NISTIR 4384

NEW NIST PUBLICATION
December 1990

DESIGN ISSUES FOR CONFORMANCE TESTING OF THE PHIGS STANDARD

John Cugini
Lynne S. Rosenthal

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
National Computer Systems Laboratory
Gaithersburg, MD 20899

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

NIST

NISTIR 4384

DESIGN ISSUES FOR CONFORMANCE TESTING OF THE PHIGS STANDARD

John Cugini
Lynne S. Rosenthal

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards
and Technology
National Computer Systems Laboratory
Gaithersburg, MD 20899

August 1990

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary
NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
John W. Lyons, Director

Design Issues for Conformance Testing of the PHIGS Standard

John V. Cugini
Lynne S. Rosenthal

National Computer Systems Laboratory
National Institute of Standards and Technology

ABSTRACT: Conformance testing for the Programmer's Hierarchical Interactive Graphics System (PHIGS) standard presents certain novel difficulties, especially the indirect effect of many functions, and the inaccessibility to the program of visual effects. The model of logical inference offers a way to organize a system of the complexity needed to overcome these problems. This complexity makes the use of certain database concepts quite valuable in allowing users to comprehend the system. The problem of inaccessible effects can be addressed only by careful design of the user interface, so as to minimize the operational difficulty and subjectivity inherent in testing such features.

KEYWORDS: conformance testing; graphics standards; human factors; PHIGS standard; validation of software

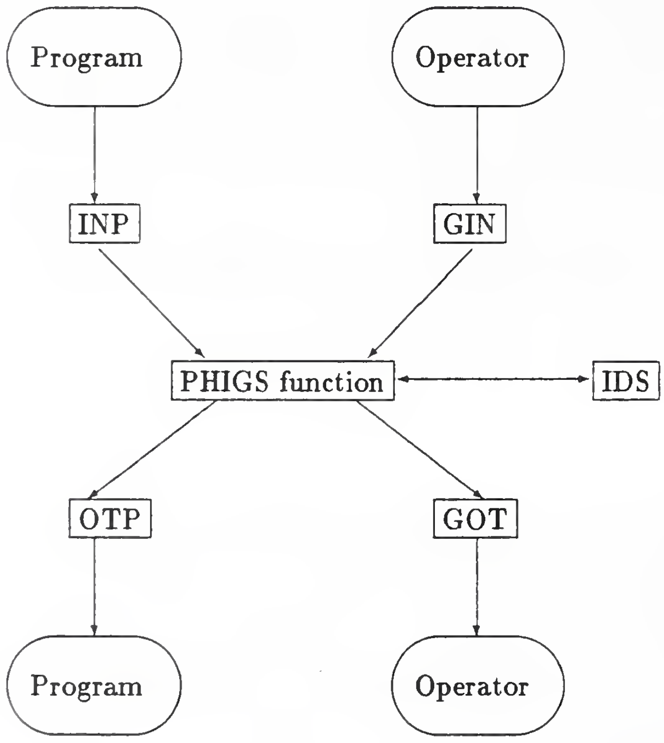
1 Overview

Standards for graphics programming [GKS85, PHIGS88], while similar in some respects to programming language standards, present special challenges for conformance testing. The National Computer Systems Laboratory (NCSL) of the National Institute of Standards and Technology (NIST) has undertaken the development of a PHIGS Validation Test (PVT) system [CUGI90] for the recently adopted standard for PHIGS. In this test suite, we have attempted to address systematically some of the problems posed by the nature of graphics standards.

The PHIGS standard, unlike programming language standards, and to some extent unlike the GKS standard, is built very largely around a state-machine concept. Many functions, such as POLYLINE, which might normally be thought to generate graphic output, do not directly do so. Rather they either set or inquire a complex state environment. The two most significant data structures are those describing the state of the workstation, upon which graphical objects may be drawn, and the *Centralized Structure Store* (CSS) which contains a hierarchical database logically describing graphical objects. It is only when these two are made to interact, by *posting* parts of the CSS to a workstation that a program generates any visible results. These visible results are, moreover, not subject to automatic testing, since PHIGS (unlike GKS) provides no function for reading pixels from the screen. Thus, most of the functions to be tested have effects which are either *indirect*, or *inaccessible* to the program.

