Beyond UML and MDA/MDD/MDE: Software Modeling Revolution Based on Complexity Science
- An Introduction to NSM (Nonlinear Software Modeling approach)

Jay Xiong

NSEsoftware, LLC. USA

jay@nsesoftware.com; jayxiong@yeah.net

Abstract

“Model driven considered harmful” (Harry Sneed); all existing Model-Driven approaches are outcomes of reductionism and superposition principles that the whole of a system is the sum of its parts, so that with them almost all software modeling activities are performed linearly, partially, and locally; they offer Top-Down software development processes without upstream movement at all, making the defects introduced into a software product in the requirement development phase and the design phase easily propagate to the software maintenance phase; they violate the law of the nature of human-being - people are nonlinear and easy to make mistakes which need to be corrected by themselves; they violate the essential principles of complexity science, particularly the Holism principle and the Nonlinearity principle; they try to use linear approaches to resolve the issues of a nonlinear software system where a small change can result in large differences to a later state - Butterfly-Effect.

This paper introduces a nonlinear, holistic, global, and dynamic software modeling approach based on complexity science, driven by platform-independent Java source code or the source code of platform-dependent programming languages, with which one kind of source (the source code) is used for human understanding of a complex software through new type models meaningful for representing high-level abstractions and suitable to be automatically generated from the source code of stub programs or regular programs, and computer understanding of the complex software using the source code itself or the transformed source code. NSM brings revolutionary changes to software modeling by shifting the foundation from reductionism and superposition principle to complexity science, enabling software design to become pre-coding and coding to become further design (Top-Down + Bottom-Up), and making the generated models dynamically traceable to the requirements and the source code, indirectly executable through the source code for dynamic defect removal using a new software testing method called Transparent-box innovated, and always consistent with the source code. Working with the other components of NSE (Nonlinear Software Engineering paradigm), this approach makes it possible to help software organizations improve their product quality by several orders of magnitude, double their productivity and project success rate, and halve their cost Simultaneously.

With NSM it is true that “The Code is the Design” (Jack W. Reeves).

Keywords: software modeling, MDE, MDA, MDD, software requirement engineering, software design, coding, testing, quality assurance, maintenance

Abbreviations:
UML: Unified Modeling Language
MDE: Model-Driven Engineering
MDA: Model-Driven Architecture
MDD: Model-Driven Development
CMMI: Capability Maturity Model Integration
SEMAT: Software Engineering Method and Theory
OMG: Object Management Group
NSE: Nonlinear Software Engineering paradigm, innovated by me (Jay Xiong) and implemented by me and my colleagues
3J Graphics: J-Chart, J-Diagram, and J-Flow which are new types of chart/diagrams innovated by me
Transparent-Box: a new software testing method innovated by me and implemented by me and my colleagues
NSM: Nonlinear Software Modeling approach
LSM: Linear Software Modeling approaches
1. Introduction

What is a model? “A model of a system is a description or specification of that system and its environment for some certain purpose” (OMG). “A model is an abstraction of a (real or language based) system allowing predictions or inferences to be made.” (Thomas Kühne, http://www.mm.informatik.tu-darmstadt.de/staff/kuehne_old/publications/papers/what-is-a-model-dagstuhl.pdf)

With the existing software modeling approaches, called LSM (Linear Software Modeling approaches) here, two kinds of sources are used – one is in models/diagrams drawn manually or using graphic editors for human understanding of a complex software product, while another one is in textual format for computer understanding of the complex software product.


“MDA – the future of UML

The future of UML may be a recent OMG initiative called Model Driven Architecture (MDA)... MDA defines a vision for how software can be developed based on models... In MDA software is produced through a series of model transformations aided by an MDA modeling tool. An abstract computer-independent model (CIM) is used as basis for a platform-independent model (PIM). The PIM is transformed into a platform-specific model (PSM) that is transformed into code.”

But about MDA, Harry Sneed pointed out [2] that

“Model driven considered harmful

* Model-driven tools magnify the mistakes made in the problem definition
* Model-driven tools create an additional semantic level to be maintained
* Model-driven tools distort the image of what the program is really like
* The model cannot be directly executed. It must first be transformed into code which may behave other than expected.
* Model driven tools complicate the maintenance process by creating redundant descriptions which have to be maintained in parallel.
* Model driven tools are designed for top-down development.
* Top-down functional decomposition creates maintenance problems”

* Summary:
  • If a UML design can really replace the programming code as envisioned by Jacobson in his paper, UML all the way down, then it becomes just another programming language.
  • The question then comes up as to what is easier to change
    – The design documents or
    – The programming language
  • This depends on the nature of the problem and the people trying to solve it. If they are more comfortable with diagrams, they can use diagrams. If they are more comfortable with text, they should write text.
  • Diagrams are not always the best means of modeling a solution. A solution can also be described in words. The important thing is that one model is enough – either the code or the diagrams. They should be reproducible from one another.”.

The proposed NSM approach driven by source code is the solution to have source code as the single source for both human understanding of a complex software product with much better graphics (holistic, colorful, dynamic, virtual, interactive, traceable, linkable, auto-convertible, accurate, precise, and always consistent with the source code) automatically generated from the source code, and computer understanding of the complex software product with the source code. NSM makes design become pre-coding, and coding become further design (Top-Down plus Bottom-Up). It is performed through forward engineering using stub programs having dummy modules with an empty body or some simple logic and/or some function call statements, and reverse engineering using regular source code. Different from all existing software modeling approaches offering static modeling capability only, NSM offers dynamic modeling and engineering capability with program execution and dynamic testing to greatly improve the ability to remove critical defects introduced into a software product in the entire software life-cycle including the requirement development phase and the product design phase particularly through the applications of the innovated Transparent-box software test method combining structural testing and functional testing together seamlessly. With NSM software maintenance is performed nonlinear, holistically, globally, and quantitatively with side-effect prevention in the implementation of requirement changes or code modifications. NSM works closely with the other components of NSE (Nonlinear Software Engineering paradigm [3]) to offer revolutionary solutions to solve the critical issues existing with software development.

2. Are All Existing Model-Driven Approach Moving to a Wrong Direction?

In my opinion, all existing model-driven approaches for software development are moving to a wrong direction with no bright future but wasting time and efforts. The reasons are simple:

(1) They are outcomes of linear thinking, reductionism, and the superposition principle that the whole of a complex system is the sum of its parts, so that with them all software modeling and engineering tasks are performed linearly, partially, and locally;
(2) They offer sequential engineering processes for Top-Down software development without upstream movement at all, making the defects introduced in upstream easy to propagate to downstream – the defect removal cost increases tenfold several times;
(3) They violate the law of the nature of human-being - people are nonlinear and easy to make mistakes in thinking, working, reading and writing something, etc., so that there is a need for them to correct the mistakes made by themselves in upstream process when the team are working in downstream process;

(4) They are outdated (even if they have not been completely implemented yet), because they do not catch up with science research progress – they violate the essential principles of complexity science (the greatest achievement in science research after Relativity and Quantum Mechanics), particularly the Nonlinearity Principle, the Dynamic Principle, and the Holism Principle that the whole of a complex system is greater than the sum of its components, the behaviors and characteristics of the whole emerge from the interaction of its components and the interaction between it and the environment.

(5) Believe or not, all existing model-driven approaches will be failed sooner or later - the reason is: they try to use linear approaches to solve the critical issues existing with a nonlinear system – software engineering paradigm and a software product (with many interactive components working in a changing environment dynamically) where a small change at one place can result in large differences to a later state of the entire software product – “Butterfly-Effect”.

In my opinion, no matter how excellent is a new software development approach or methodology, if it offers linear solutions to a few components of the software engineering paradigm, it will not be able to make significant contribution (not to mention revolutionary contribution) in resolving the critical issues existing with today’s software development (low quality and productivity and project success rate, and high cost and risk), including

* MDA (Model-Driven Architecture), MDD (Model-Driven Development), and MDE (Module-Driven Engineering) which focus on software modeling improvement;
* CMMI (Capability Maturity Model Integration [4]) which focuses on software engineering process improvement (the failure rate of the implementation of CMMI/CMMI is about 70% [5]);
* Agile software development methods which weaken software design and documentation to make the developed software products much more difficult to maintain (unless they are used for very small projects), and
* SEMAT (Software Engineering Method and Theory, http://www.semat.org/bin/view ) which focuses on software development methodology evolution.

No, it is impossible for any one of them to make significant contribution in resolving those critical issues from theory and practices:

(1) **from theory**: according to the Holism Principle of complexity science, the behaviors and characteristics of a complex system emerge from the interaction among its all components, and the interaction between the system and its working environment, can not be inferred simply from the behavior of its any individual part. Software engineering paradigm itself is a complex system consisting of many components including the models, the engineering processes, the software development methodology, the software testing paradigm, the software quality assurance paradigm, the documentation paradigm, the software maintenance paradigm, the software project management paradigm, and more, so that, such as MDE, only improving the models and the code generation can not bring revolutionary changes to the whole of the software engineering paradigm;

(2) **from practices**: It is also easy to understand from practices that the existing software engineering paradigm is entirely an outcome of linear thinking, reductionism, and superposition principle that the whole of a complex system is the sum of its parts, so that with it almost all software development tasks and activities are performed linearly, partially, and locally - its all components are outdated, do not comply with the essential principles of complexity: the software testing paradigm is outdated because most critical software defects are introduced into a software product in the requirement development phase and the design phase, but the dynamic testing of the product is performed after coding – as concluded by NIST (National Institute of Standards and Technology) that “Briefly, experience in testing software and systems has shown that testing to high degrees of security and reliability is from a practical perspective not possible. Thus, one needs to build security, reliability, and other aspects into the system design itself and perform a security fault analysis on the implementation of the design.” (“Requiring Software Independence in VVSG 2007: STS Recommendations for the TGDC,” November 2006 http://vote.nist.gov/DraftWhitePaperOnSlimVVSG2007-20061120.pdf ); the software quality assurance paradigm is outdated which depends on inspection and software testing after production, violating Deming’s total quality principle that “Cease dependence on inspection to achieve quality. Eliminate the need for inspection on a mass basis by building quality into the product in the first place.” [6]; the maintenance paradigm is outdated with which software maintenance is performed linearly, blindly, partially, and locally – “Over three decades ago, software maintenance was characterized as an ‘iceberg’. We hope that what is immediately visible is all there is to it, but we know that an enormous mass of potential problems and cost lies under the surface. In the early 1970s, the maintenance iceberg was big enough to sink an aircraft carrier. Today, it could easily sink the entire navy!”[7], “the fundamental problem with program maintenance is that fixing a defect has a substantial (20-50 percent) chance of introducing another.”[8] – it means that without making revolutionary changes to the other parts of the existing software engineering paradigm including the software testing paradigm, the software quality assurance paradigm, the software maintenance paradigm, the software documentation paradigm, the software visualization paradigm, the software project management paradigm, it is impossible for MDE or MDA or MDD or CMMI or Agile, or SEMAT to make significant contribution to solve the major critical issues existing with today’s software development. In fact all of them are also outcomes of linear thinking, reductionism, and superposition principle – with those approaches there is no upstream movement at all in the software development process as shown in Fig. 1 (the MDA Agile process). The products developed with those approaches are un-maintainable: any requirement change or code modification will make the products unstable day by day.  

