INTERNATIONAL STANDARD

**ISO/IEC 13673**

First edition
2000-05-01

# Information technology — Document processing and related communication — Conformance testing for Standard Generalized Markup Language (SGML) systems

*Technologies de l'information — Traitement documentaire et communication connexe — Tests de conformité pour langage normalisé de balisage généralisé (SGML)*

Reference number
ISO/IEC 13673:2000(E)

## Contents

Page

## Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this International Standard may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

International Standard ISO/IEC 13673 was prepared by ANSI (as ANSI X3.190) and was adopted, under a special "fast-track procedure", by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by national bodies of ISO and IEC.

Annex A forms a normative part of this International Standard. Annex B is for information only.

## Introduction

ISO 8879:1986 and 8879:1986/A1:1988, *Information processing – Text and office systems – Standard Generalized Markup Language (SGML)*, define when a system is a conforming SGML system. The determination of whether a system is a conforming SGML system is of value both to potential users of such systems and to their developers. This determination is, however, a complex process. To this end, efforts are underway to develop test suites to validate conformance. Standardization of development and use of test suites assures consistency of results and informs the public of the implications of the tests. Such formalism is provided by this standard, which includes

- guidelines for the content of individual tests;
- rigorous conventions for naming test cases and the constructs used within them;
- formatting and comment conventions;
- conventions for classifying test cases;
- conventions for documenting test suites;
- definition of a Reference Application for SGML Testing (RAST) that indicates how an SGML parser interprets a test;
- definition of a Reference Application for Capacity Testing (RACT) that reports a parser's capacity calculations;
- conventions for reporting a system's performance on a test suite.

This standard also addresses conformance to the related standard, ISO 9069:1988, *Information Processing – SGML support facilities – SGML Document Interchange Format (SDIF)*, as SDIF is needed to connect the several entities of an SGML document into a single object for interchange within OSI.

This standard may be used by those who develop SGML test suites, those who build SGML systems to be evaluated by such suites, and those who examine an SGML system's performance on a test suite as part of the process of selecting an SGML tool.

# Information technology — Document processing and related communication — Conformance testing for Standard Generalized Markup Language (SGML) systems

## 1 Scope

This standard addresses the construction and use of test suites for verifying conformance of SGML systems. Its provisions assist those who build test suites, those who build SGML systems to be evaluated by such suites, and those who examine an SGML system's performance on a test suite as part of the process of selecting an SGML tool.

In particular, this standard includes:

– criteria for the organization of test suites, including naming conventions, documentation conventions, and specification of applicable concrete syntaxes and features. Among other advantages, these conventions facilitate any non-SGML automatic processing that may be convenient for the developers or the users of the tests;

NOTE – An example of such non-SGML processing is sorting tests by name.

– a standard form for describing test results that makes clear what has been proven or disproven by the tests;

– the specification of a Reference Application for SGML Testing (RAST) that interprets all markup to allow machine comparison of test results for documents conforming to ISO 8879. RAST indicates in a standard way when tags, processing instructions, and data are recognized by the parser, replacing references and processing markup declarations and marked sections appropriately. RAST tests information likely to be passed by a general-purpose SGML parser to an application but does not test additional information that some parsers provide;

– the specification of a Reference Application for Capacity Testing (RACT) that reports a validating parser's capacity calculations. An SGML system that supports this application indicates its ability to report capacity errors regardless of whether it supports variant capacity sets;

– the specification of test procedures related to SDIF data streams.

This standard applies to the testing only of aspects of SGML implementation and usage for which objective conformance criteria are defined in ISO 8879.

NOTE – Among the aspects of an SGML system not addressed by this standard are error recovery, phrasing of error messages, application results, and documentation (including the system declaration).

## 2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this International Standard. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

ISO 646:1983, *Information processing – ISO 7-bit coded character set for information interchange*

ISO 8879:1986, *Information processing – Text and office systems – Standard Generalized Markup Language (SGML)*

ISO 8879:1986/A1:1988, *Information processing – Text and office systems – Standard Generalized Markup Language (SGML) Amendment 1*

ISO 9069:1988, *Information processing – SGML support facilities – SGML Document Interchange Format (SDIF)*

## 3 Precedence of ISO 8879

Any discrepancy between any provision of this standard and ISO 8879 should be resolved in accordance with the latter. Furthermore, should any future effective edition of ISO 8879 contradict any provision of this standard, a test suite for the future version will be considered to conform to this standard only if the discrepancy is resolved in accordance with the effective edition of ISO 8879. In particular, the precedence of ISO 8879 applies to the definitions in clause 4, the description of RAST in clause 14, and the description of ESIS in annex A.

Should there be any internal inconsistencies within this standard between annex A and the remainder, implementors of conforming test suites shall rely on the provisions in annex A.

## 4 Definitions

NOTE – None of the terms defined below are used or defined in ISO 8879. Should such definitions be added to some future version of ISO 8879, the precedence of ISO 8879 will apply in accordance with clause 3.

**4.1 anomalous test case:** A test case that deviates from some requirement of ordinary tests because the tested SGML construct is incompatible with that requirement.

**4.2 application modules:** Components of an SGML system other than the parser and entity manager.

**4.3 effective edition:** The current edition of a standard including any amendments, addenda, or other modifications.

**4.4 Element Structure Information Set:** Information comprising the element structure that is described by SGML markup (the element structure information set is defined in annex A).

**4.5 ESIS:** Element Structure Information Set.

**4.6 equivalent SGML documents:** SGML documents that, when parsed with respect to identical DTDs and LPDs, have an identical ESIS.

**4.7 internal entity:** An entity whose replacement text appears in an entity declaration.

**4.8 lexicographic order:** An order in which distinct strings are arranged by comparing successive letters. One string appears before another if it is a prefix of the second, or if, according to the following conventions, in the first position where they differ, the character in the first string precedes the character in the second string. Printable characters precede nonprintable characters. One printable character precedes another if the ISO 646 character number of the first is smaller than the ISO 646 character number of the second. In particular, the space character precedes all other printable characters and any other printable character precedes a second one if the first precedes the second character in the list of printable characters given in 4.14. A nonprintable character precedes another if its character number in the document character set is smaller than the character number of the second in the document character set.

NOTE – For strings consisting only of printable characters, this order is independent of concrete syntax.

**4.9 major SOO:** A statement of objective for several related tests in a test suite.

**4.10 markup-sensitive SGML application:** An SGML application that can act on SGML markup as well as element structure.

**4.11 minor SOO:** A statement of objective that describes the particular principle of the SGML language that distinguishes an individual member of a group of related tests.

**4.12 nonprintable character:** A character that is not a printable character (see 4.14).

**4.13 ordinary test case:** A test case that follows the naming, organizing, and formatting conventions itemized in this standard and identified as requirements for ordinary tests (see *anomalous test case*).

**4.14 printable character:** A character with ISO 646 character number in the range 32 to 126 inclusive. These characters consist of the space character and all the following:

```
! " # $ % & ' ( ) * + , - . / 0 1 2
3 4 5 6 7 8 9 : ; < = > ? @ A B C D
E F G H I J K L M N O P Q R S T U V
W X Y Z [ \ ] ^ _ ` a b c d e f g h
i j k l m n o p q r s t u v w x y z
{ | } ~
```

**4.15 RACT:** Reference Application for Capacity Testing.

**4.16 RAST:** Reference Application for SGML Testing.

**4.17 Reference Application for Capacity Testing:** An SGML application that reports capacity calculations (defined in clause 15).

**4.18 Reference Application for SGML Testing:** An SGML application that reports ESIS information (defined in clause 14).

**4.19 SOO:** Statement of objective.

**4.20 statement of objective:** A brief description of the aspect of the SGML language explored in an individual test case or a group of related tests.

**4.21 structure-controlled SGML application:**

An SGML application that operates only on ESIS information and the “`APPINFO`” parameter of the SGML declaration; a structure-controlled application operates on the element structure described by SGML markup, never on the markup itself.

**4.22 test case (or test):** An SGML document included in a test suite.

**4.23 tested system:** An SGML system that is evaluated by inspection of the results it produces on the test cases of a test suite.

**4.24 test suite:** A documented collection of SGML documents intended to exercise an SGML system in order to indicate whether the system conforms to the specifications of ISO 8879.

## 5 Use of SGML test suites

Because of the wide variation possible in SGML systems, no single test suite is adequate for testing how well all SGML systems conform to the requirements of ISO 8879. Some SGML systems produce SGML documents, others process SGML documents to obtain various results, still others both read and produce SGML documents. Some systems are restricted to documents with particular document type declarations, others can process arbitrary documents meeting the constraints of the *system declaration*. A test suite intended for a more general system contains test cases that cannot be processed by a more restrictive system; a test suite for a restrictive system does not adequately explore the capabilities of a more general one.