2 Testability

Let us examine in detail how the problem of indirectness arises within PHIGS. PHIGS interacts with the external world directly only via the parameters of its functions (visible to the program) and its graphic effects (visible to the operator). Schematically, we can think of the input/output structure of PHIGS functions as depicted in Figure 1.



INP:	Input parameters
OTP:	Output parameters
GIN:	Graphic input
GOT:	Graphic output
IDS:	Internal data structures

Figure 1: Input/output structure of PHIGS functions

No single PHIGS function has all these potential inputs and outputs; the figure describes a framework within which one can categorize the influences on, and effects of, each function. In order for a function to be automatically and directly testable, its only input must be INP and its only output OTP. Only a few PHIGS functions (utilities for calculating transformation matrices) fit this description. At the other extreme, functions for archiving portions of the

CSS to a file (or retrieving them), depend strongly on the IDS and have their main effect on the IDS. In order to test such a function, we must wrap it within a known environment, so that the compound data flow has directly accessible first inputs and last outputs. For example, we might construct the following template for testing the <archive structure> function (n.b., names of PHIGS functions in angle brackets) as shown in Figure 2.

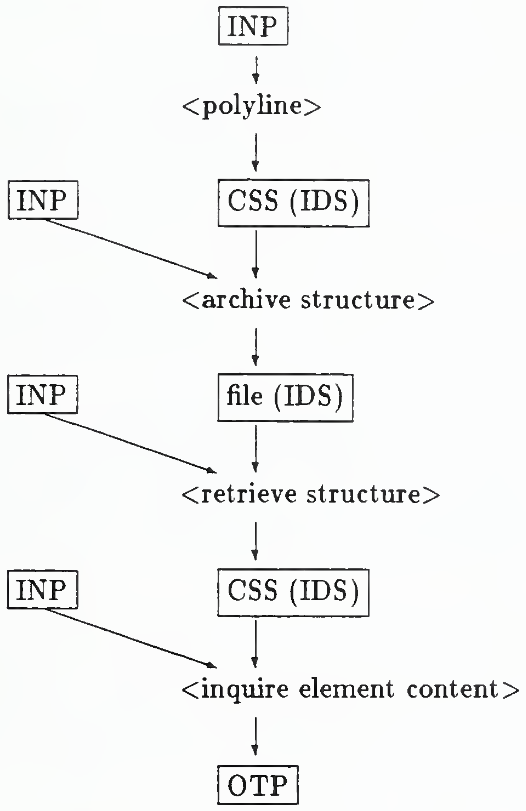


Figure 2: Testable segment for <archive structure>

In general, for a template to be testable, all its inputs must be known and controlled by the program or operator, and its output must be visible, either to the program or operator. The idea is to arrange functions so that all the IDSs are at an intermediate stage of the segment. Of course, the price paid is that in general only *combinations* of functions can be tested.

3 PVT as Logical System

We need some model to guide the construction of testable code segments, rather than approaching this as an ad hoc task. In particular, we must solve the problem of relating such test segments back to the standard which supposedly justifies them.

One helpful way of understanding conformance testing is within the framework of deductive logic. In a logical system, the basic operation is that of inference. From a finite number of given premises, one can infer a set of conclusions. If the premises are true and the inference is valid, then the conclusions must also be true. Conversely, if the inference is valid and the conclusion is false, then one of the premises relied upon must be false. This latter mode of reasoning is the one typically used in conformance testing.

3.1 Application to PHIGS standard

The PHIGS standard can be understood as a set of premises or axioms for a logical system. The premises are of the form: "For all X, if X is a conforming implementation, then X must behave in the following way: ..."; i.e., the standard defines the behavior of a conforming implementation. When we test an alleged implementation, P, we rely on the additional premise: "P is a conforming implementation." From these premises we can then infer a great deal about the behavior of P. If we discover that the actual behavior of P contradicts this theoretical behavior, we can then conclude that the premise that P conforms is false.