3
The major drawbacks of LSM

About UML and the model driven approaches, more criticisms/concerns have been made by many people:

“The elaboration of use-case scenarios is stupid” [9]

“one of the disadvantages of this approach is the significant amount of work burden the developer with specifying all the constraints supporting the transformation” [10]

“A central tenet of MDD is that there are multiple representations of artifacts inherent in a software development process, representing different views of levels of abstraction on the same concepts. To the extent that these are manually created, duplicate work and consistency management are required. A similar problem was found in the software verification work of the 1970s and 1980s, which required two different versions of the same software to be written—one for specification and one for execution.” [11]

“MDA concerns

Some key concepts that underpin the MDA approach (launched in 2001) were first elucidated by the Shlaer-Mellor method during the late 1980s. Indeed a key absent technical standard of the MDA approach (that of an action language syntax for Executable UML) has been bridged by some vendors by adapting the original Shlaer-Mellor Action Language (modified for UML. However during this period the MDA approach has not gained mainstream industry acceptance; with the Gartner Group still identifying MDA as an "on the rise" technology in its 2006 ‘Hype Cycle’, and Forrester Research declaring MDA to be "D.O.A." in 2006. Potential concerns that have been raised with the OMG MDA approach include:

- Incomplete Standards: The MDA approach is underpinned by a variety of technical standards, some of which are yet to be specified (e.g. an action semantic language for xtUML), or are yet to be implemented in a standard manner (e.g. a QVT transformation engine or a PIM with a virtual execution environment).

- Vendor Lock-in: Although MDA was conceived as an approach for achieving (technical) platform independence, current MDA vendors have been reluctant to engineer their MDA toolsets to be interoperable. Such an outcome could result in vendor lock-in for those pursuing an MDA approach.

- Idealistic: MDA is conceived as a forward engineering approach in which models that incorporate Action Language programming are transformed into implementation artifacts (e.g. executable code, database schema) in one direction via a fully or partially automated “generation” step. This aligns with OMG's vision that MDA should allow modeling of a problem domain's full complexity in UML (and related standards) with subsequent transformation to a complete (executable) application. This approach does, however, imply that changes to implementation artifacts (e.g. database schema tuning) are not supported. This constitutes a problem in situations where such post-transformation “adapting” of implementation artifacts is seen to be necessary. Evidence that the full MDA approach may be too idealistic for some real world deployments has been seen in the rise of so-called "pragmatic MDA". Pragmatic MDA blends the literal standards from OMG's MDA with more traditional model driven mechanisms such as round-trip engineering that provides support for adapting implementation artifacts. … [12]
There are more drawbacks of the Model-Driven approaches offering Top-Down software development approaches, pointed out by software experts [2]:

- McCracken/Jackson: Life-Cycle Concept Considered Harmful. “System requirement can never be stated fully in advance, not even in principle, because the user doesn’t know them in advance - not even in principle”.

- Gladden: Conventional Life-Cycle Approach Exacerbates Maintenance Problem. “Each modification to the requirements adversely effects the system by impacting each subsequent task ... the result is a vicious cycle compounding the maintenance problem ... requirements are always incomplete when development begins”.

- Balzer: Specification and Implementation are Intertwined. “In actual practice developments steps are not refinements of the original specification, but instead redefine the specification itself ... there is a much more intertwined relationship between specification and implementation than the standard rhetoric would have us believe”.

- Rich/Waters: “Writing a complete specification in a general-purpose specification language is seldom easier and often incredibly harder than writing a program. Furthermore, there has been little success in developing automatic systems that compile efficient programs form specifications”.

- Demillo, Perles, Lipton: “If a formal program is transformed from an informal specification then the transformation process itself must necessarily be informal ... in the end, the program itself is the only complete and accurate description of what the program will do”.

- Fetzer: “From a methodological point of view programs are mere conjectures and testing is an attempted and all too frequent successful refutation ... reasoning about programs tends to be non demonstrative, implicative and non additive ...”.

- Perles: “People must plunge into activities that they do not understand and people cannot create perfect mechanism...”.

- Sneed: “The only way to make the specification a complete and accurate description of the program is to reduce it to the semantic level of the program. However, in so doing, the specification becomes semantically equivalent to the program”.

Here, I want to add the following points to LSM approaches:

1. As I mentioned above, those approach are outcomes of linear thinking, reductionism, and the superposition principle, so that with them all software modeling activities are performed linearly, partially, and locally to produce many small pieces of models/diagrams with the big picture of an entire software product being missing.

2. They are based on Constructive Holism principle that software components are developed first, then, “Assemble the product from the product components, ensure the product, as integrated, functions properly and deliver the product.” [4]

3. The code generated by them is incomplete unless the models are ugly enough with a lot of detailed information to make themselves hard to view and understand (It is why I don’t believe Executable UML [13] will be a real solution).

4. Without bi-directional, automated, and self-maintainable traceability, the models/diagrams provided are hard to review and inspect for static defect removal.

5. The models/diagrams provided are not executable for debugging, and not testable for dynamic defect removal - their quality is hard to ensure.

6. The models/diagrams provided do not virtually exist, require a lot of space to store and a long time to display.

7. It is hard to keep the models/diagrams consistent with the source code.

8. They can do nothing for legacy software products.

9. They offer two kinds sources approaches to software modeling (Fig. 2) with one in models/diagrams format for people to understand a complex software system, while another one in textual format (the source code) for computer “understand” the software system – there is a big gap between the two sources.
3. The Challenges and the Solutions for Code-Driven Software Modeling (Fig. 3)

As we all know, traditional code-driven engineering approaches do not support software modeling for high-level abstraction, making the developed software product hard to understand and hard to maintain.

The Challenges for NSM:

(1) New types of models/diagrams are needed which should be meaningful for describing high-level abstraction (such as the relationship of an actor and the actions performed by the actor, or the relationship of an event and the corresponding responses) and/or low-level program logic, and suitable to be automatically and completely generated from the platform-independent Java (Java-DSL) source code or even a platform-dependent language (such as C++) source code.

(2) The new types of models/diagrams automatically generated should be holistic, colorful, interactive, dynamic, and traceable.

(3) Making the platform-independent Java (Java-DSL) programming language or even a platform-dependent programming language suitable (without changes) to be used for high-level abstraction to generate the new types of models/diagrams.

(4) Developing fully automated tools to generate the new types of models/diagrams directly from the source code of the platform-independent Java (Java-DSL) program or even a platform-dependent program.

(5) A holistic model/diagram (such as a call graph for an entire software product) can be very big and very complicated, so that it should be generated dynamically and virtually without being stored in memory or a hard disk using the hard copies, with capability to highlight an element and all of the related elements.

(6) A new software testing method should be innovated to combine functional testing and structural testing together seamlessly, can be dynamically used in requirement development phase and software design phase – in the case where there is no an output from a program being tested dynamically, it should be able to help users check whether the real execution path covers an expected path or the critical part of an expected path specified in control flow.

(7) A facility for establishing automated and self-maintainable traceability among all related models/diagrams/documents and test cases and source code should be innovated to make a developed software product truly maintainable.

The solutions:

(1) New types of models/diagrams are innovated by me (Jay Xiong) to meet the needs, which are called 3J-Graphics: J-Chart, J-Diagram, and J-Flow to be described in detail later.

(2) The 3J-Graphics can be generated directly from the source code of the platform-independent Java programs or a platform-dependent program written in C, C++, or VB. They are holistic, colorful, interactive, dynamic, and traceable to and from the source code.

(3) With the 3J-Graphics and the corresponding tools, the platform-independent Java (Java-DSL) programming language or even a platform-dependent programming language is suitable (without changes) to be used for high-level abstraction to generate the new types of models/diagrams such as the Actor and the Action notations similar to the Use Case diagram of UML.

(4) A set of fully automated tools have been developed for directly generating the new types of models/diagram from a dummy source code using dummy modules or a regular program in reverse engineering.
(5) All models/diagrams are generated dynamically and virtually without storing a hard copy in the computer memory or the hard disks. Users can select an element from the generated models/diagrams to highlight it with all related elements, or trace it to highlight an untested path with most untested branches, etc..

(6) A new software testing method called Transparent-box testing method has been innovated by me and implemented by me and my colleagues, which combines functional testing and structural testing together seamlessly, can be used dynamically in the entire software development lifecycle, including the requirement development phase and the design phase too – to each test case for testing a software product, the corresponding tool will not only check whether the output (if any, can be none in requirement development phase and the software design phase) is the same as what is expected, but also help users check whether the program execution path cover the expect path or some critical segments of the expected path specified in J-Flow diagrams.

(7) With the innovated Transparent-box testing method, the corresponding tools can also automatically establish the traceability among the related models/diagrams/documents and test cases and the source code based on the test case description. The automated and self-maintainable traceabilities are holistic, bi-directional, accurate, precise, and dynamic – when a test case is selected by a user to perform forward tracing or traced when a user selects a source module or even only one code statement to perform backward tracing, not only the corresponding models/diagrams/documents will be opened from the location specified by the corresponding bookmarks, but a corresponding program will also be executed to handle some models/diagrams/documents generated by a third party with special formats, or play back the captured GUI operations for regression testing, and so on. With these kinds of traceabilities, software maintenance can be performed nonlinearly, holistically, and globally with side-effect prevention in the implementation of requirement changes or code modifications.