NOTE – An SGML test suite indicates whether the modules of an SGML system that process SGML do so according to the specifications of ISO 8879. Testing a system's SGML capabilities does not indicate whether it correctly performs a desired application in other respects.

### 5.1 Comprehensive test suites

SGML test suites shall be comprehensive. A general-purpose SGML test suite shall provide tests that explore conformance to every required aspect of the SGML language and to every aspect of supported optional features. Similarly, a test suite for a particular application shall provide tests to explore every aspect of the SGML language used in that application.

NOTE – An application-specific test suite may not be able to test all required constructs of SGML and cannot indicate whether the underlying SGML parser conforms to the requirements of ISO 8879 for such constructs. For example, attributes cannot be tested if an application does not happen to use any. Thus, a test suite for such an application cannot predict conformance of attribute handling in an implementation of another application built with the same parser.

This standard defines requirements for testing general SGML systems. Test suites intended for more restrictive environments may deviate from these requirements only where the requirements are incompatible with the system to be tested. For example, the conventions for selecting generic identifiers cannot be followed in a system restricted to a document type declaration that uses other conventions.

A test suite for a validating SGML system shall include erroneous test cases to investigate comprehensively the system's ability to detect errors. A nonvalidating SGML system can be tested with such a test suite, but its results on erroneous documents are not predictable.

## 5.2 The role of SGML in a tested system

The way a test suite is used depends on whether the tested system processes existing SGML documents, or produces SGML documents.

### 5.2.1 Systems that read SGML

A system that acts upon existing SGML documents is tested by examining the results it produces from every test in a comprehensive test suite. However, the variation in SGML systems means these results may take any number of forms. As a result, there is no unique method for determining whether a tested system correctly processes a test case.

The remainder of this subsubclause discusses various methods for evaluating test suite results produced by a system that processes SGML documents. Of these methods, RAST provides the most information and should be used whenever possible.

#### 5.2.1.1 Evaluating with RAST

RAST (see clause 14) is a simple SGML application designed to validate a parser's recognition of the Element Structure Information Set (ESIS). ESIS (see annex A) is the information exchanged by a parser and other components of a program that implements a structure-controlled application. RAST reflects the ESIS of an SGML document with a minimal amount of additional information in such a way that the results it produces from two SGML documents using the same concrete syntax will be the same if and only if the two documents have the same ESIS. An SGML system that supports RAST is easily tested by machine comparison of RAST results to known correct RAST output for every document in a test suite.

NOTE – There is no requirement that an SGML system support RAST. However, it should be easy to implement RAST with any general-purpose SGML system that provides a software-development environment for building SGML applications.

#### 5.2.1.2 Comparing with equivalent documents

An SGML system that does not support RAST can be tested to some extent through a structure-controlled application with the following properties:

- The application is not restricted to one or more specified document type definitions;
- The application's output is machine-readable (for example, it is a computer file rather than printed paper or sound). Such applications include, for example, one that counts the number of elements in a document or one that produces a vocabulary list of the unique words that occur within the *content* of a document.

The test procedure involves comparing the application's output on sets of equivalent, but not identical, SGML documents. Identical output must be produced for such documents. This criterion alone cannot demonstrate a system's conformance to ISO 8879. For example, the criterion is satisfied by a system that produces identical output for all documents, equivalent or not. More information is obtained if the application produces different results for documents that are not equivalent. Note, however, that the simple word-list application just described does not meet this stricter constraint, since there could be documents with very different element structure that use the same vocabulary.

NOTE – Implementors of test suites that consist of sets of equivalent documents should verify that members of each set are indeed equivalent by confirming that RAST produces the same output for every member in the set.

#### 5.2.1.3 Evaluation through error recognition

The correctness of a validating SGML parser can, in large measure, be demonstrated if the parser a) reports erroneous SGML documents to be invalid and b) reports valid documents to be conforming. This type of testing can be done regardless of how errors are reported (possibilities include visual and auditory signals as well as error messages). However, some aspects of SGML parsing – for instance, significance of record ends and correct interpretation of default attribute values – do not affect whether the document is valid and hence cannot be tested in this way. Comprehensive testing of markup minimization in this manner is also difficult. Furthermore, a system that reports an erroneous document to be in error need not be conforming; the system may have accepted the erroneous construct and misinterpreted some correct markup.

#### 5.2.1.4 Other forms of evaluation

Knowledge of particular applications can be used to design system-specific methods of reporting all or part of the ESIS information in a document. The reported information is an indication of the conformance of the tested system's parser to ISO 8879.

### 5.2.2 Systems that generate SGML

A system that produces SGML documents is tested by processing representative output with a system that reads SGML documents. A test suite therefore consists of test cases that produce a comprehensive collection of output documents.

NOTE – This procedure shows whether the tested system produces conforming SGML documents from the test cases; it provides no information about whether the output is correct in other respects.

### 5.2.3 Systems that both read and produce SGML

A system that both processes and generates SGML documents can be tested separately as a system that reads SGML and as one that produces SGML. Depending on the relationship between the input and output documents, a comparison of the two may provide additional results. Although such a comparison is application dependent, it may reveal information about SGML conformance. One form of comparison is testing whether input and output are equivalent SGML documents (which can be done by a character-by-character comparison of their RAST results). This comparison is useful, for example, in testing a text editor that can both import and export SGML documents. Such an editor's SGML parsing can be tested by importing each test in a test suite and immediately exporting the unedited document; the result should be an equivalent document. Similarly, a tool that replaces a minimal SGML document with an equivalent one using various forms of markup minimization should produce output equivalent to its input. For some applications, it may be useful to verify that input and output are identical. Other forms of comparison depend on particular applications.

### 5.2.4 Systems that use SGML as an intermediate form

A system may use SGML even if both its original input and final output have some other form. Such a system creates an SGML document and then processes it to obtain another result. Depending on the implementation, it may be possible to test the embedded SGML parser in another application. Furthermore, if the intermediate SGML document can be saved, the system can be evaluated as a system that produces SGML. In other cases, system-specific testing is required.

## 6 Test suite documentation

This clause describes information that shall be included in the documentation that accompanies a test suite. This information shall be available to all potential and actual users of the test suite and shall be repeated in any report generated after a system is tested.

### 6.1 General documentation

The documentation shall include the following:

– one or more identifiers, such as ISO 8879:1986(E) or ISO 8879:1986/A1:1988(E), indicating the effective edition of ISO 8879 used in preparing the test suite;

– one or more identifiers, such as ISO/IEC 13673:2000(E), indicating the effective edition of this standard used in preparing the test suite and in any implementations of RAST and RACT used to generate results of those applications provided with the test suite;

– the following statement, translated if the document is not in English:

A test suite can indicate that an SGML system is nonconforming by providing a test on which the system fails. However, no test suite can prove that an SGML system is fully conforming or predict the results the system would obtain on untested documents.

– the following statement, translated if the document is not in English:

When a tested system produces results other than those expected by a test suite, the discrepancy may result from an error in either the test suite or the tested SGML system.

– a description of the types of SGML system that can be tested by the test suite. This description, for example, indicates whether the test suite is restricted to a particular application. It also identifies any provisions of this standard that could not be observed – naming conventions that are incompatible with an application's document type declaration, for instance;

– indication of whether the test suite explores validation as well as conformance of SGML documents; in other words, whether some test cases are deliberately erroneous documents;

– description of the document character sets used in the test suite in the syntax of the *docu-*

*ment character set* parameter of the SGML declaration, with descriptive comments, if desired;

– a list of all optional SGML features addressed by the test suite in the syntax of the *feature use* parameter of the *system declaration*, with descriptive comments, if desired;

– a list of all optional SGML features not covered by the test suite, with a statement that the results do not predict the tested system's performance on documents using these features. The list is presented in the syntax of the *feature use* parameter of the *system declaration*, with descriptive comments, if desired;

– a description of the concrete syntaxes included in the test suite in the syntax of the *concrete syntax scope* and *concrete syntaxes supported* parameters of the *system declaration*, with descriptive comments, if desired;

– a description of the capacity sets included in the test suite in the syntax of the *capacity set* parameter of the *system declaration*, with descriptive comments, if desired;

– an indication of whether some test cases include explicit SGML declarations or all test cases have implied SGML declarations;

– indication of whether the test suite is accompanied by RAST results for individual tests and, if so:

  – the *system declaration* of the implementation of RAST used to create the results. Descriptive comments may be added. The *system declaration* shall not indicate that an optional feature is supported unless the implementation is able to interpret all processing instructions that direct RAST's processing of that feature (see 14.6.13, 14.6.14, and 14.6.15).