Although this outline is neat in theory, in practice there are a number of difficulties. First and foremost, the standard is not cast as a series of logical premises, but as a document composed of English prose. Second, in order to infer a seemingly simple conclusion, one often needs a large number of premises from the standard including several which are not of direct interest to the test at hand. Finally, it is difficult to relate all this logical machinery to the actual code.

Despite the impracticality of carrying out the full realization of the pure logical model, it serves as an ideal which can suggest ways of structuring the actual system. In particular, the PVT uses the notion of semantic requirements (SRs), which play the role of the premises of the standard, and test case (TC) results, which play the role of conclusions.

3.2 Semantic Requirements

The PVT system consists of modules (see below), each of which contains a set of semantic requirements. These SRs represent a partial axiomatization of a given topical area of the standard: the premises for that section of the standard. Like logical premises, well-designed SRs should be:

1. Independent - If one SR implies another we keep only the stronger of the two, since the other is redundant.

2. Complete - The SRs should require everything that the standard requires.

3. Consistent - The SRs should not contradict each other.

4. Specific - Each SR must have some testable consequence, perhaps in conjunction with other SRs. Broad generalities are to be avoided in favor of lower-level concrete assertions.

5. Simple - An SR should not be a long list of requirements; to a reasonable extent, each SR should state an atomic rule about conforming implementations.

Even given these guidelines, there is no uniquely best way of formulating the SRs for a given module. Once a set of SRs has been developed, however, we do have a clear statement of the requirements to which an implementation will be held. This serves to sharpen any interpretation question which may emerge. These cases serve as feedback to the standardization process so that inconsistent or incomplete specifications in the standard may be corrected.

3.3 Test Cases

From a single SR, it is usually impossible to infer any conclusion about the behavior of an implementation. Thus, it is generally not the case that an SR will be testable in isolation. The typical situation, rather, is that from a set of SRs some conclusion can be drawn which is directly testable. Such a conclusion is the basis for a test case (TC) within the module. Each module contains, not only a set of SRs, but a set of programs each of which contains in turn a set of TCs. These TCs are the executable core of the PVT system. Each TC consists of an explicit conclusion about the behavior of a conforming implementation, worded so as to state what "should" happen.

The important thing is to relate these TCs back to the SRs. Each TC is associated with a set of SRs in order to suggest how a proof that all conforming implementations succeed in the TC might be constructed. There is, in general, a many to many relationship between SRs and TCs: one TC will typically depend on several SRs, and each SR may be used by several TCs. Passing the TCs is no guarantee that an implementation really does conform; it might well violate the standard in untested areas. But failure in a valid TC does strictly imply failure to conform.

4 PVT as a database

We have seen that there are several types of entities involved in a conformance test system: PHIGS functions, PHIGS data structures, programs, semantic requirements, test cases, and the PHIGS standard itself. Users of the test system need to navigate around this set of objects. In particular, conformance tests must be informative about the relationship of the test to the standard, otherwise the advantages of the logical model are lost. We took it as a central goal not merely that the tests be logically correct, but also that interested users be given a way to see for themselves that the expected outcome is correctly grounded in the specifications of the standard.

The database paradigm provides well-understood techniques for addressing these issues. The "navigation" problem can be thought of as a database design problem. We developed a database schema to express the relationships among the objects mentioned. By adopting strict conventions for the documentation, we not only enhance human readability, but also allow automatic capture of the "data" to be used in comprehending the PVT system.

The SRs anchor the system; they specify the precise behavior of PHIGS functions and data. Therefore, the SRs also serve as the reference points for related entities. Specifically, each SR is annotated with a list of related functions, data structures, and references to the standard. By adopting a canonical numbering of the functions and data structures and documenting the references according to a well-defined format (see examples in next section), we allow cross-reference tables to be built automatically once the original annotation is done.

Besides enabling users to navigate within the system, an equally important goal of this approach is that it gives us a good coverage metric: we can see which functions and data structures have been probed by the total PVT system.