4. The Foundation of NSM

The foundation of NSM is complexity science. NSM complies with the essential principles of complexity science, particularly the Nonlinearity principle and the Holism principle (see Fig. 4 the framework for NSM development), so that with NSM almost all modeling tasks and engineering tasks are performed holistically and globally.

Fig. 3 NSM (Nonlinear Software Modeling approach)
5. Description of NSM

As shown in Fig. 3, with NSM, one kind source is used for both human understanding and computer understanding of a software product. The models/diagrams are automatically generated from the source code - either a dummy program using dummy modules (having an empty body, or only a set of function call statements), or a regular program through reverse engineering. The generated models/diagram and the source code are traceable.

We have the right to be wrong, but we also have the right to be right: NSM makes design become pre-coding, and coding become further design (Top-Down plus Bottom-Up).

With LSM, there is a one-time design process that complies with the linear process models (either an one time waterfall model, or an iterative/incremental model which is “a series of Waterfalls”[14], etc.) without upstream movement at all – the designers have no right to be wrong. But we are human beings rather than God – people are nonlinear, and it is easy for people to make mistakes in thinking, reading, writing, hearing, decision-making, etc.

With NSM, we have the right to be wrong in the design, but we also have the right to be right – for instance, in the coding phase, if we find something wrong with the product design, then we correct them through coding – later on we can easily update the design by rebuilding the database for automatically generating all related design documents/diagrams – NSM making design become pre-coding, and coding become further design (Top-Down plus Bottom-Up). NSM process model is shown in Fig. 5.
As shown in Fig. 5, usually customers only know their partial requirements at the beginning of a software product development, they need a learning process for them to understand the requirements, so that software requirement changes and new requirements will happen often which should be implemented with side-effect prevention through bi-directional traceability. For this purpose, all versions (including incomplete versions) of a software product being developed should be executable for customers to try to get their feedbacks. If multiple implementation is required, a platform-independent programming language such as Java should be used, otherwise the target programming language such as C or C++ can be used directly for both high-level system abstraction and low-level program design. It is true that most target programming languages are not designed for high-level system abstraction, but using special graphical notations we can easily extend a target programming language to become a high-level system abstraction programming language without changing it at all.

With NSM, three types of interactive and traceable diagrams (J-Chart, J-Diagram, and J-Flow) can be automatically generated at any phase from the source code of a stub program (in forward engineering) or a regular program (in reverse engineering) as shown in Fig. 6.

5-1 3J-Graphics (J-Chart, J-Diagram, and J-Flow) Innovated

The dynamic and interactive J-Chart/J-Diagram/J-Flow is a trinity: an Object-Oriented chart/logic diagram/Control flow diagram, the chart/diagram generator which is always running when the chart/diagram is shown, and the interface (using the chart/diagram itself) between the generator and the users for the users to control the chart/diagram dynamically with a multi-way on-line traceability/cross reference facility through which the users can view the related objects easier and better.

J-Chart

J-Chart not only can be used to represent the class inheritance relationship, the function call graph, and the class-function coupling structure graphically of a software product, but can also be used to display incremental unit test order or the related test coverage and quality data in bar graphics overlaid on each module-box to help users view the overall results of testing and quality measurement. J-Chart is useful in software modeling, system understanding, inspection, test planning, test result display, re-engineering, and software maintenance.

J-Chart can be automatically generated from a dummy program in “Bone Programming” for high-level abstraction, or a regular program including a legacy program.

A comparison between J-Chart and the most traditional call graphs

<table>
<thead>
<tr>
<th>J-Chart</th>
<th>Traditional Call Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is it holistic for directly showing a very complex software product?</td>
<td>Yes</td>
</tr>
<tr>
<td>Is it interactive for highlighting a path or getting related information?</td>
<td>Yes</td>
</tr>
<tr>
<td>Is it traceable such as to highlight a module with the all related modules?</td>
<td>Yes</td>
</tr>
<tr>
<td>Is it supported to use a module as the root</td>
<td></td>
</tr>
</tbody>
</table>
Can a bar-chart be added to a module-box to show related information? Yes No
Can the source code be directly edited from a module-box? Yes No
Can the logic diagram be linked from a module-box? Yes No
Can the control flow diagram be linked from a module-box? Yes No
Can a bottom-up coding orders be assigned to the modules? Yes No

When used for software version comparison, can different colors be used to show “un-changed” modules”, “changed modules”, “deleted modules”, and “added new modules” separately? Yes No

*1: Some tools claim that they can provide dynamic function call graph, but I have not seen their application examples.

J-Chart notations are shown in Fig.7.

<table>
<thead>
<tr>
<th>J-Chart</th>
<th>Notations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Press the right mouse button to pop up a function menu.</td>
</tr>
<tr>
<td>Class</td>
<td>Member function</td>
</tr>
<tr>
<td></td>
<td>Press the right mouse bottom to pop up a member function menu.</td>
</tr>
<tr>
<td>Macro</td>
<td>Press the right mouse bottom to pop up a function menu.</td>
</tr>
<tr>
<td></td>
<td>Overloading non-member function</td>
</tr>
<tr>
<td></td>
<td>Press the right mouse button to pop up an overloading menu</td>
</tr>
<tr>
<td>Class</td>
<td>Overloading member function and virtual function</td>
</tr>
<tr>
<td></td>
<td>Press the right mouse button to pop up a function menu.</td>
</tr>
<tr>
<td>Class</td>
<td>Overloading member function</td>
</tr>
<tr>
<td></td>
<td>Press the right mouse button to pop up a function menu</td>
</tr>
<tr>
<td>Class</td>
<td>Virtual function</td>
</tr>
<tr>
<td></td>
<td>Press the right mouse button to pop up a function menu</td>
</tr>
<tr>
<td>Class</td>
<td>Press the right mouse button to pop up a class menu</td>
</tr>
<tr>
<td>Template Class</td>
<td>Press the right mouse button to pop up a class menu</td>
</tr>
</tbody>
</table>

**J-Diagram**

J-Diagram not only can be automatically generated from source code in all levels including the class hierarchy tree, class structure diagram, and the class member function logic diagram with un-executed class/function/segments/condition outcomes being highlighted, but also can be automatically linked together for an entire software product to make the diagrammed code traceable in all levels. J-Diagram can be automatically converted into J-Flow diagram. J-Diagram is particularly useful in Object-Oriented software understanding, inspections, walkthroughs, testing, and maintenance.
### The major differences between J-Diagram and most Flow Charts

<table>
<thead>
<tr>
<th></th>
<th>J-Diagram</th>
<th>Flow Charts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is it structured?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can it show an entire software product very complex?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Is it uniqueness?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Is the location of the program logic indicated?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can it show the result of test coverage measurement?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can it show the branch execution frequency?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Does it offer traceability between related elements?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can it be converted to a control-flow diagram?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Does it exist virtually without huge space to store?</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

J-Diagram notations are shown in Fig. 8.
Fig. 8 J-Diagram notations
J-Flow

Most traditional control flow diagrams are un-structured. They often use the same notation to represent different program logic, and cannot display the logic conditions and the source code locations. J-Flow diagram is Object-Oriented and structured, uses different notations to represent different logic with capability to show logic execution conditions and the corresponding source code locations. J-Flow is particularly useful in logic debugging, path analysis, test case and code correspondence analysis, and class/function level test coverage result display with unexecuted elements (path, segments, and unexecuted condition outcomes) highlighted.

J-Flow diagram can be converted to and from J-Diagram automatically. With NSM, J-Flow not only can be used to show a program control flow, but also can be used to show a “best path for testing” with most untested branches and the execution conditions automatically extracted for semi-automatic test case design in unit testing.

J-Flow plays an important role in software traceability among all related documents (requirements specification, models/diagrams, test requirements, etc.) and test cases and source code. All the traceability operations use J-flow forwardly and backwardly. When a user select a requirement and click a related test case used to test that requirement implementation forwardly from a window where the test case (test case description) will be automatically shown in blue color, and the corresponding test coverage result will be shown in J-Flow in another window with the classes, functions, and branches being tested by that test case being highlighted in red color.

The major differences between J-Flow and traditional control flow diagram:

<table>
<thead>
<tr>
<th></th>
<th>J-Flow</th>
<th>Traditional Control Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is it structured?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can it show an entire software product very complex?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Is it uniqueness?</td>
<td>Yes</td>
<td>No (arbitrary)</td>
</tr>
<tr>
<td>Is the source code locations of the control flow indicated?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can it show the result of test coverage measurement?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can it show the branch execution frequency?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can it be automatically converted to a logic diagram?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can it highlight a path with most untested elements?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Does it exist virtually without huge space to store?</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

J-Flow notations are shown in Fig. 9.
Besides J-Graphics, an ActionPlus diagram (using lines rather blocks to show program logic) can also be generated.
5-2 The HAETVE Technique


With HAETVE the graphical notations for representing an actor and an action using platform-independent Java programming language are shown in Fig. 10 where the notation used for representing an actor is originally designed for representing a recursive program module.

![Actor and Action Notations for Java](image)

Fig. 10 Notations for representing an actor and an action for Java

The corresponding source code in Java dummy programming is listed as follows:

```java
public class notations {
    public static void Bank_Customer () {
        Bank_Customer ();
    }
    public static void Deposit_Money () {
    }
}
```

Why use Java? It is because that Java is a platform-independent programming language, so that the results obtained in the modeling are also independent from the target languages and platforms. If there is a need, the dummy java source code can be transformed to a target language source code.

The notations for representing an actor and an action using C/C++ programs are shown in Fig. 11.

![Actor and Action Notations for C/C++](image)

Fig. 11 Notations for representing an actor and an action for C/C++

The corresponding dummy source code written in C/C++ is listed as follows separately:

```c
Void Bank_Customer ()
{
    Bank_Customer ();
}

Void Deposit_Money ()
{
}
```

For the Actor-Action type applications, HAETVE is similar to the Use Case approach [4], and is easy to map to Use Case notations as shown in Fig. 12 and Fig. 13.
An example of a use-case diagram with an actor and three use cases.

Fig. 1 Notation mapping between Use Cases (A) and HAETVE (B)

The analysis result of Use Cases can also be mapped to HAETVE as shown in Fig. 12.
An application example is shown in Fig. 15 and Fig. 19.