NOTE – Ideally, the implementation of RAST should support all character sets, variant concrete syntaxes, optional features, and variant capacity sets addressed in the test suite. Since such an implementation may not be available when the test suite is constructed, however, it is important that any discrepancies be fully described.

  If the test suite provides test cases for optional features not supported by the implementation of RAST, RAST results shall not be provided for those particular tests;

  – indication of whether the particular implementation of RAST that generated the results is capable of producing the error indication, `#ERROR` (see 14.6.2);

– indication of whether the test suite provides RACT results for individual tests;

– the number of test cases in each category listed in clause 13, as well as identification of any new categories defined for this test suite, with the number of test cases in each.

## 6.2 Test case documentation

The documentation shall also include a statement of objective (SOO) for each test. The SOO describes the primary aspect of the SGML language described in the test case. SOOs are clear and concise statements, which may be direct quotations from ISO 8879, possibly from syntax productions, notes, indented examples, or annexes. A test's SOO appears as a comment within the test case. Furthermore, the SOOs for all tests in the test suite shall be listed in a separate report. The SOO report allows an individual to review the scope and some of the accuracy of the test suite without inspecting the test cases themselves. The document shall include the name of the test case corresponding to each SOO.

When a test suite includes variations of one principle, readability of the SOO documentation can be increased by extracting the common principle into a major SOO and the variations into minor SOOs. The SOO comment in the test case then is the concatenation of the associated major and minor SOOs. An example of a major SOO is “A *prolog* can begin with *other prolog*.” Associated minor SOOs might be:

– An *other prolog* can be a *comment declaration*;

– An *other prolog* can be an *s* separator;

– An *other prolog* can be a *processing instruction*.

## 6.3 Naming SOOs

Each SOO, including major and minor SOOs, shall be given an eight-character name. Letters in SOO names are always lowercase. Each SOO name shall consist of a three-character unique identifier followed by a five-character clause identifier.

The first character in the unique identifier is a letter. If the letter is ‘g’, ‘p’, or ‘i’, than the test shall be a conforming or erroneous document according to the following table:

| First letter | Identifies a test of a |
|---|---|
| g | conforming (or “good”) document |
| p | erroneous prolog |
| i | erroneous document instance |

Any other first letter may be used, but this standard does not assign meaning to other letters.

NOTE – For example, implementors of large test suites might define additional conventions when there are more SOOs in one of the categories in the above table than there are three-character combinations beginning with a particular letter. Implementors might also use different initial letters to avoid duplicating the unique identifiers of an earlier test suite.

The second and third characters of the unique identifiers may be letters or digits, and no significance is attached to the choice of characters.

The clause identifier is a five-character code indicating the clause in ISO 8879 defining the primary aspect of SGML to be tested. Each character is a letter or digit corresponding to a numeric value. Digits represent themselves; the letter ‘a’ corresponds to the number 10; ‘b’ corresponds to 11, etc. The letter ‘z’ is used for all numbers over 34. The first character identifies the clause, the second character the subclause, the third the subsubclause, the fourth the subsubsubclause, and the final digit the paragraph.

Clause headings are not counted for the purpose of this numbering. All other text blocks whose semantics require they be formatted starting at the beginning of a line are considered to be paragraphs for this purpose. For example, each syntax production, note or paragraph within a note, indented example, list item, and list heading is counted as a separate paragraph.

For tests relevant to higher-order subdivisions in ISO 8879, zeros are used for the lower-order clause number. For example, a test of a document with an erroneous prolog based on the second paragraph of Clause 10.4.2 would be given a name of the form `pxxa4202` where “`xx`”' represents two arbitrary letters or digits.

The paragraph number may be left as 0, if the SOO is not associated with a particular paragraph.

As mentioned in clause 6, a test suite identifies the effective edition of ISO 8879 on which it is based. This information is needed to interpret clause identifiers.

When a test involves a construct defined in a figure, the first three characters in the clause identifier are `fig`. (It is not expected that any future version of ISO 8879 will add a subsubclause numbered 15.18.16, so these clause identifiers are effectively unique.) The fourth character is the figure count (using the 1–9, a–z numbering scheme just described). The last digit identifies the row in the figure, if relevant, and is otherwise `0`.

NOTE – The assignment of clauses to SOOs is subjective. For example, individuals may disagree whether a test primarily investigates a system's handling of an `ATTLIST` declaration or of an attribute value.

### 6.4 Revising SOOs

As a test suite is revised over time, SOO names shall remain stable. If a SOO is deleted, its name may not be assigned to a new SOO. The text of the SOO may be corrected, however. A single SOO may be converted into a major SOO with several variations, a major SOO may become a minor SOO, and a minor SOO may become a major SOO or a single SOO. Furthermore, the clause identifier may be corrected. For example, the SOO author may initially associate a SOO relating to attribute values with the clause defining a relevant declaration; in a revision, he may consider it more accurate to identify the SOO with the clause that deals with the specification of attribute values.

## 7 Types of tests

SOOs, and corresponding tests, fall into two main (possibly overlapping) groups:

– Normative, those that test a system's adherence to the SGML standard;

– Volume, those that test the quality of an implementation.

The normative category can be further divided into SOOs and tests that

– relate to a single construct of SGML;

– relate to a single combination of SGML constructs.

SOOs for normative tests are often quotations, or paraphrases of quotations, from ISO 8879.

The volume category can be further divided into SOOs and tests that

– exercise a tested system with variations of normative tests;

– stretch a tested system's capabilities (e.g., exploring memory limits, maximum integer size on a computer system, etc.).

Within these groups, ordinary tests are those that conform to the naming, organization, and formatting requirements of this standard. Since these requirements are compatible with, but more restrictive than, those of ISO 8879, it is conceivable that an erroneous SGML system might correctly process all ordinary tests but be unable to handle other SGML documents. Therefore, to conform to this standard, a test suite shall include at least one anomalous test that deviates from each requirement for ordinary tests. An anomalous test shall conform to all requirements for ordinary tests except those, identified in its SOO, that it intentionally violates.

## 8 General requirements for individual tests

All tests in a test suite shall meet the following requirements:

– Tests are classified by the primary part of the SGML language addressed. However, a complete SGML document contains multiple constructs (e.g., both a *prolog* and a *document instance*). Tests can be grouped into overlapping categories according to the constructs they test. A standard definition of categories is provided in clause 13. Comments in every test identify all relevant categories;

– Tests are commented to identify closely related tests in the test suite. For example, suppose one test verifies that a name of maximum length is accepted and another verifies that it is an error if a name contains too many characters. Comments within each test should mention the other;

– Each test is identified as an erroneous SGML document or as a conforming document;

– Some tests are designed to verify that a system is not making a particular mistake. Such tests are written so that a system that makes the mistake is likely to interpret a conforming document as nonconforming or a nonconforming document as conforming;

– Insofar as possible, tests of nonconforming documents contain at most one error;

– Some SGML implementations use separate programs to process the prolog and the document instance. For the convenience of implementors of such systems, tests are classified by whether they exercise the prolog or the document instance;

– Each test illustrates a single SOO, or a single major SOO combined with a single minor SOO. However, multiple instances of the designated aspect of the language may appear in an individual test. For example, to illustrate that (with appropriate naming rules) case is not significant in generic identifiers, a single test may include start-tags in which the same generic identifier is entered in lowercase, uppercase, and mixed upper- and lowercase;

NOTE – When a test is intended to illustrate some conjunction of different SGML constructs, the combination is identified in the SOO. Thus, the test still illustrates a single SOO.

– The size of a test is minimized to exclude superfluous content; every construct used in the test case is directly relevant to the principal aspect of SGML being tested. This guideline is not enforced to the extreme of sacrificing the readability or comprehensibility of the test. For example, since `#PCDATA` is defined as zero or more characters, the minimal string satisfying each instance of `#PCDATA` is the empty string. Tests are more readable, however, if `#PCDATA` is realized with a short phrase relevant to the test's SOO.

NOTE – Adherence to these guidelines is often subjective. For example, individuals may disagree about whether a particular pair of tests should be commented as being closely related.

## 9 Test case naming conventions

Each ordinary test has an eight-character name, possibly with a three-character suffix. The test name is the same as the name of the corresponding SOO, as defined in 6.3 (the name of the minor SOO, if the test is based on a major and minor SOO pair). Suffixes are added to test names when a test suite includes equivalent documents. All equivalent documents have the same eight-character name and different, arbitrarily assigned suffixes.

