methodologies in software engineering may be developed based on the informatics and semantic properties and laws of software.

REFERENCES

- [1] Aho, A.V., R. Sethi, and J.D. Ullman (1985), *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publication Co., New York.
- [2] Bjorner, D. and C. B. Jones (1982), *Formal Specification and Software Development*, Prentice Hall, Englewood Cliffs, NJ.
- [3] Cutnell, J.D. and K.W. Johnson (1998), *Physics*, 4th ed, John Wiley & Sons, Inc. New York.
- [4] Dijkstra, E.W. (1975), Guarded Commands, Nondeterminacy, and the Formal Derivation of Programs, *Communications of the ACM*, Vol.18, No.8, pp.453-457.
- [5] Goguen, J. and G. Malcolm (1996), *Algebraic Semantics of Imperative Programming*, MIT Press.
- [6] Gries, D. (1981), *The Science of Programming*, Springer-Verlag, New York.
- [7] Hartmanis, J. (1994), On Computational Complexity and the Nature of Computer Science, 1994 Turing Award Lecture, *Communications of the ACM*, Vol.37, No.10, pp.37-43.
- [8] Hoare, C.A.R. (1969), An Axiomatic Basis for Computer Programming, *Communications of the ACM*, Vol.12, No.10, pp.576-580.
- [9] Hoare, C.A.R. (1985), *Communicating Sequential Processes*, Prentice-Hall Inc., London.
- [10] Louden K.C. (1993), *Programming Languages: Principles and Practice*, PWS-Kent Publishing Co., Boston.
- [11] McDermid, J. A., ed. (1991), *Software Engineer's Reference Book*, Butterworth-Heinemann Ltd., Oxford, UK.
- [12] Schmidt, D. (1988), *Denotational Semantics: A Methodology for Language Development*, Wm . C. Brown Publishers, Dubuque, IA.
- [13] Scott, D. (1982), Domains for Denotational Semantics, In *Automata, Languages and Programming IX*, Springer-Verlag, Berlin, pp. 577-613.
- [14] Slonneg, K. and B. Kurts (1995), *Formal Syntax and Semantics of Programming Languages*, Addison-Wesley Pub. Co.
- [15] Wang, Y. and King, G. (2000), *Software Engineering Processes: Principles and Applications*, CRC Press, Boca Raton, FL, 752 pp.
- [16] Wang, Y. (2002), The Real-Time Process Algebra (RTPA), *Annals of Software Engineering: An International Journal*, Vol. 14, Kluwer Academic Publishers, pp. 235-274.
- [17] Wang, Y. (2002), On Cognitive Informatics, Keynote Lecture, *Proc. of 1st IEEE International Conference on Cognitive Informatics (ICCI'02)*, Calgary, Canada, IEEE CS Press, August, pp.34-42.
- [18] Wang, Y. (2003), On Cognitive Informatics, *Brain and Mind: A Transdisciplinary Journal of Neuroscience and Neurophilosophy*, Vol.4, No.2, pp.151-167.
- [19] Wang, Y. (2003), Using Process Algebra to Describe Human and Software System Behaviors, *Brain and Mind*, Vol. 4, No. 2, pp. 199-213.
- [20] Wang, Y. and Y. Wang (2006), Cognitive Informatics Models of the Brain, *IEEE Trans. on Systems, Man, and Cybernetics (Part C)*, Vol. 36, to appear.
- [21] Wang, Y. (2004), On the Big-R Notation: Describing Interactive and Recursive Behaviors and Architectures in

Computing, *Technical Report of TESERC, Univ. of Calgary*, TESERC04-006, Oct., pp. 1-9.

- [22] Wang, Y. (2006), *Software Engineering Foundations: A Multidisciplinary Perspective*, CRC Software Engineering Series, Vol.2, CRC Press, CRC Press, Boca Raton, FL, to appear.
- [23] Wang, Y. (2005), On the Mathematical Laws of Software, *Proceedings of the 18th Canadian Conference on Electrical and Computer Engineering (CCECE'05)*, Saskatoon, SA, Canada, May 1-4, pp. 1086-1089.



Yingxu Wang is a Professor of Cognitive Informatics and Software Engineering and Director of Theoretical and Empirical Software Engineering Research Center (TESERC) at the University of Calgary. He received a PhD in Software Engineering from The Nottingham Trent University, UK, in 1997, and a BSc in Electrical Engineering from Shanghai Tiedao University in 1983.

He was a Visiting Professor in the Computing Laboratory at Oxford University during 1995, and has been a full professor since 1994.

Dr. Wang is a Fellow of WIF, a P.Eng of Canada, a Senior Member of IEEE with the Societies of Computer, SMC, and Communications, and a member of ACM, ISO/IEC JTC1, and the Canadian Advisory Committee (CAC) for ISO. He is the founder and steering committee chair of the annual IEEE International Conference on Cognitive Informatics (ICCI). He is Editor in Chief of the International Journal of Cognitive Informatics and Natural Intelligence (IJCI&NI), Editor in Chief of the World Scientific Book Series on Cognitive Informatics, and Editor of the CRC Book Series in Software Engineering. He was the Chairman of the Computer Chapter of IEEE Sweden during 1999-2000. He has accomplished a number of EU, Canadian, and industry-funded research projects as principal investigator and/or coordinator, and has published over 260 papers and 6 books in software engineering and cognitive informatics. He has served on numerous editorial boards and program committees, and as guest editors for a number of academic journals. He has won dozens of research achievement, best paper, and teaching awards in the last 25 years, particularly the IBC 21st Century Award for Achievement “*in recognition of outstanding contribution in the field of Cognitive Informatics and Software Science.*”