Fig. 15 An application example of Use Case Analysis

Fig. 16 The result mapping to the Use Case Analysis shown in Fig. 15:

(A) Using regular function notations;
(B) Using Class member function notations

UML Activity diagram can be mapped to J-Diagram as shown in Fig. 17 and Fig. 18:
A sample programming source code used for representing the corresponding product design specified in the activity diagram shown in Fig. 18 is listed as follows:

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <iomanip.h>
#include "my_include.h"

void main(int argc, char** argv)
{
    user_login_name = get_user_login_name();
    saved_user_password = get_saved_user_password();
    if(!user_registered())
        /* handling user register. */;
    else /* user enters Email and Password. */;
    is_true =1;
    while(is_true)
    {
        if(!strcmp(saved_user_password, user_password))
            /* user Enters Email and Password. */;
        else /* user Enters Email and Password. */;
        is_true =1;
    }
    if(!strcmp(saved_user_password, user_password))
        /* user Enters Email and Password. */;
    else /* user Enters Email and Password. */;
    is_true =1;
    while(is_true)
    
    /* system presents selection list; user selects a title; system presents details about selected book; */
    if(user_want_to_buy_this_book())
        { /* user adds this book to shopping car; */
            if(user_wants_to_continue_shopping())
            else break;
        }
    else /* user displays notification; */;
    /* user processes to check out. */;
    if(the_shipping_address_changed_since_last_order())
        /* user enters new address. */;
    else /* user confirms address. */;
    /* user selects shipping options. */;
    if(user_uses_same_credit_card())
        /* user confirms the card. */;
    else /* user adds new credit card information. */;
    if(user_wants_cancel_the_order())
        { /* user cancels order process. */
            exit;
        }
    /* user confirms order; system returns confirmation number. */;
    break;
    }
    cout << "This program has been executed successfully.";
}
```

The included file:

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <iomanip.h>
#include "my_include.h"

void main(int argc, char** argv)
{
    user_login_name = get_user_login_name();
    saved_user_password = get_saved_user_password();
    if(!user_registered())
        /* handling user register. */;
    else /* user enters Email and Password. */;
    is_true =1;
    while(is_true)
    {
        if(!strcmp(saved_user_password, user_password))
            /* user Enters Email and Password. */;
        else /* user Enters Email and Password. */;
        is_true =1;
    }
    if(!strcmp(saved_user_password, user_password))
        /* user Enters Email and Password. */;
    else /* user Enters Email and Password. */;
    is_true =1;
    while(is_true)
    
    /* system presents selection list; user selects a title; system presents details about selected book; */
    if(user_want_to_buy_this_book())
        { /* user adds this book to shopping car; */
            if(user_wants_to_continue_shopping())
            else break;
        }
    else /* user displays notification; */;
    /* user processes to check out. */;
    if(the_shipping_address_changed_since_last_order())
        /* user enters new address. */;
    else /* user confirms address. */;
    /* user selects shipping options. */;
    if(user_uses_same_credit_card())
        /* user confirms the card. */;
    else /* user adds new credit card information. */;
    if(user_wants_cancel_the_order())
        { /* user cancels order process. */
            exit;
        }
    /* user confirms order; system returns confirmation number. */;
    break;
    }
    cout << "This program has been executed successfully.";
}
```

The included file:

```c
class Customer_passwd {
private:
    char login_name[80];
    char password[80];
public:
    int check_password(char* login_name)
    int change_password(char* login_name, char* password) {};
};
```

```c
class Customer_infor: Customer_passwd {
private:
```
char name[80];
char shipping_address[80];

public:
    int update_info(char* name, char* shipping_address){}
/* other member function. */
};

char* user_password;
char* saved_user_password;
char* user_login_name;
int there_is_at_least_one_selection;
int is_true;

int m=1,n=3;
char* p="abc";

int user_registred(){return m;}
char* get_saved_user_password(){return p;}
char* get_user_login_name(){return p;}
int user_want_to_buy_this_book(){return m;}
int user_wants_to_continue_shopping(){return n;}
int the_shipping_address_changed_since_last_order(){return n;}
int user_uses_same_credit_card(){return n;}
int user_wants_cancel_the_order(){return n;}
int user_enter_search_criteria(){return n;}

The J-Diagram generated from the listed dummy programming source code is shown in Fig. 18.
Fig. 18 The J-Diagram generated from the listed dummy source code to map to the activity diagram of UML shown in Fig. 17, shown with the test coverage measurement result after dynamic testing.

With HAETVE, a sample event-responds notations is shown in Fig. 19.
As shown in Fig. 20, with HAETVE, a class is represented in:

1. J-Chart
2. J-Diagram
3. J-Flow diagram
4. ActionPlus diagram

With HAETVE a Time-Event relationship is shown in comments as follows:
/* Time-Event table:

<table>
<thead>
<tr>
<th>Timing</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events</td>
<td>Event1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>event2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>event3</td>
</tr>
<tr>
<td>Timing</td>
<td>t4</td>
<td>t5</td>
<td>t6</td>
</tr>
<tr>
<td>Events</td>
<td>Event4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>event5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>event6</td>
</tr>
</tbody>
</table>

...*/

A Rich List of Models/Diagrams which Can Be Generated Automatically from source code

Besides the Actor-Action diagram, the Event-Responds diagram, and the Activity diagram, all the following models/diagrams can also be automatically generated from a regular program or a legacy software product:

A. In System-Level

(1) The function call graph of an entire software system
(2) The class inheritance chart of an entire software system
(3) The class and independent function relation chart of an entire software system
(4) The program tree of an entire software system
(5) The overall MC/DC test coverage measurement result of an entire software system shown in J-Chart
(6) The overall performance measurement result of an entire software system shown in J-Chart

(7) The overall Cyclomatic complexity measurement result of an entire software system shown in J-Chart
(8) The logic diagram of an entire software system shown in J-Diagram.
(9) The logic diagram of an entire software system shown in J-Diagram with test coverage and untested elements highlighted
(9) The control flow diagram of an entire software system shown in J-Flow.
(10) The control flow diagram of an entire software system shown in J-Flow with test coverage and untested elements highlighted
(11) The logic diagram of an entire software system shown in ActionPlus diagram.
(12) The logic diagram of an entire software system shown in ActionPlus diagram with test coverage and untested elements highlighted (see Fig. 21).
(13) The overall version comparison result shown in J-Chart with un-changed modules shown in blue, changed modules in red, deleted modules in brown, and added modules in green.
Fig. 21 An ActionPlus diagram shown with test coverage and untested elements highlighted.

Fig. 22 shows a typical system-level logic diagram with related information represented in J-Diagram.

Fig. 22 A typical system-level logic diagram shown with related information in J-Diagram.
B. In File-Level

(1) The logic diagram of a source file shown in J-Diagram.
(2) The logic diagram of a source file shown with test coverage and untested elements highlighted
(3) The control flow diagram of a source file.
(4) The control flow diagram of a source file shown in J-Flow with test coverage and untested elements highlighted
(5) The logic diagram of a source file shown in ActionPlus diagram
(6) The logic diagram of a source file shown in ActionPlus diagram with test coverage and untested elements highlighted
(7) The version comparison result of a source file shown with un-changed files shown in blue, changed files in red, deleted files in brown, and added files in green. (see Fig. 23).

C. In Class-level

(1) The logic diagram of a class shown in J-Diagram.
(2) The logic diagram of a class shown with test coverage and untested elements highlighted (note: a class can not be directly executed, so that the test coverage data of a class is collected from its instances).
(3) The control flow diagram of a class.
(4) The control flow diagram of a class shown in J-Flow with test coverage and untested elements highlighted
(5) The logic diagram of a class shown in ActionPlus diagram
(6) The logic diagram of a class shown in ActionPlus diagram with test coverage and untested elements highlighted

D. In Function–level/Module-level

(1) The logic diagram of a function shown in J-Diagram.
(2) The logic diagram of a function shown with test coverage and untested elements highlighted (note: a class can not be directly executed, so that the test coverage of a class is collected from its instances).
(3) The control flow diagram of a function.
(4) The control flow diagram of a function shown in J-Flow with test coverage and untested elements highlighted
(5) The logic diagram of a function shown in ActionPlus diagram
(6) The logic diagram of a function shown in ActionPlus diagram with test coverage and untested elements highlighted
(7) An untested path with most untested branches being highlighted in J-Flow diagram with the path execution conditions

Fig. 23 File-level software version comparison
being extracted for semi-automated test case design (see Fig. 2).

Fig. 2
An untested path with most untested branches being highlighted in J-Flow diagram with the path execution conditions being extracted.

5-3 Beyond UML

5-3-1 Holistic

With HAETVE, the models/diagrams can be generated from an entire software product with its all source code written in the platform-independent Java language or a platform-dependent programming language to show the product structure, the class relationship, the and overall static And dynamic properties of the product.

A sample J-Chart representing the call graph of an entire software product is shown in Fig. 25.
A J-Chart shown with the Cycromatic complexity (the number of decision statements) measurement result of an entire software product is proved in Fig. 26.
5-3-2 Fully Automatable

With NSM all models/diagrams can be automatically and completely generated from the source code.

Many models/diagrams of UML are not automatable, such as the Use Cases diagram, because they are outside of the software systems being developed as shown in Fig. 27.

![Fig. 27 The actors of a Use Cases diagram are outside the software product being developed](image)

Different from UML models/diagrams, with HAETVE, all parts of a models/diagram are inside the software system being developed. For instance, the Actors of the Actor-Action relationship diagram are represented inside the software being development for easy to perform requirement validation and verification, but will not affect the program execution as shown in Fig. 28, Fig. 29, Fig. 30, and Fig. 31.

![Fig. 28 HAETVE notations mapping to Use Cases shown in Fig. 26.](image)

![Fig. 29 Static check of the Actor-Action relationship within a software product being development](image)
Actors inside a software product being developed can be used to perform dynamic requirement validation and verification as shown in Fig 30, even if there are only two layers with the system body and some sub-system bodies only designed before implementation.

Fig. 30 Dynamic check example 1 of the actor-action relationship without affecting the product dynamic behaviors

(a) Fig. 31 Dynamic checks example 2 of the Actor-Action relationship:

   (a) Found an error – the “New_Order” function does not executed after testing -
      The problem comes from a typing mistake:
      if(strcmp(argv[1],"New_Ordor")==0)
              New_Order();
   (b) After correcting the mistake (changing New_Ordor to New_Order) a correct result is obtained
5-3-3 Traceable for highlighting an element with the related elements

With HAETVE, the generated models/diagram are traceable to make them much easy to review for static defect removal. An application example of the traceability is shown in Fig. 32.