NOTE – On computer systems where files are identified by a name and an extension, it may be convenient to place each test case in a separate file whose name is the same as the test name and whose extension is the same as the suffix.

## 10 Requirements for SGML names and literals

It may be efficient in some environments to combine multiple tests into a single document. Therefore, names shall be unique to each test. Names shall also indicate the function of the named object. Similarly, literals shall indicate their function. To meet these goals, SGML names and literals in ordinary tests are selected as follows:

– Each name or literal other than a number or number token begins with the three-character unique identifier of the test case name;

– Only lowercase letters are used in names other than reserved names and in literals;

– The test's *document element* has the same name as the test;

– In literals and in names other than that of the document element, the unique identifier is followed by a hyphen, a one-letter code to illustrate the construct being named, and a sequence number within a test. The one-letter codes are

`a` – Attribute;

`e` – Entity;

`g` – Generic identifier;

`i` – Unique identifier;

`l` – Link set;

`m` – Minimum literal;

`n` – Notation;

`s` – Short reference map;

`t` – Link type;

`v` – Value of attribute;

`x` – Miscellaneous.

The sequence numbers following these codes are assigned in the order in which they occur in the test case and are unique to each code. Leading zeros are prefixed to the sequence numbers if necessary so that, within a test, each number following a given code has the same length.

Thus, a test case named `p7b94402` that uses two entities, one attribute, and ten values would name the entities `p7b-e1` and `p7b-e2`, the attribute `p7b-a1`, and the values `p7b-v01` through `p7b-v10`.

NOTE – Although RAST produces *attribute information* in lexicographic order, the naming conventions ensure that lexicographic order usually corresponds to the order in which attributes appear in the corresponding `ATTLIST` declaration. The coincidence of the lexicographic and declaration orders simplifies manual comparison of an ordinary test and its RAST output.

The naming conventions are not applied to anomalous tests that are incompatible with them.

NOTE – For example, this naming scheme cannot be used to test:

– Name tokens that are not names;

– A parameter entity with the same name as a general entity;

– Name-length violations; and

– A variant concrete syntax's use in a name of name characters that are not name characters in the reference concrete syntax.

## 11 Conventions for testing string length

Ordinary tests dependent on the length of strings place the current string length as a decimal integer within the string in the following locations:

– immediately following each ***RS*** character;

– every ten characters within the string;

– immediately before each ***RE*** character.

***RS*** and ***RE*** characters are included in the character count. The last digit of the string length appears in the indicated position.

NOTE – For example, if the string is more than ten characters long, the digit '1' appears as the ninth character and '0' as the tenth.

The ten-character marker is omitted if the characters to represent it overlap the beginning- or end-of-record marker, or abut the preceding ten-character marker with no intervening spaces. Note that these requirements apply to the SGML source document rather than the RAST result.

```
<?
3....10........20........30........40........50........60
64....70........80........90.......100.......110.......120
125..130.......140.......150.......160.......170.......180
185..190.......200.......210.......220.......230.......240>
```

**Figure 1 - A 240-character processing instruction**

Figure 1 gives an example of the string-length convention. It shows a processing instruction that is 240 characters long, including the ***RS*** character at the beginning, and the ***RE*** character at the end, of each line.

## 12 Source document formatting conventions

Conventions for indentation, use of blank records and comments (especially at the beginning and end of each test) promote readability of tests and human scanning of information and also simplify tasks that the user of a test suite may wish to automate. For example, a fixed position for the generic identifier of the document element makes it easy to extract.

Therefore, ordinary tests follow the general pattern given below:

```
<!DOCTYPE name [
<!--Categories:
category1
category2
   .
   .
   .
-->
<!--
short description of test, including its SOO
-->
declaration1
declaration2
   .
   .
   .
]>
document instance
```

An example of such a test is the following:

```
<!DOCTYPE g5i79413 [
<!--Categories:
attribute
-->
<!--
An empty attribute value literal can
be specified if the type of the
attribute value is CDATA (Clause
7.9.4.1, Paragraph 3).
-->
<!ELEMENT g5i79413 - - ANY>
<!ELEMENT g5i-g1 - - (#PCDATA)>
<!ATTLIST g5i-g1
          g5i-a1 CDATA #IMPLIED>
]>
<g5i79413>
<g5i-g1 g5i-a1="">
</g5i-g1>
</g5i79413>
```

In particular:

– The first several records of each ordinary test contain, in order, each starting in the first character of the record:

a) the start of the *document type declaration*, through the ***dso***;

b) the beginning of a *comment declaration*, introduced by the word “Categories:”;

c) one or more records each containing a single category name as described in clause 13 (in uppercase, lowercase, or a mixture);

d) the closing delimiters for the *comment declaration*;

e) the opening delimiter for a second *comment declaration*;

f) one or more lines containing a brief description of the test. This description begins with the test's SOO, including the associated clause and paragraph numbers. The SOO may be followed by additional text explaining how the elements, data, and other constructs within the test support the principle described by the SOO.

g) the closing delimiters for the *comment declaration*;

h) one or more records containing the *document type declaration subset*, composed of several *declaration*s;

i) the closing delimiters for the *document type declaration*;

j) the *document element* begins in the next record.

– There are no spaces or blank records at the beginning or end of a test;

– Each test ends with a record end;

– In markup declarations, *ps*+ in the syntax productions of ISO 8879 is usually realized by a single ***SPACE***, or by a comment with a single ***SPACE*** on each side. To avoid a record longer than 60 characters, such a ***SPACE*** can be replaced by an ***RE***, and successive records within a single declaration may be indented. Other than an ***RE*** that separates two declarations, nothing is inserted in an ordinary test for a *ps*∗ or *ds*∗ in a production of ISO 8879, unless a separator is required;

– Each markup declaration begins in the first character of a new record.

To facilitate processing on multiple computer systems, an additional formatting requirement for ordinary tests is that records are limited to 60 characters.

Anomalous tests that deviate from the requirements for ordinary tests are required when necessary to test a particular aspect of SGML. For example, a comprehensive test suite must contain at least one test in which a markup declaration begins elsewhere than the first character of a record.

## 13 Test categories

As mentioned in clause 12, tests are classified by category. Some categories are listed below, along with an indication of whether they refer to constructs that conforming systems must support or to optional constructs. A particular test suite may define its own categories in addition to those listed here:

`ambiguous content model` (optional)
`anomalous test` (required)
`attribute` (required)
`capacity` (required)
`character data` (required)
`character reference` (required)
`character set` (optional)
`comment` (required)
`CONCUR` (optional)
`content model` (required)
`data tag` (optional)
`default entity` (required)
`delimiter-in-context` (required)
`document instance` (required)
`element` (required)
`exception` (required)
`explicit link` (optional)
`external identifier` (required)
`FORMAL` (optional)
`general entity` (required)
`implicit link` (optional)
`marked section` (required)
`markup declaration` (required)
`markup minimization` (optional)
`multicode syntax` (optional)
`non-SGML character` (optional)
`non-SGML data entity` (required)
`notation` (required)
`OMITTAG` (optional)
`optional report` (optional)
`parameter entity` (required)
`processing instruction` (required)
`prolog` (required)
`quantity` (required)
`rank` (optional)
`record boundary` (required)
`replaceable character data` (required)
`required or optional status of element` (optional)
`specific character data` (required)
`SDIF` (optional)
`separator` (required)
`SGML declaration` (optional)
`short reference` (optional)

`SHORTTAG` (optional)
`simple link` (optional)
`SUBDOC` (optional)
`tag` (required)
`unique ID` (required)
`variant concrete syntax` (optional)

## 14 The Reference Application for SGML Testing (RAST)

This clause defines RAST.

NOTES

1 As discussed in clause 3, conflicts between the definition of RAST and ISO 8879 are resolved according to ISO 8879, and conflicts between the definitions of RAST and ESIS are resolved according to ESIS.

2 The result of applying RAST to a particular document is itself neither an SGML document nor an interchange format. Its sole purpose is to indicate whether a document was parsed correctly.

3 RAST produces identical results from variations of an SGML document that are equivalent according to the provisions of ISO 8879. For example, RAST generates the same output for two SGML documents that are identical except for one or more of the following variations:

– The attribute specifications within a start-tag appear in different orders;

– One document includes a start-tag with an attribute specification in which the attribute value specification happens to be the same as the default value; the other document omits this attribute specification;

– One document includes an ***RE*** that is ignored according to the provisions of ISO 8879; the other document omits this ***RE***;

– One document uses omitted tag minimization; the other does not;

– One document uses short references; the other does not.