Figure 3 contains a diagram exhibiting the main features of this database, namely its entities and the relationships among them. Note the central role of the SRs.

5 PVT Architecture

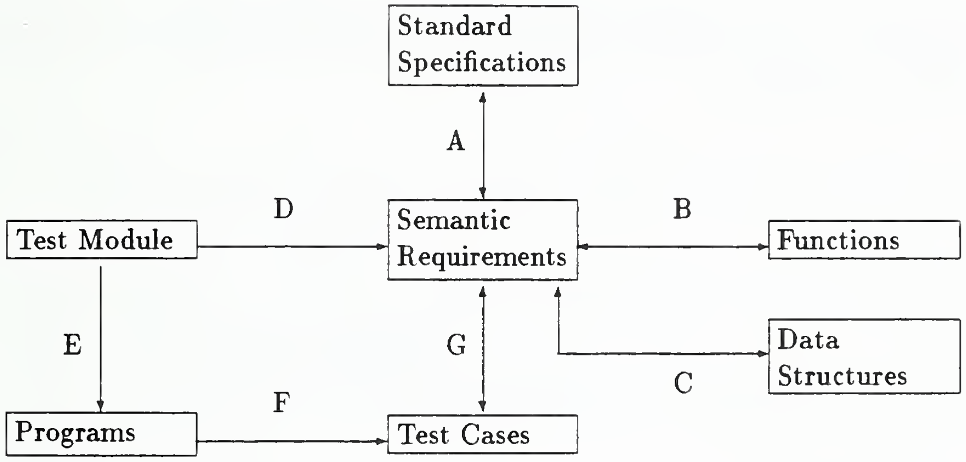
Given the design goals and models set forth, the question becomes how to realize these in an actual system of software and documentation. The representation of the abstract entities should be such as to be easily accessible by both automatic processes and the human user.

5.1 Modularity

The set of all SRs for the entire PHIGS standard is far too large to be handled as a single entity. Rather, they are divided into topically coherent subsets, which we call modules. Ideally there would be no interaction between modules and strong logical interdependence within each module.

Recalling the many-to-many relationship between SRs and TCs, we strive to keep each TC dependent mainly on the SRs of its own module. Thus, the rationale for grouping a set of SRs and TCs into one module is that their functions and data structures must be tested together, i.e., their behavior is strongly interrelated. Hence, the module's set of SRs and TCs form a "natural cluster."

Although the SRs of a module are closely related, the TCs may be numerous or diverse enough to justify further subdivision into several programs. We tried to strike a reasonable balance, grouping closely related TCs within single programs, yet keeping the programs fairly small and comprehensible. We thereby avoid unwieldy programs at one extreme, and the high overhead of one test case per program at the other.



An arrowhead next to an entity indicates that there may be many instances of the entity. For example, the D relationship between Test Module and Semantic Requirements is one-to-many; the B relationship between Semantic Requirements and Functions is many-to-many.

Relationships:			Captured by:
A. SR	is derived from	Standard Specs	#S in SR
B. SR	depends on	Function	#F in SR
C. SR	depends on	Data Structure	#D in SR
D. Module	tests	SR	SR in DOC.TXT
E. Module	contains	Program	design in DOC.TXT
F. Program	contains	TC	"TEST:" in design
G. TC	directly tests	SR	SR list in TCs

Figure 3: The PVT as a database

Thus, the PVT suite is organized as a set of logically independent modules, each of which contains a set of SRs and a set of TCs, with the TCs distributed among several programs. Concretely, each module will consist of a) a single documentation file, conventionally named DOC.TXT, containing the SRs and the design of the programs, and b) several source code files, one per program.

```
SR4.  <Inquire predefined polyline representation> returns the
linetype, linewidth scale factor, and polyline colour index for a
predefined polyline bundle.
#F 263
#D 6.10.2
#S n
#T P02/1 P02/5 P03/2

...

SR6.  No two of the first 5 predefined polyline bundles have
identical sets of attributes.
#F 263
#D 6.10.2
#S 4.5.2/43/3 4.14/113/3 4.14/114/1 6.7/322/1
#T P02/5
```