![Fig. 32 An example shown with the traceability in a call graph shown in J-Chart notations.](image)

An application example for static defect removal through the traceability is shown in Fig. 33.

![Fig. 33 Defect removal through review with traceability](image)

(a) A defect found; (b) The defect found is removed
5-3-4 Dynamic

Different form the UML models/diagrams which are statically exist, the models/diagrams automatically generated with HAETVE exist dynamically:

(1) They are dynamically generated from the tiny database in real time when requested by the users, and displayed dynamically;

(2) When a user moves a model/diagram shown on a window from a location to another location, there is not real models/diagram movement at all, but only a new one is dynamically generated from the new location – so that, for instance, moving a holistic J-Diagram showing the program logic of an entire software product from block 100 to block 100000 will take the same operation time as moving it from block 100 to block 101 – only moving one block.

5-3-5 Virtual

With HAETVE, the models/diagrams are virtually existing with no hardcopies stored in the computer memory and the hard disk (unless the users want to print them in postscript or other format) to greatly reduce the space needed. When a holistic model such as the call graph of an entire software product is requested by a user to display it on a window, only the corresponding part of the chart will be dynamically generated to fit the size of the window – no more or less. But from a user’s point of view, there is no difference as the entire chart is shown. As mentioned above, when there is a need for the user to move the chart from one location to another one, a new part of the chart will be dynamically generated and displayed from the new location.

5-3-6 Interactive

With HAETVE, when a model/diagram is shown, the corresponding model/diagram generator is always working and waiting for users’ operation commands through the interface – using the model/diagram itself. For instance, when a call graph of an entire software product is shown in J-Chart, the users can request the corresponding chart generator to show extra information such the code test coverage, the percentage of the run time spent in each function, etc.. Fig. 34 shows the rich operation options for a generated J-Chart.

Fig. 34 The rich operation options provided with J-Chart

Fig. 35 shows how a user can select any function box as the new root to generate a sub-call graph from a J-Chart showing the call graph of an entire software product.
**Fig. 35** An application example of the interactive J-Chart for generating a sub-call graph

**Fig. 36** shows the process for the users to request the model/diagram generator to display the location where a runtime error happened (shown with an “EXIT” word added) in a J-Flow diagram.

If a diagram is better than a thousand words in describing a complex system, then an interactive and dynamic diagram will be much better than ten thousand words in representing the complex system.
5-3-7 With HAETVE, the automatically generated models/diagrams can not only represent the static characteristics of a software product, but can also represent the dynamic measurement results of a software product.

With HAETVE, the models/diagrams are executable through the source code, and automatically generated from the source code and the database with dynamic program measurement data, so that the generated models/diagram can not only represent the static characteristics of a software product, but can also show the dynamic measurement results of the product.

A call graph and a program logic diagram shown with the MC/DC (Modified Condition/Decision Coverage) test coverage is provided in Fig. 37.

Fig. 37 An application example using J-Chart automatically generated to show the percentage of the code tested for each module, and J-Diagram automatically generated to show the MC/DC (Modified Condition/Decision Coverage) test coverage measurement results with the untested conditions and branches highlighted in small black boxes.

A call graph shown in J-Chart notation with the program performance measurement result is provided in Fig. 38.

Fig. 38 A call graph shown with the percentage of the execution time spent in each module, and the execution times for each module.
5-3-8 **Being able to represent the message sending and receiving between class instances**

With HAETVE, the generated J-Diagram representing the program logic of a program can also represent the message sending and receiving between class instances through automated and self-maintainable traceability - see Fig. 39 an application example.

![Diagram](image)

Fig. 39 An application example of message sending and receiving between class instances represented in a J-Diagram with the automated and self-maintainable traceability

5-3-9 **Always Consistent with the Source Code**

With HAETVE, all models/diagrams are automatically generated from the source code, so that they are always consistent with the source code. After code modification, a updated database can be rebuilt to keep the consistency between the models/diagrams and the source code.
6. Beyond MDE/MDA/MDD

With NSM there are something beyond MDE/MDA/MDD.

6-1 Top-down Plus Bottom-up Development

With NSM, software design becomes pre-coding, and coding becomes further design (Top-down + Bottom-up) – see Fig. 40.

For instance, clicking on a module-box from the generated call graph to directly edit the source code of that module, as shown in Fig. 41.

Fig. 40 Design and coding with NSM

Fig. 41 Directly coding from a call graph generated in design process
Coding can also become re-design of the software product as shown in Fig. 42 and Fig. 43.

Fig. 42 Two function call statements are added in the coding process of the state4::transition (unsigned char) module designed without using them before.

Fig. 43 After rebuilding the database, the corresponding design documents are updated.

### 6-2 Incremental Development Support

NSM supports incremental development as shown in Fig. 44 and Fig. 45.

Fig. 44 Incremental development process (1)

Fig. 45 Incremental development process (2)
6-3 Dynamic Testing Performed in all Phases in the Software Development Lifecycle from the First Step Down to the Retirement of a Software Product

For instance, at the beginning of an EDA product hierarchy design, only one module - the main() function for handling command-line options may be designed with the source code listed as following:

```c
#include <stdio.h>
#include <string.h>

void main(int argc, char** argv)
{
    int ERROR_CODE;
    if(argc != 3 && argc != 4)
        printf("Error found in the command-line.\n");
    else if (argc == 3)
        if(strcmp(argv[1],"global_placement")==0)
            // calling g_placement(argv[2]);
        else if(strcmp(argv[1],"global_routine")==0)
            // calling g_routing(argv[2]);
        else if(strcmp(argv[1],"detailed_placement")==0)
            // calling d_placement(argv[2]);
        else if(strcmp(argv[1],"detailed_routing")==0)
            // calling d_routing(argv[2]);
        else if(strcmp(argv[1],"partitionning")==0)
            // calling partitioning(argv[2]);
        else if(strcmp(argv[1],"ordering")==0)
            // calling ordering(argv[2]);
    else
        // calling printf("Invalid name: %s\n",argv[1]);
    } else if (strcmp(argv[2],"dbs_build") == 0)
        // calling dbs_build(argv[2],argv[3]);
    else printf("Error! Invalid name: %s\n",argv[1]);
}
```

Of course, there is no real meaningful output with the dummy main() function, so that it is not suitable to perform functional testing using traditional Black-box test method.

For being able to dynamically test a software product in the all phases of the software development lifecycle, a new software testing method called Transparent-box testing method (Fig. 46) has been innovated which combines functional testing and structural testing together seamlessly – to each test case used to dynamically test a program, the corresponding tool will not only check whether the output (if any, can be none) is the same as what is expected, but also help users check whether the real program execution path covers the expected execution path or the critical part of the expected execution path specified in control flow, and then automatically establishes the bi-directional traceability among all related models/documents and test cases and the source code through Time Tags automatically inserted into both the test cases description part and the code test coverage database for mapping them together, and some keywords written into the comment part of a test case description to indicate the format of a related model/diagram/document, the model/diagram/document/file path, and the bookmark used to open the model/diagram/document from the specified location when the test case is selected for forward tracing, or traced from a source module or a branch when a user clicks on the related module or branch shown in J-Flow diagram backwardly.

![Fig. 46 The innovated Transparent-box testing method](image)
So, even if there is only the main() function, we can perform dynamic testing using the Transparent-box method. Fig. 47 shows an application example with the test results and the untested elements highlighted in small black box/point.

6-4 Making All Modes, Diagrams, Documents and Test Cases and the Source Code Traceable Bi-directionally

The facility for automated and self-maintainable traceability is shown in Fig. 48. This kind traceability not only can be established automatically, but also can be updated automatically – when a test case or the source code is modified, after re-built the database and running the test case, the corresponding traceability will be updated without manual modification.

The simple rules for designing the corresponding test cases are as follows:
(a) An empty line means a separator between different test cases.
(b) A `#` character at the beginning position of a line means a comment.
(c) Within comments, users can use some keywords such as @WORD@, @HTML@, @EXCEL@, @PDF@, and @BAT@ to indicate the format of a document, followed by the full path name of the document, and a bookmark (used for finding inconsistent defects).
(d) Within comments, users can use [path] and [/path] pair to indicate the expected execution path using control flow notation (segment numbers) for a test case (used for finding logic defect).
(e) Within comments, users can use Expected Output to indicate the expected value to be produced (for finding functional defects).
(f) Within comments, users can also use [Not_Hit] and [/Not_Hit] marks to indicate modules or branches (segments) which are prohibited for the related test case execution to enter (used for protection of some special modules/segments).
(g) After the comment part, there is a line to indicate the directory for running the corresponding program.
(h) The final line(s) in a test case description is the command line(s) (which may start a program with the GUI) and the options.