The above list is not exhaustive.

### 14.1 Concrete syntax of the tested document

RAST was designed to produce human-readable results from SGML documents that use the core concrete syntax, the reference concrete syntax, or some variant concrete syntax that differs from the reference concrete syntax only in its choice of short-reference delimiters.

RAST can also process documents that use other concrete syntaxes. However, for some syntaxes, such as those where the characters '`/`' and '`&`' are used as name-start characters, the result may be difficult for humans to interpret. The result is suitable, however, for machine comparison.

### 14.2 System identifiers

There is one way in which tests prepared for use with RAST may vary from system to system. System identifiers are interpreted in a system-specific fashion, and a RAST implementation need not control this interpretation. As a consequence, system identifiers in tests may have to be changed manually from system to system, and the RAST result may legitimately differ in the displayed values of system identifiers.

Two ways of minimizing the effect of this problem exist:

– minimizing the number of tests incorporating system identifiers;

– selecting system identifiers likely to be acceptable on a wide range of computer systems.

NOTE – Many computer systems accept file names of up to eight letters or digits followed by a period and a three-letter extension.

### 14.3 Processing instructions that direct RAST

RAST operates on information exchanged between an SGML parser and the rest of an SGML system. Annex A lists the information that moves in each direction. When information flows to the parser from other components, RAST determines the information content from the system data of certain processing instructions. These processing instructions are easily recognized because their system data always begin with the characters `rast`.

In particular, RAST uses processing instructions to determine which document type declarations and which link type declarations are active, to select link rules, and to determine whether to parse SGML subdocument entities. The format and interpretation of these processing instructions are given in 14.6.13, 14.6.14 and 14.6.15. The format is defined in productions using the same notation as that used in 14.6 to define RAST. To avoid confusion, productions for processing instructions are identified by letters instead of the numbers that label the productions defining the RAST result.

## 14.4 Requirements for implementing RAST

An SGML system that implements RAST needs certain capabilities beyond those required by ISO 8879. In particular, such a system shall be able to:

– produce the RAST result in a machine-readable form;

– sort strings in lexicographic order;

– interpret and act on the processing instructions defined in 14.6.13, 14.6.14 and 14.6.15, in order to test certain optional features.

Furthermore, the RAST result is expressed in a system-dependent character set. All characters mentioned in the definition of the RAST result in 14.6.2 appear in delimiter strings in the reference concrete syntax, are name characters in the reference concrete syntax, appear in the document input to RAST, or are uppercase counterparts of lowercase letters in the input document.

## 14.5 Notation

The RAST result is described formally in a variation of the production notation found in ISO 8879. The equals sign in each production is preceded by a square-brackets enclosed list of the productions that use the syntactic variable being defined. In addition, each syntactic variable after the equals sign is followed by the number of the production that defines it, also enclosed in square brackets.

The terminals of this notation are listed below:

– ***Name***, A name as defined by the concrete syntax of the parsed document (shown in the style of a “terminal variable,” even though it is not a character class)

NOTE – If the concrete syntax permits, a ***Name*** may contain nonprintable characters.

– ***Data Character***, A printable character;

– ***LE***, The system representation of the end of an output line. The ***LE*** may in fact be the same character that is used in input as the ***RE***.

NOTE – Line ends must be represented in a system-dependent manner in order to allow the RAST representation to be easily examined and manipulated by humans.

The processing instructions that direct RAST’s processing of optional SGML features as described in 14.3 use the additional terminal:

– ***String***, a sequence of SGML characters.

## 14.6 Syntax of the RAST result

This subclause defines RAST through a formal description of its result.

RAST generates one or more lines for each component of a document's structure. Lines representing data are clearly distinguishable from lines representing markup, and each type of markup is distinct. The presence of an ***LE*** at the end of each item of markup means that each item of markup and each piece of data starts on a new line.

### 14.6.1 Uppercase and lowercase letters in the RAST result

RAST outputs many names and literals from its input document. Clause 10 requires many letters in names and literals in test cases to be lowercase. In contexts in which ISO 8879 mandates substitution of the uppercase counterpart for a lowercase letter, RAST outputs the uppercase form.

NOTE – The requirements of clause 10 and uppercase substitution allow RAST to test interpretation of SGML naming rules.

### 14.6.2 The RAST result

[1] RAST result [11]=
(*processing instruction data*[18]$*$,
((*active link*[26]$*$,
*base document element*[2]) |
*concurrent document element*[33]+),
*processing instruction data*[18]$*$) |
((“`#ERROR`” | “`#RAST-PI-ERROR`”), ***LE***)

RAST produces the single word `#ERROR` if the parsed document is not a valid SGML document. An implementation of RAST within a conforming SGML system that is not a validating SGML system need not be able to produce the error indication. An implementation of RAST that interprets the processing instructions defined in 14.6.13, 14.6.14 and 14.6.15 produces `#RAST-PI-ERROR` if a processing instruction with system data beginning `rast` does not meet the requirements of one of those three subsubclauses.

NOTES

1 An implementation of RAST may need to discard a partial output file to produce either error indication.

2 RAST generates `#RAST-PI-ERROR` when it expects a processing instruction that was not provided.

3 `#RAST-PI-ERROR` indicates an error in a test case or in an implementation of RAST. It should never appear in the RAST results provided by a test suite.

RAST uses *processing instruction data* to show the system data of all processing instructions that occur in a document, in the order in which the processing instructions appear. It reports processing instructions that occur in the prolog prior to generating any *active link*, *base document element*, or *concurrent document element*. Similarly, it reports processing instructions that appear at the end of the SGML document after it finishes the *base document element* or the last *concurrent document element*. All other processing instructions are reported within the *base document element* or a *concurrent document element* (a single processing instruction may be reported within more than one *concurrent document element*).

RAST produces at least one *concurrent document element* when one or more active document types are identified. Otherwise, it produces a *base document element*, possibly preceded by one or more *active link* specifications. RAST's use of *active link* is explained in 14.6.14; that of *concurrent document element* in 14.6.15.

### 14.6.3 Elements

[2] base document element [1]=
  *document element*[3]

[3] document element [2]= *simple link information*[28]∗, *parsed element*[4]

[4] parsed element [3, 5, 33]= *element start*[6], *parsed content*[5]∗, *element end*[7]

[5] parsed content [4]=
  *external entity*[17] | *parsed element*[4] |
  *processing instruction data*[18] |
  *data line*[20] | *internal sdata entity*[19] |
  *link set information*[30]

[6] element start [4]= "`[`", ***Name***,
  (***LE***, ((*attribute information*[8]+,
  *link information*[29]?) |
  *link information*[29]))?,
  "`]`", ***LE***

[7] element end [4]= "`[/`", ***Name***, (***LE***, *link set information*[30])?, "`]`", ***LE***

Each element in an SGML document is represented in the RAST result by a *parsed element*. The document element, all proper subelements, and all included elements are represented in the same manner. This representation includes both an *element start* and an *element end*, even if the element is required to be entered without an end-tag because it has declared content `EMPTY` or has a specified content-reference attribute.

The ***Name*** in the *element start* and *element end* surrounding the (possibly empty) *parsed content* of an element is the generic identifier of the element being represented. The closing bracket ending an *element end* always appears on the same line as the element's name. The closing bracket ending an *element start* appears on a line by itself unless the *element start* has no *attribute information* and no *link information*.

RAST represents every item of content with *parsed content*. It reports items of *parsed content* in the same order as the corresponding information appears in the parsed document.

### 14.6.4 Attributes

RAST reports *attribute information* for each attribute in the element's `ATTLIST` declaration. RAST reports attributes in the lexicographic order of the attribute names. This order is not necessarily the order in which the attributes are specified in the document instance, but is ordinarily the order in which they appear in the `ATTLIST` declaration (see clause 10).

[8] attribute information [6, 12, 27, 28, 31, 32]=
  ***Name***, "=", ***LE***, (("`#IMPLIED`", ***LE***) |
  ((*markup data*[21] | *internal sdata entity*[19])∗, (*external id information*[14] |
  *entity information*[9]+)?))

The ***Name*** in an item of *attribute information* is that of the attribute for which information is being given.

For unspecified impliable attributes, including unspecified content reference attributes, the *attribute information* is `#IMPLIED`. Otherwise, the value is represented by zero or more instances of *markup data* and *internal sdata entity*. The latter can appear only in the representation of character data attribute values. In all other respects, the values of all types of attributes appear the same.