Figure 4: Example of Semantic Requirements

Figure 4 contains an excerpt from the semantic requirements for the module on the polyline bundle table. The lines beginning "#F" refer to related PHIGS functions, which are numbered according to their order in the standard. #D lists related data structures, #S the relevant section, page, and paragraph number in the standard, and #T the related test cases. The #T entries are generated automatically from the program design, as illustrated below in Figure 5. The module documentation contains a dictionary for the numbered entities (#F and #D), so that casual readers can decode the numbers. The system-level documentation contains three cross-reference tables indexed by function, data structure, and section of the standard. These tables allow users to identify the parts of the test system that deal with the entity of interest.

Figure 5 contains an excerpt from the design of the program which tests the requirements. This is the fifth test case in that program, hence the reference to "P02/5" under both SR4 and SR6 above in Figure 4. Note that the "TEST:" line refers to the semantic requirements under test. Executing "pass" or "fail" causes the system to generate a message consisting of the test case statement ("The first five ...") and an indicator that the test was passed or failed. These messages can be routed to any combination of three destinations: the screen,

```
PROGRAM 2: Validity of predefined polyline bundles
...

TEST: #SR 4 6
      "The first five predefined polyline bundle entries should all be
       mutually distinguishable."
do pli = 1 to 5
  <inquire predefined polyline representation> for index pli, determine
    pdlt = predefined linetype
    pdlw = predefined linewidth scale factor
    pdci = predefined polyline color index
  do pli2 = 1 to pli-1
     if svpdlt (pli2) = pdlt  and
        svpdlw (pli2) = pdlw  and
        svpdci (pli2) = pdci    then
          fail
          message about pli, pli2
          goto done
     endif
  loop
  svpdlt (pli) = pdlt
  svpdlw (pli) = pdlw
  svpdci (pli) = pdci
loop
pass

done:
```

Figure 5: Program Design Excerpt with Test Case

an individual message file for this program only, or a cumulative message file for the whole system.

5.2 Tree Organization

The standard is not just an undifferentiated set of requirements, but has clearly delineated functional areas. We considered using the sequence of function definitions in section 5 of the standard or the data structures of section 6 as an organizing principle, but it seemed finally that the more conceptual organization of section 4 provided the best model, especially given the goal that each module deal with all the strongly related requirements of a topic. For instance, a set and inquire function often must be tested together and obviously belong in the same module, even though in section 5 all the inquires are treated as a separate group.

The standard is organized in a hierarchical structure. Moreover, most computer systems provide a hierarchy for their file systems. It therefore seemed reasonable to take advantage of this built-in order and organize the modules into a topical hierarchy, or tree, to be realized using the tree structure of the file directory system.

6 Real Effects Testing

Version 1.0 of the PVT system incorporates the design concepts discussed above, but was deliberately limited to the state-machine semantics of PHIGS. The next version must address the issues surrounding the so-called "real effects" of PHIGS programming, those associated with the "GIN" and "GOT" interfaces of PHIGS functions described above under Testability.

PHIGS semantics are described in English. These descriptions appeal freely to notions such as "dotted line," "yellow," "invisible," and so on - the common-sense vocabulary of human visual perception. These terms are taken as primitive; for instance, there is no attempt in the standard to define "dotted line". Barring exotic technology such as robotic vision, this means that a test system for PHIGS must rely on human operators. A good operator interface then becomes essential to the tests, not just an incidental feature. Such an interface should minimize both the difficulty of operating the tests, and the subjectivity involved in verifying graphical input and output.

The only system which has confronted these issues is the conformance test suite for GKS [GKST89]. The GKS tests typically present the operator with a non-trivial visual display. The operator must have at hand the (paper) documentation for the test, which asks several yes/no questions regarding the display. E.g., for linetype, the display includes a depiction of the sun, with rays of various linetypes emanating from it. The operator script includes the questions:

> Do four rays and clouds appear on the display surface, with annotation (in anti-clockwise order starting from the horizontal) of 'SOLID', 'DASHED', 'DOTTED',

and 'DASHED-DOTTED'?

and

Is each ray drawn in a linetype that is recognisable as 'SOLID', 'DASHED', 'DOTTED', and 'DASHED-DOTTED', in the order described above?