Usually, with NSE in the beginning of product design, some documents should be ready for use, including the corresponding requirement specification, the test requirement specification, the prototyping documents, the product development plan, etc., so that according to the test requirement specification and the command-line options (GUI operation options), we can design a corresponding test script file as follows:

```plaintext
# An example of using transparent-box to prevent defects in requirement analysis phase
# and initial design phase of a software product development.
# test case 1
# test purpose: to find the inconsistency among all related documents for the global placement sub-system
# The related requirement specification: @HTML@ C:\EN_transparent_box\new_EDA_specifications.htm#G_PLACEMENT
# The related prototype design document: @WORD@ C:\EN_transparent_box\prototyping.doc bmname g_placement
# The related development plan: @BAT@ C:\EN_transparent_box\ganttprof.bat
# The expected execution path:
# [path] main (int, char**) {s0, s2}  # [NOT_HIT] !path ![NOT_HIT]
# Expected output: none
# The directory and the execution command:
C:\EN_transparent_box  
new_EDA global_placement -dbs=my_dbs
  
# test case 2
# test purpose: to find the inconsistency among all related documents for the global routing sub-system
# of a software product, NEW_EDA.
# The related requirement specification: @HTML@ C:\EN_transparent_box\new_EDA_specifications.htm#G_ROUTING
# The related test requirement specification: @WORD@ C:\EN_transparent_box\Test_Requirement_Specification.doc
# The related prototype design document: @WORD@ C:\EN_transparent_box\prototyping.doc bmname g_router
# The expected execution path:
# [path] main (int, char**) {s0, s3}  # [NOT_HIT] !path ![NOT_HIT]
# The expected output: none
# The directory and the execution command:
C:\EN_transparent_box  
new_EDA global_routing -dbs=my_dbs
  
# test case 3
# test purpose: to find the inconsistency among all related documents for the detailed placement sub-system
# of a software product, NEW_EDA.
# The related requirement specification: @HTML@ C:\EN_transparent_box\new_EDA_specifications.htm#D_PLACEMENT
# The related prototype design document: @WORD@ C:\EN_transparent_box\prototyping.doc bmname d_placement
# The related development plan: @BAT@ C:\EN_transparent_box\ganttprof.bat
# The expected execution path:
# [path] main (int, char**) {s0, s4}  # [NOT_HIT] !path ![NOT_HIT]
# Expected output: none
# The directory and the execution command:
C:\EN_transparent_box  
new_EDA detailed_placement -dbs=my_dbs
  
# test case 4
# test purpose: to find the inconsistency among all related documents for the detailed routing sub-system
# of a software product, NEW_EDA.
# The related requirement specification: @HTML@ C:\EN_transparent_box\new_EDA_specifications.htm#D_ROUTING
# The related prototype design document: @WORD@ C:\EN_transparent_box\prototyping.doc bmname d_routing
# The related development plan: @BAT@ C:\EN_transparent_box\ganttprof.bat
# [path] main (int, char**) {s0, s5}  # [NOT_HIT] !path ![NOT_HIT]
# Expected output: none
C:\EN_transparent_box  
new_EDA detailed_routing -dbs=my_dbs
```

39
Now it is the time to perform defect prevention and defect propagation prevention through dynamic testing and review using traceable documents and the test cases and the source code:

After running the test cases, we can perform forward tracing and backward tracing to find and remove defects – see Fig. 49 to Fig. 50.

Fig. 49 Tracing the second test case will find two inconsistency defects:
(1) The real execution path did not cover the expected execution path specified as main(int, char**) {s0, s3} – but segment s3 is highlighted as untested;
(2) The bookmark for opening the global routing section of the prototyping document, g_router, pointed to the wrong section – the global placement section.

Removing the defects:
(1) find the location for the first defect and modify the main() program - modifying

```
FROM:

    else if(strcmp(argv[1],"global_routine")==0)
    ; // calling  g_routine(argv[2])
```

```
TO:

    else if(strcmp(argv[1],"global_routine")==0)
    ; // calling g_routin(argv[2]);
```

Fig. 50 Locating the mistake of the bookmark, g-router
(2) find the mistake related to the bookmark, g_router (see Fig. 51), and fix it (see Fig. 52):

Fig. 51 Fixing the bookmark setting mistake

After removing the two defects and rebuilding the database, a correct new result is obtained as shown in Fig. 52.

Fig. 52 Verifying that the two defects have been removed through backward tracing from segment s3 (the test case 2 was traced and the related documents were opened without defects)

6-5 Whole Lifecycle Quality Assurance mainly through Defect Prevention and Defect Propagation Prevention

With NSM, the whole lifecycle quality assurance is mainly based on defect prevention and defect propagation prevention through:

(1) Dynamic testing using the Transparent-box method (see section 6-4);
(2) Inspection and review using traceable documents and source code;
(3) Software visualization.

As shown in Fig. 53 the difference between traditional quality assurance approaches and NSE/NSM - with NSM the software quality is ensured with the priority as:

(1) Defect prevention
(2) Defect propagation prevention
(3) Refactoring with side-effect prevention
(4) Deep and broad testing and quality measurement
An application example of defect prevention between a function call statement and the called function is shown in Fig. 5 through incremental unit coding and testing with a bottom-up coding order assigned automatically in the call graph shown in J-Chart.

Working with the NSE Software Quality Assurance paradigm (NSE-SQA) closely, the NSM modeling approach makes it possible to help software development organizations improve their product quality by several orders of magnitude.

Table 2 shows a comparison result in the efficiency about various existing software quality assurance technologies which are all offered by NSE with great improvements (Note: the item and the data written in italics come from the published reports provided by Software Productivity Research based on the analysis of 12000 software projects [15]).

But it is needed to point out that besides those existing software quality assurance technologies, NSM with the entire NSE paradigm offers much more new technologies to ensure the quality of a software product developed:

1. Defect prevention performed in all phases
2. Defect propagation prevention in all phases
3. Refactoring for too complex modules with side-effect prevention in the implementation of module changes
4. Dynamic testing using the innovated Transparent-box method in all phases
5. Software maintenance performed nonlinearly, holistically, and globally with side-effect prevention in the implementation of requirement changes or code modifications, and more.

The conclusion is that compared with the existing linear approaches, it is possible for NSM with NSE to help software organizations improve their product quality by several orders of magnitude.
Table 2 The possibility for NSM (working with NSE-SQA) to help to help software development organizations remove 99.99% of the defects in their software products.

<table>
<thead>
<tr>
<th></th>
<th>Defect Removal Technology</th>
<th>Highest Defect Removal Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Requirement review</td>
<td>50%</td>
</tr>
<tr>
<td>NSM</td>
<td>Requirement review with traceable documents</td>
<td>&gt;50%</td>
</tr>
<tr>
<td>2</td>
<td>Top level design review</td>
<td>60%</td>
</tr>
<tr>
<td>NSM</td>
<td>Top level design review using traceable documents and charts</td>
<td>&gt;60%</td>
</tr>
<tr>
<td>3</td>
<td>Detailed functional design review</td>
<td>65%</td>
</tr>
<tr>
<td>NSM</td>
<td>Detailed functional design review using traceable documents</td>
<td>&gt;65%</td>
</tr>
<tr>
<td>4</td>
<td>Detailed logic design review</td>
<td>75%</td>
</tr>
<tr>
<td>NSM</td>
<td>Detailed logic design review using traceable diagrams</td>
<td>&gt;75%</td>
</tr>
<tr>
<td>5</td>
<td>Code inspection</td>
<td>85%</td>
</tr>
<tr>
<td>NSM</td>
<td>Code inspection with bi-directional traceability</td>
<td>&gt;85%</td>
</tr>
<tr>
<td>6</td>
<td>Unit testing</td>
<td>50%</td>
</tr>
<tr>
<td>NSM</td>
<td>Unit testing incrementally according to the assigned bottom-up testing order plus MC/DC test coverage analysis capability and graphical representation of the test result</td>
<td>&gt;50%</td>
</tr>
<tr>
<td>7</td>
<td>New function testing</td>
<td>65%</td>
</tr>
<tr>
<td>NSM</td>
<td>New function testing</td>
<td>&gt;65%</td>
</tr>
<tr>
<td>8</td>
<td>Integration testing</td>
<td>60%</td>
</tr>
<tr>
<td>NSM</td>
<td>Integration testing</td>
<td>&gt;60%</td>
</tr>
<tr>
<td>9</td>
<td>System testing</td>
<td>65%</td>
</tr>
<tr>
<td>NSM</td>
<td>System testing combining structural and function testing seamlessly</td>
<td>&gt;65%</td>
</tr>
<tr>
<td>10</td>
<td>External beta testing</td>
<td>75%</td>
</tr>
<tr>
<td>NSM</td>
<td>External beta testing with traceable documents and the source code</td>
<td>&gt;75%</td>
</tr>
<tr>
<td>11</td>
<td>Cumulative Efficiency</td>
<td>99.99%</td>
</tr>
<tr>
<td>NSM</td>
<td>Cumulative Efficiency with Defect prevention in the entire software development life cycle.</td>
<td>&gt;99.99</td>
</tr>
</tbody>
</table>

6-6 Complying with the NSE Software Development Method Based on Generative Holism principle

NSM complies with the NSE software development methodology supporting Top-Down plus Bottom-up software development (see Fig. 55.), driven by defect prevention and traceability, based on Generative Holism principle that the whole of a stub system (with dummy modules) will be development first as an embryo of the system, then the components of the system will be developed incrementally to make the whole grow up, so that all versions of the system can be made executable and customer testing can begin early. If a runtime error happens, it is much easy to locate the source of the error. With NSE software development method, software modeling will not end before the retirement of a software product, because the requirement changes will
The differences between Constructive Holism based software development and Generative Holism based software development is shown in Fig. 56.

More detailed information about NSE software development methodology can be found in Chapter 10 of the NSE-Book [3].

![Fig. 55 NSE software development methodology](image)

![Fig. 56 The differences between Constructive Holism based software development (a) and Generative Holism based software development (b)](image)

### 6-7 Following the Nonlinear NSE Software Engineering Process Model

NSM follows the NSE Software Engineering Process Model (see Fig. 57) which is nonlinear, incremental, and iterative bi-directionally to support Top-Down plus Bottom-up software development.

The major differences between the linear software engineering process models followed by LSM and the nonlinear NSE software engineering process model can be shown as the major difference of the one-way traffic with only one lane and the two-way traffic with multiple lanes (see Fig. 58).

More detailed information about the nonlinear NSE Software Engineering Process Model can be found in Chapter 8 of the NSE-Book [3].
The NSE Software Engineering Process Model

Fig. 57 The NSE Software Engineering Process Model

(a) An One-Way Traffic with only one lane; (b) A Two-Way Traffic with multiple lanes

The major differences between the one-way traffic with only one lane and the two-way traffic with multiple lanes: (a) An One-Way Traffic with only one lane; (b) A Two-Way Traffic with multiple lanes
6-7 Intelligent Version Comparison in System-level, File-level, and Module-level

A call graph shown in J-Chart notation with the version comparison result of an entire software product is proved in Fig. 59 where unchanged modules are shown in blue, changed modules in red, deleted modules in brown, and added modules in green. To a changed module, the users can further view the detailed differences in statement-level.

![Fig. 59 A J-Chart shown with the version comparison result of an entire software product](image)

6-8 Working with the NSE Maintenance Paradigm to Make Software Maintenance Being Performed Holistically and Globally with Side-Effect Prevention in the Implementation of Requirement Changes and Code Modifications

NSM works closely with the NSE software maintenance paradigm whose process model is shown in Fig. 60. With NSM software maintenance is performed holistically and globally with Side-Effect Prevention in the implementation of requirement changes and Code Modifications.