Each notation attribute value that appears in the RAST result is followed immediately by the *external id information* for the notation.

Each value for a general entity name attribute or a general entity name list attribute is followed by the *entity information* for each of the entity names that appear in the attribute value. Where there is more than one entity name, the *entity information* is given for the different entity names in the same order as the entity names appear in the attribute value.

NOTE – The attribute value is displayed as recognized by an SGML parser – after entity expansion and, where appropriate, removal of extra spaces and case shifting of letters. In attribute values that consist of two or more names, name tokens, numbers, number tokens, entity names or ID references, consecutive tokens are separated by one ***SPACE***.

### 14.6.5 Entity information

[9] entity information [8]=
(*external entity information*[10] | *internal entity information*[13]), "`#END-ENTITY`", ***LE***

[10] external entity information [9, 17]=
("`#SUBDOC`", ***LE***, *external id information*[14], *parsed subdocument*[11]?) |
("`#CDATA-EXTERNAL`", ***LE***, *entity information details*[12]) |
("`#SDATA-EXTERNAL`", ***LE***, *entity information details*[12]) |
("`#NDATA-EXTERNAL`", ***LE***, *entity information details*[12])

[11] parsed subdocument [10]=
"`#PARSED-SUBDOCUMENT`", ***LE***, *RAST result*[1]

[12] entity information details [10]=
*external id information*[14], "`#NOTATION=`", ***Name***, ***LE***, *external id information*[14], *attribute information*[8]

[13] internal entity information [9]=
("`#CDATA-INTERNAL`", ***LE***, *markup data*[21]) |
("`#SDATA-INTERNAL`", ***LE***, *markup data*[21]*)

Whether *external entity information* includes a *parsed subdocument* is discussed in 14.6.13.

In *entity information details*, the first *external id information* is that declared for the entity, and the second is that declared for its notation. RAST reports this information both when the entity is referenced in content and when the entity's name appears in the value of a general entity name attribute or a general entity name list attribute.

RAST reports *attribute information* for each attribute declared for the associated notation. As with start-tag attributes, data attributes are displayed in lexicographic order by attribute name.

### 14.6.6 External identifiers

[14] external id information [8, 10, 12]=
(*public identifier information*[15], *system identifier information*[16]?) |
*system identifier information*[16] |
("`#SYSTEM`", ***LE***, "`#NONE`", ***LE***)

[15] public identifier information [14]=
"`#PUBLIC`", ***LE***, (*markup data*[21]+ | ("`#EMPTY`", ***LE***))

[16] system identifier information [14]=
"`#SYSTEM`", ***LE***, (*markup data*[21]+ | ("`#EMPTY`", ***LE***))?

RAST generates *public identifier information* when it reports an external identifier that has a public identifier. If the external identifier also has a system identifier, RAST also generates *system identifier information*. When reporting an external identifier that does not have a public identifier, RAST generates *system identifier information*. In both *public identifier information* and *system identifier information*, `#EMPTY` indicates the empty literal. RAST uses `#NONE` within *system identifier information* to indicate that an external identifier has neither a public identifier nor a system identifier (that is, that the external identifier consists solely of the reserved name `SYSTEM`).

NOTES

1 RAST displays the text of public and system identifiers as normalized by an SGML parser. For example, extra spaces are removed from public identifiers.

2 RAST reports the public and system identifiers for notations and external data entities and does not reflect any resolution the entity manager makes of these identifiers. No meaningful resolution of an external data entity is needed if an SGML document will never be processed by an application other than RAST, as is the case for documents intended only for testing SGML parsers.

### 14.6.7 External entities

[17] external entity [5]= "`[&`", ***Name***, ***LE***, *external entity information*[10], "`]`", ***LE***

The ***Name*** in an *external entity* is the name of the referenced general entity.

The closing bracket ending an *external entity* always appears on a line by itself.

### 14.6.8 Processing instructions

[18] processing instruction data [1, 5]= "`[?`", (***LE***, *data line*[20]+)?, "`]`", ***LE***

The data lines in *processing instruction data* contain the text of a processing instruction recognized by the SGML parser.

The closing bracket ending *processing instruction data* appears on a line by itself unless the processing instruction being represented contained no text.

NOTE – RAST produces *processing instruction data* even for those processing instructions, defined in 14.6.13, 14.6.14.1, and 14.6.15, that it interprets to determine processing of `SUBDOC`, `LINK` and `CONCUR` features.

### 14.6.9 Data

[19] internal sdata entity [5, 8]= "`#SDATA-TEXT`", ***LE***, *markup data*[21]∗, "`#END-SDATA`", ***LE***

[20] data line [5, 18]= ("`|`", ***Data Character***+, "`|`", ***LE***) | *special character*[22]

[21] markup data [8, 13, 15, 16, 19]= ("`!`", ***Data Character***+, "`!`", ***LE***) | *special character*[22]

[22] special character [20, 21]= ("#RS", ***LE***) | ("#RE", ***LE***) | ("#TAB", ***LE***) | ("#", *character number*[23], ***LE***)

[23] character number [22]= *nonnegative integer*[24]

[24] nonnegative integer [23, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51]= "`0`" | (*non-zero digit*[25], ("`0`" | *non-zero digit*[25])∗)

[25] non-zero digit [24]= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

A number that is used as a *character number* is a decimal number, with no extra leading zeros, that represents a character.

### 14.6.10 Uniform display of data

RAST displays all the following data values in a uniform manner:

– `#PCDATA` content;

– processing instructions;

– attribute values;

– the expansion of specific-character data entities;

– public and system identifiers.

All the above values are surrounded by exclamation marks except for `#PCDATA` content and processing instructions, which are surrounded by vertical bars. The only differences are the limitations imposed by ISO 8879 on the data that may appear in each context. For example, public identifiers may not contain record-ends.

### 14.6.11 Internal SDATA entities

An *internal sdata entity* in `#PCDATA` or a character data attribute value need not contain any characters. If multiple adjacent specific character data entity references occur, RAST produces an *internal sdata entity* for each reference.

NOTES

1 The text of a specific character data entity is represented differently when the entity is referenced in content than when the entity's name appears in the value of a general entity name or general entity name list attribute. In the first case, the text is surrounded by lines containing `#SDATA-TEXT` and `#END-SDATA`, and in the second `#SDATA-INTERNAL` and `#END-ENTITY`.

2 References to `SDATA` entities can occur in SGML documents in contexts not reflected in ESIS and therefore not reported by RAST, for example, in an attribute value literal of an attribute whose declared value is other than `CDATA`.

### 14.6.12 Data characters

RAST shows data characters as follows:

– A printable character always appears as a ***Data Character***. The character code that is used to represent the letter 'A', for instance, is always displayed as 'A', even in specific character data, where it may not represent the letter 'A';

– The character codes that are used to represent record-start, record-end, and tab always appear as `#RS`, `#RE`, and `#TAB`, respectively, on separate lines, even in specific character data. The tab character is reported as `#TAB` regardless of whether it is declared in the SGML declaration to be a function character.

NOTE – Since, unlike other data characters, these representations are not surrounded by vertical bars or exclamation marks, the occurrence of one of these characters is distinct from a sequence of data characters that happens to spell one of these codes;

– All other characters are represented by a character number.

NOTE – The above requirements pertain only to data characters. All characters are represented by themselves within a ***Name.*** In particular, while RAST represents nonprintable characters other than record-start, record-end, and tab by character numbers when they occur as data characters, it outputs such characters directly when they occur in a ***Name***.

RAST never produces more than 60 ***Data Character***s on a single output line. A sequence of more than 60 consecutive ***Data Character***s is divided into several lines, each, except possibly the last, containing exactly 60 ***Data Character***s (plus whatever other characters are required on the line). In particular, RAST produces multiple instances of *data line* or *markup data* when more than 60 consecutive ***Data Character***s occur.

Every line of data contains some data characters; that is, RAST does not generate a data line when an element's content is empty or the value of a character data attribute has no characters.

NOTE – Exclamation marks rather than vertical bars are used to delimit data that appear inside markup. This convention, which applies to attribute values, specific character data entities, and system and public identifiers, makes the distinction between data within markup and other data clear to human readers of the RAST result.