The operator then writes his/her answers on the sheet, perhaps with comments, and this constitutes the result of the test.

In version 2.0 of the PVT, we plan a somewhat different approach. First, as far as possible, operator responses to the tests will be performed via the computer, rather than paper and pencil, if for no other reason than to allow machine-readable recording of the results. Second, we will emphasize simpler screen displays and questions. It seems reasonable to suppose that this will lessen the potential for confusion, and help to isolate non-conforming behavior. Third, we hope to test the adequacy of the visual display more directly. For instance, instead of asking the operator whether certain linetypes are "recognisable," we can test recognizability by requiring the operator to match linetypes with their official verbal description. The screen could display lines of each of the four types (with perhaps a fifth non-standard type as well), and then the operator would be prompted to point to the dotted one, the dashed one, etc. His/her ability to do so correctly is proof enough of their recognizability. Conversely, if the operator is unable to distinguish the dotted line from the others, it is a fair inference that the implementation does not adequately meet the specifications of the standard.

The PHIGS features for geometric transformations (which provide 3-D perspective views of objects) are an important part of the standard, yet difficult to verify visually, because of the fine judgment involved. We believe these features are susceptible to test by simulation. After the 2-D capabilities of PHIGS have been verified, the test system could include a simulator for the 3-D features built on the trusted 2-D operations. This simulator need not be fast, just logically correct. It would then be possible to generate an expected display, perhaps overlapping and slightly offset from the actual display generated directly by PHIGS. The operator then need only look for non-parallel lines, which we assume would be quite conspicuous.

7 Summary

Conformance testing becomes more difficult as software standards address more complex computational applications. Nonetheless, by drawing on a variety of techniques which have originated in non-graphical areas of computer science, the testing process can be made reliable, comprehensible, and predominantly automated.

References

[CUGI90] John Cugini, Mary T. Gunn, Lynne S. Rosenthal, *User's Guide for the PHIGS Validation Tests (Version 1.0)*, NISTIR 90-4349, National Institute of Standards and Technology, Gaithersburg, MD, 1990.

[GKS85] *Computer Graphics - Graphical Kernel System (GKS) Functional Description*, ANSI X3.124-1985, American National Standards Institute, New York NY, 1985.

[GKST89] *GKS Validation Test Suite*, Version 2.1, National Institute of Standards and Technology, Gaithersburg, MD.

[PHIGS88] *Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS) Functional Description, Archive File Format, Clear-Text Encoding of Archive File*, ANSI X3.144-1988, American National Standards Institute, New York NY, 1988.

NIST-114A
(REV. 3-90)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NUMBER
NISTIR 3484

2. PERFORMING ORGANIZATION REPORT NUMBER

3. PUBLICATION DATE
AUGUST 1990

4. TITLE AND SUBTITLE
DESIGN ISSSUES FOR CONFORMANCE TESTING OF THE PHIGS STANDARD

5. AUTHOR(S)
John Cugini, Lynne Rosenthal

6. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)
U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
GAITHERSBURG, MD 20899

7. CONTRACT/GRANT NUMBER

8. TYPE OF REPORT AND PERIOD COVERED

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

10. SUPPLEMENTARY NOTES

11. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

Conformance testing for the PHIGS graphics standards presents certain novel difficulties, especially the indirect effect of many functions, and the inaccessibility to the program of other effects. The model of logical inference offers a way to organize a system of the complexity needed to overcome these problems. This complexity makes the use of certain database concepts quite valuable in allowing users to comprehend the system. The problem of inaccessible effects can be addressed only by careful design of the user interface, so as to minimize the operational difficulty and subjectivity inherent in testing such features.

12. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)
Conformance testing; graphics standards; human factors; PHIGS; testing of software; vlaidation of software

13. AVAILABILITY

X	UNLIMITED
	FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).
	ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.
X	ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.

14. NUMBER OF PRINTED PAGES
15

15. PRICE
A02

ELECTRONIC FORM