![Fig. 60 NSE Software Maintenance Process Model](image)

With LSM, software maintenance often takes more than 75% of the total effort and total cost in a software product because

1. LSM offers Top-Down engineering process and the dynamic software testing is performed after coding, so that software defects introduced in upstream are easy to propagate to software maintenance phase;
2. The implementation of requirement changes and code modifications is performed linearly, blindly, partially and locally – each time when a defect is removed, there is a 20-50 percent chance to introduce a new defect into the system [8] to make the system unstable day by day.
3. The regression testing is performed inefficiently using all test cases.
With NSM the effort and cost spent in software maintenance will be almost the same as that spent in the software development process because:

1. Both processes support requirement change and code modifications with side-effect prevention in the implementation of the changes;
2. The software quality is ensured through defect prevention and defect propagation prevention from the beginning of the product development to the end of the product;
3. Dynamic software testing is performed in the entire software development lifecycle using the Transparent-box testing method, so that only a few defects may propagate to the maintenance phase.

It is why with NSM about 50% of the total effort and the total cost in a software product development can be saved (see Fig. 61) – it is equal to double the productivity and halve the cost.

![Diagram 1](image1.png)
![Diagram 2](image2.png)

Fig. 61 Estimated effort and cost spent in software development and software maintenance

An application example is shown in Fig. 62 - The Implementation of a requirement change with side-effect prevention
Fig. 62 The implementation of a requirement change with side-effect prevention: (A) From the requirement to be changed to find the related test cases through the document hierarchy description table; (B) Perform forward tracing from the test case(s) to find the related program modules; (C) Perform backward tracing from the module(s) to be modified to find how many requirements are related (in this case, two requirements are related, so that the modification must satisfy both); (D) Tracing the module to be modified to find how many other modules are related (which may need to modify too) from the corresponding call graph; (E) Check the consistency between a modified module and the all statements calling it using the logic diagram automatically generated from the source code with traceability; (F) Tracing a modified source code segment (a set of statements with the same execution conditions) or a modified module to find the corresponding test case(s) which can be used to re-test it efficiently.
More detailed information about NSE software maintenance can be found in Chapter 18 of the NSE-Book [3].

6-8 Combining Software Development Process and Project Management Process Together Closely

NSE combines software development process and project management process together closely, making the project management documents (such as the project development schedule plan and the project cost reports) also traceable with the requirement implementation and the source code to find and fix problems in time.

Fig. 63 is an application example showing that the project development schedule plan is traced with the requirements through one of the corresponding test cases.

Fig. 63 A schedule chart traced and opened when performing forward tracing from the related requirement/test case

6-9 Support for Balancing the Development of Multiple Related Projects

With NSM, it is easy to balance the development of several related projects by making the progress report and the development schedule plan of a project traceable from and to another project (see Fig. 64).

Fig. 64 Multiple Project Management

6-10 Real Time Communication Support

NSM supports real time communication through project web sites and project BBS (Bulletin Board System) shared by the product development team internally or by the product development team and the customers for progress updates, handling unexpected events, new technology reviews, etc., and makes the pages of the project web site and the BBS traceable with the requirements and the source code. An application example is shown in Fig. 65.
Fig. 65 An application example of tracing a requirement/test case to open a web page from Google

6-11 Dynamic Requirement Validation and Verification


With NSM, dynamic validation and verification are support: from the recommended document relationship table created with bookmarks to specify the corresponding locations, users can easily find a requirement with all of the related documents and the test cases as shown in part (A) of Fig. 62.

With the automated and self-maintainable traceability, the process of requirement validation and verification is similar to the process for the implementation of requirement changes shown in Fig. 62 but with the forward tracing part only without the backward tracing.

Requirement validation and verification can be done statically to verify whether the required test cases are designed, and whether the corresponding source code is coded and tested by the related test cases – the code test coverage will be shown in J-Flow diagram in a windows with the tested elements highlighted in red color.

With NSM, Requirement validation and verification can also be performed dynamically, such as that when a related test case is selected to perform forward tracing to show what modules and branches can be tested, the GUI operations captured before when the test case was executed can be played back dynamically through the use of the @BAT@ keyword and the design of the corresponding batch file, so that the customers can view and review the dynamic behavior of the software product for the implementation of the related requirement. Application examples of dynamic requirement validation and verification are shown in Fig. 66 and Fig. 67.
Fig. 66 An application example of dynamic requirement validation. For validating a requirement (Bubble Sort)

Fig. 67 An application example used to validate another requirement (Quick sort)
Comparing Fig. 66 and Fig. 67 we can easily find that there is only one batch file re_run.bat used for all test cases. In this file there is only one command-line:

```
C:\isa_examples\play -hsi=C:\isa_examples\English_examples\analyzed_for_review\sortdemo\sortdemo.hsi -
dbspath=C:\isa_examples\English_examples\analyzed_for_review\sortdemo\dbs\ -
tdb=C:\isa_examples\English_examples\analyzed_for_review\sortdemo\sortdemo.tdb -
```

The command line shows that there is only a sortdemo.tdb file used for storing the captured GUI operations of the all test cases within the same test script file - how can the support platform Panorama++ plays the corresponding GUI operations only back selectively for the specific test case clicked by the user? It is done through the use of the different Time Tags (see the line 2 of the description of test case 2 in Fig. 66 and line 2 of the description of test case 4 in Fig. 67) which are automatically inserted into the description part of each test case for mapping the test cases and the corresponding test coverage data and the GUI operations captured.

Fig. 68 shows how time tags are automatically inserted into the test case script. Fig. 69 and Fig. 70 show application examples using the time tags for forward traceability, and Fig. 72 shows an application example of using the time tag to perform backward traceability.

---

**Fig. 68** How are Time tags automatically inserted into the test script

**Fig. 69** An application example of forward traceability established

**Fig. 70** Another application example of forward traceability established
An application example of backward traceability established

The major features of this kind of traceability innovated by me and implemented by me and my colleagues include:

1. Automatable
2. Self-maintainable – no matter the source code is modified or the test cases are changed or both are modified, after re-executing the software product with the test script, new time tags will be automatically inserted into both the description part of each test case and the corresponding source code test coverage database to map them again.
3. Accurate - this traceability facility is based on the dynamic execution of the test cases and test coverage measurement and the time tags to map the test cases and the source code tested, so that it is accurate.
4. Precise - this traceability facility is precise to the highest level – up to the code statement-level.
5. The established traceability is extended to include all related documents, all related project management documents, and web pages through the use of some keyword such as †HTML†, †WORD†, †EXCEL†, †PDF†, and †BAT† used in the description part of each test case to indicate the related document format, the file path and the bookmark to be used to open the traced documents from the corresponding position (see Fig. 66 and Fig. 67, the application examples).

This kind traceability is particularly useful for requirement validation and verification, defect prevention, and the software maintenance with side-effect prevention in the implementation of requirement changes or code modifications.

6-12 Automated Inconsistency Checking and Warning

With NSM, the models/diagrams automatically generated are always consistent with the source code through an inconsistency checking facility working with the corresponding database.

When the source code is modified after the model/diagram generation and a user try to view the models/diagrams again, a warning message will be posted to tell the user that the database should be updated with the new source code, and the old models/diagrams are prohibited to view anymore. After re-building the database, the updated models/diagrams can be viewed without a problem. Fig. 72 shows an application example step by step.
Fig. 72 An application example of inconsistency check and warning:
(a) Step 1: loading the makefile of a program
(b) Step 2: building the database
(c) Step 3: loading the database built
(d) Step 4: viewing the generated models/diagrams
(e) Step 5: making changes to the source code (the highlighted part are changes)
(f) Step 6: when a user tries to view the models/diagram generated again, a warning message is posted and the models/diagrams are prohibited to view
(g) Step 7: after updating the database, new models/diagrams can be viewed with the corresponding changes
6-13 Re-Modeling/Engineering/Reverse Engineering for Legacy Software Products

With NSM, the models/diagrams are generated from source code, so that a legacy software product can also be used to generate the models/diagrams through reverse engineering.

6-14 Efficiently Handling the Issue of Complexity, Inconsistency, Conformity, Invisibility, and Changeability

With NSM, the issues of Complexity, Inconsistency, Conformity, Invisibility, and Changeability can be handled efficiently. As pointed by Brooks,[8], Complexity, Inconsistency, Conformity, Invisibility, and Changeability are essential issues with software development which should be solved efficiently. Fig. 73 shows the new techniques innovated with NSM for handling those essential issues.

For instance, for handling the issue of changeability, any requirement change or code modification is implemented holistically and globally with side-effect prevention through various traceabilities in both the software product development and maintenance. More detailed information about these new techniques with NSM can be found in Chapter 6 of the NSE-Book [3].

6-15 Complete Integrated Tool Support

With NSM all of the techniques innovated for it are implemented and supported by the corresponding tools integrated into the Panorama++ product which is the NSE support platform.

6-16 Special Support for UML Users

As we all know that not all UML diagrams/notations are automatable or transformable to a platform-independent programming language such as the Java programming language.

For supporting some users who want to use NSM with some UML models/diagrams manually created or drawn using a graphic editor provided by a tool vendor, NSM allows the use of a batch file and the keyword @BAT@ in the comment part of the description of a test case to indicate the file path of the batch file – when the related test case is selected to perform forward tracing or the test cases is traced by a source module or a code branch (which can be tested by the test case) backwardly, the corresponding batch file will be dynamically executed.

In the case that a UML model/Diagram is stored in PDF format, the user can use the @PDF@ keyword in the description part of a test case to make it traceable with the related requirement and the source code. An application example is shown in Fig. 74 where a Use Cases diagram is traced with the corresponding requirement using the @PDF@ keyword written in the description part of a test case to indicate the format of the Use Cases file and the file path (the location of the file). At the same time, a J-Chart also be traced which not only maps to the Use Cases diagram, but also shows that the correspondent source code has been tested even if it is used for high-level abstraction of the product.

Fig. 75 shows a UML Use Cases diagram traced from the corresponding source code backwardly.
Fig. 74 An example of making UML models/Diagrams traceable with the requirement

Fig. 75 An example of making UML models/Diagrams traceable with the source code
7. The Major Features of NSM

The major features of NSM:

(1) NSM is based on complexity science, complying with the essential principles of complexity science, particularly the Nonlinearity Principle, the Dynamic Principle, and the Holism Principle, so that with NSM almost all software modeling and engineering activities are performed nonlinearly, holistically, and globally, rather than linearly, partially, and locally.