### 14.6.13 SGML subdocument entities[1)]

[a] parse subdocument = “rast-parse-subdoc: ”, (“yes” | “no”)

A *parse subdocument* processing instruction indicates whether or not a following *external entity information* for an SGML subdocument entity includes a *parsed subdocument*. When RAST encounters a *parse subdocument* processing instruction, it adds it to the end of a list of saved *parsed subdocument* processing instructions. When a reference to an SGML subdocument entity occurs, or the name of an SGML subdocument entity occurs as the value of a general entity name attribute or a token in the value of a general entity name list attribute, RAST removes the *parse subdocument* processing instruction at the beginning of the list. If the processing instruction specifies no, or if the list was empty, RAST does not output a *parsed subdocument* in the entity’s *external entity information*. If the processing instruction specifies yes, RAST outputs a *parsed subdocument*.

SGML subdocument entity names in attribute values are processed in the order RAST reports them. In particular, if an element has two general entity name attributes and the values of both are names of SGML subdocument entities, RAST uses the first applicable *parse subdocument* processing instruction to determine whether to report a *parsed subdocument* for the value of the attribute whose name lexicographically precedes the other. If the value of a general entity name list attribute contains multiple tokens that are names of SGML subdocument entities, RAST applies *parse subdocument* processing instructions to the tokens in the order they appear in the attribute value.

If an SGML subdocument entity for which the applicable processing instruction specifies `yes` is not a valid SGML document, RAST outputs `#ERROR` as the result of the entire document; in other words, RAST does not output the error indication simply as a *parsed subdocument* for the invalid subdocument. Otherwise, if such an entity contains a processing instruction with system data beginning `rast` that does not meet the requirements of this subsubclause or of 14.6.14.1 or 14.6.15, RAST outputs `#RAST-PI-ERROR` as the result of the entire document.

### 14.6.14 LINK

This subsubclause describes how RAST reports the optional link features of SGML.

#### 14.6.14.1 Processing instructions used with link features

[b] active lpd = “`rast-active-lpd:`”, ***Name***, (“`,`”, ***Name***)∗

[c] link rule selection = “`rast-link-rule:`”, ***String***

RAST interprets the first *active lpd* or *active dtd* (see 14.6.15) processing instruction to appear in the document. Each ***Name*** in an *active lpd* processing instruction is that of a link type declaration to be made active. If the naming rules of the concrete syntax specify uppercase substitution for general names, all letters in the ***Name*** must appear in uppercase.

NOTE – When the naming rules specify uppercase substitution, a ***Name*** that appears with some lowercase letters in a link type declaration must appear with all uppercase letters in a processing instruction. This requirement on processing instructions allows RAST to process tests involving the optional link features regardless of whether letters in a link type name appear in uppercase or lowercase. Without it, an implementation of RAST would need access to the naming rules of the concrete syntax to investigate an SGML system’s support of the uppercase substitution specified in ISO 8879. Since the naming rules are not

1) Recall from 14.3 that productions for processing instructions that occur within test cases are identified by letters to distinguish them from the productions identified by numbers that define the RAST result.

included in ESIS, the above requirement on processing instructions increases the feasibility of implementing RAST in any particular ESIS-based SGML system.

The following conditions are errors that cause RAST to output `#RAST-PI-ERROR` as the result of the entire document:

– The processing instruction appears after the start of the base document type declaration;

– A ***Name*** in the *active lpd* is not the link type name of a link type declaration in the prolog;

– The same string is used as more than one ***Name*** in the *active lpd*;

– The processing instruction lists more ***Name***s than the SGML declaration permits to be active;

– The source document type name in an explicit or implicit link specification is not that of the base document type declaration.

A *link rule selection* appears in the document instance. It specifies the value of a link attribute to use for selecting a link rule when the next element starts. In particular, RAST selects a link rule that has at least one attribute whose interpreted and, if the attribute is not character data, tokenized value is the ***String*** in the *link rule selection*.

A sequence of adjacent *link rule selection*s specifies link attributes for successive following elements. Such a sequence is useful when a test case involves a sequence of omitted start-tags.

The following conditions are errors that cause RAST to output `#RAST-PI-ERROR` as the result of the entire document:

– No *link rule selection* is given prior to an element with more than one applicable link rule;

– More than one rule, or none, have at least one attribute whose interpreted, tokenized value is ***String***.

### 14.6.14.2 Active links

[26] active link [1]= “`#ACTIVE-LINK=`”, ***Name***, ***LE***, (“`#INITIAL`”, ***LE***, *result element specification*[27]+)?, “`#END-ACTIVE-LINK`”, ***LE***

[27] result element specification [26, 30]= “`[`”, ***Name***, ***LE***, *attribute information*[8]+, “`]`”, ***LE***

RAST generates an *active link* for each active link type declaration. The ***Name*** in an *active link* is that of an active link type declaration. The ***Name*** in a *result element specification* is the generic identifier of the result element. If present, `#INITIAL` introduces the result element specifications of link rules whose source element specification is implied in the initial link set. If there is more than one *active link*, they appear in the same order as the link names appear in the *active lpd* processing instruction; the *result element specification*s appear in the lexicographic order of the ***Name***s.

### 14.6.14.3 Simple link information

[28] simple link information [3]= “`#SIMPLE-LINK=`”, ***Name***, ***LE***, *attribute information*[8]∗, “`#END-SIMPLE-LINK`”, ***LE***

RAST reports the *link type name* and attribute information for each active simple link in *simple link information* at the beginning of the *document element*. The *simple link information* data appear in the order the *link type name*s appeared in the *active lpd* processing instruction.

NOTE – RAST produces both an *active link* and *simple link information* for each active simple link.

### 14.6.14.4 Link Information

[29] link information [6]= (*link set information*[30], *link rule information*[31]?) | *link rule information*[31]

[30] link set information [29]= “`#LINK-SET-INFO`”, ***LE***, *result element specification*[27]+

[31] link rule information [29]= “`#LINK-RULE`”, ***LE***, *attribute information*[8]∗, *link result*[32]?

[32] link result [31]= “`#RESULT=`”, ((***Name***, ***LE***, *attribute information*[8]∗) | (“`#IMPLIED`”, ***LE***))

If there is a current link set, RAST includes *link information* within an *element start*. The *link information* includes *link set information* if the current link set has link rules whose source element specifications are implied. In this case, the rules are listed in the lexicographic order of the ***Name***s in the *result element specification*s.

If the started element is an associated element type for a link rule in the current link set, RAST reports *link rule information*. Any *attribute information* reports the link attributes of the selected rule, determined as described in 14.6.14.1.

A *link result* is included in the *link rule information* if the active link is an explicit link. It gives the result element specification with any result attributes.

RAST reports *link set information* within an *element end* if the link set current after the element has link rules whose source element specifications are implied. RAST also reports *link set information* after a link set use declaration if the specified link set has link rules whose source element specifications are implied.

### 14.6.15 CONCUR

[d] active dtd = "`rast-active-dtd:`", ***Name***, ("`,`", ***Name***)∗

[33] concurrent document element [1]= "`#CONCUR=`", ***Name***, ***LE***, *parsed element*[4]

RAST interprets the first *active dtd* or *active lpd* (see 14.6.14) processing instruction to appear in the document. Each ***Name*** in an *active dtd* processing instruction is that of a *document type declaration* to be made active. If the naming rules of the concrete syntax specify uppercase substitution for general names, all letters in the ***Name*** must appear in uppercase.The following conditions are errors:

– The processing instruction appears after the start of the base document type declaration;

– A ***Name*** in the *active dtd* is not the document type name of a document type declaration in the prolog;

– The SGML declaration does not allow all the specified document type declarations to be active.

If it uses an *active dtd* processing instruction, instead of a *base document element*, RAST produces a *concurrent document element* for each active document type declaration, in the order their names appear in the *active dtd* processing instruction. The ***Name*** in each *concurrent document element* is that of the associated active document type declaration.

## 15 The Reference Application for Capacity Testing (RACT)

To test conformance to the capacity constraints of SGML, a test suite can include one document that reaches each capacity defined in ISO 8879 and another that barely exceeds it. A validating parser must correctly process the first document and report an error on the second. However, such tests are difficult to create using the reference capacity set for two reasons. First, the limits in the reference capacity set are large, and hence can only be tested with large documents. Second, all the capacities in the reference capacity set are the same. It is therefore impossible to exceed any other capacity without exceeding "`TOTALCAP`". There is no requirement that validating or conforming SGML systems be able to define a variant capacity set. RACT was therefore defined to provide an optional method of evaluating this aspect of SGML parsing. RACT reports a validating parser's capacity calculations for an SGML document. As with RAST, there is no requirement that a parser be able to support RACT.

RACT generates one line for each capacity, in the order given in figure 5 of ISO 8879 (which defines the reference capacity set). The line consists of the capacity name, a single space, and the number of capacity points used in that category in that document.