(2) NSM uses one kind source (the source code of a platform-independent programming language (such as Java/Java-DSL) or even a platform-dependent programming language) for human understanding of a complex software product through the colorful and meaningful Models/Diagrams automatically generated from the source code for high-level abstraction, and computer understanding of the complex software product using the source code or the transformed source code, so that with NSM the models/diagrams are always consistent with the source code.

(3) NSM makes design become pre-coding, and coding become further design – offering Top-Down plus Bottom-Up software development approach.

(4) NSM offers dynamic software modeling approach rather than static one: (1) with NSM, the generated models/diagrams are existing dynamically - when a chart or a diagram is shown, the corresponding generator is always working for users’ commands to operate to meet users’ needs through the interface – using the chart/diagram itself; (2) with NSM, the generated models/diagrams are dynamically executable through the corresponding source code; (3) with NSM, the generated models/diagrams are dynamically traceable to the requirements and he source code.

(4) NSM completely solves the inconsistency issues between the generated models/diagrams and the source code.

(5) NSM brings revolutionary changes to software modeling quality by making the generated models/diagrams traceable for static defect removal, and executable through the corresponding source code for dynamic defect removal.

(6) The models/diagrams generated with NSM are accurate and precise to the source code.

(7) With NSM a software product developed through nonlinear software modeling and engineering is much easier to understand, review, change, test, and maintain.

8. General Comparison

A general comparison between NSM and LSM is described in table 3.

<table>
<thead>
<tr>
<th>Item</th>
<th>NSM</th>
<th>LSM</th>
<th>Comments on LSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>The foundation</td>
<td>Complexity science</td>
<td>Reductionism and Superposition principle</td>
<td>The wrong foundation makes almost everything wrong</td>
</tr>
<tr>
<td>The software development approach supported</td>
<td>Nonlinear, Top-Down plus Bottom-Up – design becomes pre-coding, coding becomes further design</td>
<td>Linear, Top-Down only</td>
<td>Linear Top-Down software development approaches are outdated which violate the law of the nature of human-being that people are nonlinear and easy to make mistakes, making the defects introduced to a software product in upstream easy to propagate to the maintenance phase – the product are almost un-maintainable.</td>
</tr>
<tr>
<td>Complexity layers</td>
<td>One layer (using the platform-dependent programming languages) or two layers (using the platform-independent programming language, then the result is transformed to a platform-dependent programming language)</td>
<td>Three layers: the models layer, the platform-independent language layer, and the final platform-dependent programming language layer.</td>
<td>LSM makes a software product development process and the maintenance process much more difficult and complicated to manage and control.</td>
</tr>
<tr>
<td>Is the software development method(s) based on Generative Holism or Constructive Holism ?</td>
<td>Based on Generative Holism that the whole of the system will exist first as an embryo then grow up with its components incrementally to make all versions of the whole executable, so that system-testing and customer testing can begin before detailed coding</td>
<td>Based on Constructive Holism that the whole of the system will be built very late after entire production, so that system-testing and customer testing can not begin before the production – finding problems and fixing problems will be too late.</td>
<td>Customers are nonlinear people who do not know all the requirements in the beginning of a software product development, and do not really know what they want, so that it is important to make all versions of the product executable and deliverable to the customers for them to test and try.</td>
</tr>
<tr>
<td>Is it a dynamic software modeling approach or static software modeling approach?</td>
<td>With NSM, almost everything is dynamic: • The generated models/diagram exist</td>
<td>With LSM, almost everything is static: • The created models/diagrams exist</td>
<td>Static software modeling approach are outdated – they work slowly and inefficiently, not really useful to most software developers.</td>
</tr>
<tr>
<td>Question</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>-----</td>
<td>----</td>
<td></td>
</tr>
<tr>
<td>Is the dynamic software testing performed in all phases including the requirement development phase, the design phase, the coding phase, the testing phase, and the maintenance phase?</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Is the quality assurance being performed from the first step to the retirement of a software product?</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Is the software maintenance being performed nonlinearly, holistically, and globally with side-effect prevention through various traceability?</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Are the all models/diagrams automatable?</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dynamically</th>
<th>statically</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The model/diagram generation is dynamic (generating the models/diagrams according to the size of the window from the database)</td>
<td>• The models/diagrams are displayed through model/diagram loading rather than dynamic generation.</td>
</tr>
<tr>
<td>• The model/diagram display is dynamic</td>
<td>• The model/diagram traceability is static</td>
</tr>
<tr>
<td>• The model/diagram operation is dynamic</td>
<td>• The models/diagrams can only show the static behaviors of a software product.</td>
</tr>
<tr>
<td>• The model/diagram traceability is dynamic</td>
<td></td>
</tr>
<tr>
<td>• The model/diagram conversion (from one to another) is dynamic</td>
<td></td>
</tr>
<tr>
<td>• The models/diagrams generated not only can show the static behaviors of a software product, but also can show the product dynamic behavior such as the test coverage and the percentage of the run time spent in each module.</td>
<td></td>
</tr>
</tbody>
</table>

Most critical software defects are introduced into a software product in the requirement development phase and the design phase, but with LSM the dynamic testing can only be performed after coding - even if a defect introduced in the requirement development phase and fixed after coding, the cost has been increased tenfold several times. NIST (National Institute of Standards and Technology) concluded that “Briefly, experience in testing software and systems has shown that testing to high degrees of security and reliability is from a practical perspective not possible. Thus, one needs to build security, reliability, and other aspects into the system design itself and perform a security fault analysis on the implementation of the design.”

The quality assurance approach with LSM violate Deming’s product quality principle [6] that “Cease dependence on inspection to achieve quality. Eliminate the need for inspection on a mass basis by building quality into the product in the first place.”

A software product development through Top-Down approach offered by LSM is almost impossible to maintain – as pointed by Brooks that “The fundamental problem with program maintenance is that fixing a defect has a substantial (20-50 percent) chance of introducing another” – so the software product will become unstable day by day.

The models/diagrams created manually or drawn by a graphic editor take time and hard to change.

With LSM many small pies of...
<table>
<thead>
<tr>
<th>Question</th>
<th>Yes</th>
<th>No</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Are the created/generated models/diagrams holistic?</td>
<td>Yes</td>
<td>No</td>
<td>Models/diagrams are created without the entire picture of a software product, almost impossible to dynamically trace an element with its all related elements for static defect removal.</td>
</tr>
<tr>
<td>Are the created/generated models/diagrams dynamically traceable?</td>
<td>Yes</td>
<td>No</td>
<td>With LSM, the created models/diagrams are not dynamically traceable to the requirements and the source code – hard to maintain and hard to use in software maintenance.</td>
</tr>
<tr>
<td>Are the created/generated models/diagrams interactive?</td>
<td>Yes, almost all of them are interactive</td>
<td>No, almost all of them are static, not interactive</td>
<td>With LSM, most of the created models/diagrams are not interactive, so that the users can not order them to provide more related information, or highlight an element with all of the related elements for static defect removal, etc.</td>
</tr>
<tr>
<td>Can the created/generated models/diagrams be used to not only show the static properties of a software product, but can also show the dynamic behaviors based on the product execution?</td>
<td>Yes, such as the program test coverage and the percentage of runtime spent in each module</td>
<td>No</td>
<td>The created models/diagrams being able to show only the static properties of a software product are not good enough, not very useful in product quality management and the performance improvement.</td>
</tr>
<tr>
<td>Are the created/generated models/diagrams existing virtually?</td>
<td>Yes</td>
<td>No</td>
<td>With LSM, the models/diagrams are created manually or using an graphic editor and saved in memory and/or hard disk with hardcopies - not only needing a lot of space, but the display also taking long time.</td>
</tr>
<tr>
<td>Are the created/generated models/diagrams always consistent with the source code?</td>
<td>Yes</td>
<td>No</td>
<td>The created models/diagrams not consistent with the source code will be useless in software maintenance, also not useful in the understanding of the final product.</td>
</tr>
</tbody>
</table>

### 8. Conclusion

Existing linear software modeling approaches including MDE, MDA, and MDD are outcomes of linear thinking, reductionism, and superposition principle that the whole of a complex system is the sum of its components, so that with them almost all software modeling and engineering activities are performed linearly, partially, and locally, so that LSM will not be able to efficiently solve the critical issues existing with software development - low quality and productivity and project success rate, and high cost and risk.

This paper introduced NSM, a new software modeling approach which brings revolutionary changes to software modeling by shifting the foundation from reductionism and superposition principle to complexity science, enabling software design become pre-coding and coding become further design (Top-Down + Bottom-Up), and making the generated models/diagrams dynamically and bi-directionally traceable to the requirements and the source code, indirectly executable through the source code for dynamic defect removal using the Transparent-Box testing method innovated, and always consistent with source code. Working with the other components of NSE (Nonlinear Software Engineering paradigm [3]), this approach makes it possible to help software organizations improve their product quality by several orders of magnitude, double their productivity and project success rate, and halve their cost at the same time, and efficiently handling the essential issues in software engineering: the complexity, changeability, conformity, and invisibility.

Source code itself is not the best documents of a software product, but is the best source for holistic and dynamic software modeling - the generation of the best graphics to document a software product.

NSM has been completely implemented, and fully supported by Panorama++ product - the platform for implementing NSE.

"The next century will be the century of complexity" (Stephen Hawking - January 2000). I believe that it is the time for more software companies particularly the software tool vendors to join the nonlinear software engineering revolution driven by complexity science! Yes, there are challenges, but there are also opportunities such as developing new platform-independent programming languages which are more suitable for high-level abstractions of a software product; designing better tools and notations/diagrams which can be used to map more UML modes/diagrams and can be automatically generated from platform-independent programming.
languages or platform-dependent programming languages; providing software tool packages to re-document legacy software products written in Cobol, ADA, or FORTRAN programming languages through nonlinear software modeling, etc..

With NSM, it is true that “The Code is the Design”,[16]

References

[10] Nguyen Viet Cuong, Xhevi Qafmolla, (2011), Model Transformation in Web Engineering and Automated Model Driven Development
[16] Jack W. Reeves, (http://developers.slashdot.org/story/05/03/01/2112257/the-code-is-the-design)