RACT is formally defined below, in the same notation as that used in clause 14. Note that ***LE*** and *nonnegative integer* are defined in clause 14. As in ISO 8879, ***SPACE*** denotes the space character.

[34] RACT result = *totalcap*[35], *entcap*[36], *entchcap*[37], *elemcap*[38], *grpcap*[39], *exgrpcap*[40], *exnmcap*[41], *attcap*[42], *attchcap*[43], *avgrpcap*[44], *notcap*[45], *notchcap*[46], *idcap*[47], *idrefcap*[48], *mapcap*[49], *lksetcap*[50], *lknmcap*[51]

[35] totalcap [34]= "`TOTALCAP`", ***SPACE***, *nonnegative integer*[24], ***LE***

[36] entcap [34]= "`ENTCAP`", ***SPACE***, *nonnegative integer*[24], ***LE***

[37] entchcap [34]= "`ENTCHCAP`", ***SPACE***, *nonnegative integer*[24], ***LE***

[38] elemcap [34]= "`ELEMCAP`", ***SPACE***, *nonnegative integer*[24], ***LE***

[39] grpcap [34]= "`GRPCAP`", ***SPACE***, *nonnegative integer*[24], ***LE***

[40] exgrpcap [34]= "`EXGRPCAP`", ***SPACE***, *nonnegative integer*[24], ***LE***

[41] exnmcap [34]= "`EXNMCAP`", ***SPACE***, *nonnegative integer*[24], ***LE***

[42] attcap [34]= "ATTCAP", ***SPACE***, *nonnegative integer*[24], ***LE***

[43] attchcap [34]= "ATTCHCAP", ***SPACE***, *nonnegative integer*[24], ***LE***

[44] avgrpcap [34]= "AVGRPCAP", ***SPACE***, *nonnegative integer*[24], ***LE***

[45] notcap [34]= "NOTCAP", ***SPACE***, *nonnegative integer*[24], ***LE***

[46] notchcap [34]= "NOTCHCAP", ***SPACE***, *nonnegative integer*[24], ***LE***

[47] idcap [34]= "IDCAP", ***SPACE***, *nonnegative integer*[24], ***LE***

[48] idrefcap [34]= "IDREFCAP", ***SPACE***, *nonnegative integer*[24], ***LE***

[49] mapcap [34]= "MAPCAP", ***SPACE***, *nonnegative integer*[24], ***LE***

[50] lksetcap [34]= "LKSETCAP", ***SPACE***, *nonnegative integer*[24], ***LE***

[51] lknmcap [34]= "LKNMCAP", ***SPACE***, *nonnegative integer*[24], ***LE***

The value of each *nonnegative integer* is the number of points of the indicated capacity in the document.

## 16 Test suite reports

This clause describes how to report the results of testing an SGML system with a test suite. The information described here shall be accompanied by the documentation described in clause 6.

The performance of a particular SGML system on a particular test suite cannot be adequately reported by a single score or quantity. Any attempt to do so might result in a low score for a conforming system that does not implement an optional feature repeatedly tested in the test suite. Furthermore, a user of SGML who has no need for some required construct may prefer to select a system that has incorrectly implemented that required construct in favor of a conforming system that does not provide an optional feature the user plans to use. Therefore, results of running a test suite are reported as described below:

– The report lists the following for each category of test:

  – the number of tests in the category;

  – if the tested system includes a validating parser, the number of tests in the category that the test suite and the tested system

    – agree are conforming documents;

    – agree are erroneous documents;

    – do not agree to be conforming or erroneous.

  – if both the test suite and the tested system support RAST, the number of tests in the category that are conforming documents and for which the RAST results produced by the tested system

    – are the same as those provided with the test suite;

    – differ from those provided with the test suite;

    – cannot be compared to those provided with the test suite because only one of the two implementations of RAST supports an optional feature used in the test;

  – if the test suite and the tested system both support RACT, the number of tests in the category for which the RACT result generated by the tested system is

    – the same as that provided with the test suite;

    – different from that provided with the test suite;

– names of tests on which the test suite and the tested system produce different results;

– a pairwise comparison of all test categories, constructed as follows from the category comments at the beginning of the tests. Each test category is compared to every other category by counting the number of tests in which both categories appear in the comments. This information helps the reader of the report determine the significance of the results. For example, suppose a tested system has correctly implemented a construct *x*, but misinterpreted a construct *y*. If all tests of *x* happen to involve *y* as well, the high count of tests in which the system and the test suite produce different results might suggest a problem in the implementation of *x*. Information about the overlap of *x* and *y* tests, however, informs the user that the problem might lie in the latter category.

NOTE – In practice, a test suite may be developed in conjunction with two or more implementations of RAST. All should produce the same results for every test. If this is not the case, and the implementor of the test suite cannot determine which result is correct, all variants shall be distributed with the test suite. The documentation shall clearly identify tests for which multiple results are provided.

## 17 Testing SDIF data streams

A test suite that evaluates a system's use of SDIF shall be comprehensive. That is, a general purpose SGML test suite that includes tests of SDIF shall explore conformance to every aspect of ISO 9069. A test suite restricted to a particular SGML application that uses SDIF shall provide tests to explore every aspect of SDIF relevant to that application. Every general purpose SGML test suite that tests SDIF shall test both the creation of an SDIF data stream from separate entities and the separation of an SDIF data stream into multiple entities. Applying these operations in sequence should result in recreation of the original entities, with the possible exception that corresponding entity declarations may have different system identifiers.

(Blank page)

# Annex A
(normative)

## The ISO 8879 Element Structure Information Set (ESIS)

This annex describes the Element Structure Information Set (ESIS) which is implicit in ISO 8879.

NOTE – The provisions of clause 3 regarding conflict with ISO 8879 apply.

There are two kinds of SGML application (and therefore two kinds of conforming SGML application):

a) A structure-controlled SGML application operates only on the element structure that is described by SGML markup, never on the markup itself;

b) A markup-sensitive SGML application can act on the actual SGML markup and can act on element structure information as well. Examples include SGML-sensitive editors and markup validators.

The set of information that is acted upon by implementations of structure-controlled applications is called the "element structure information set" (ESIS). ESIS includes properties of the element structure itself, plus other information. ESIS is implicit in ISO 8879, but is not defined there explicitly. The purpose of this annex is to provide that explicit definition.

ESIS is particularly significant for SGML conformance testing because two SGML documents are equivalent documents if, when they are parsed with respect to identical DTDs and LPDs, their ESIS is identical. All structure-controlled applications must therefore produce identical results for all equivalent SGML documents. In contrast, not all markup-sensitive applications will produce identical results from equivalent documents. (For example, a program that prints comment declarations or that counts the number of omitted end-tags.)

ESIS information is exchanged between an SGML parser and the rest of an SGML system that implements a structure-controlled application. Although an implementation may choose to "wire in" some of ESIS, such as the names of attributes, a structure-controlled application need have no other knowledge of the prolog than what ESIS provides.

A system implementing a structure-controlled application is required to act only on ESIS information and on the APPINFO parameter of the SGML declaration.

NOTES

1 This requirement does not prohibit a parser from providing the same interface to both structure-controlled and markup-sensitive applications, which could include non-ESIS information (e.g., the date), and/or information that could be derived from ESIS information (e.g., the list of open elements).

2 The documentation of a conforming SGML system that supports user-developed structure-controlled applications should make application developers aware of this requirement. Such a system should facilitate conformance to this requirement by distinguishing ESIS information from non-ESIS in its interface to applications. Note 1 in 15.3.5 of ISO 8879 applies only to structure-controlled applications.

In the following description of ESIS, information is identified as being available at a particular point in the parsed document. This identification should not be interpreted as a requirement that the information actually be exchanged at that point – all or part of it could have been exchanged at some other point. Similarly, there is no constraint on the manner (e.g., number of function calls) or format in which the exchanges take place.

The ESIS description includes the information associated with all of the SGML optional features. When a given feature is not in use, corresponding information is not present in the document. ESIS information is transmitted from the parser to the application unless otherwise indicated.

ESIS information applies to a single parsed document instance. Therefore, if concurrent instances are being parsed, the applicable document type name must be identified. This requirement also applies when parsing intermediate instances in a chain of active links.

ESIS information consists of the identification of the following occurrences, and the passing of the indicated information for each:

a) Initialization

– The application must inform the SGML parser of the active document types, the active link types, or that parsing is to occur only with respect to the base document type.

b) Start of document instance set

– For each active LPD, the link type name and link set information (see (m) below) for the initial link set.

c) Start of document element only

– For each active simple link, the link type name and attribute information (see (j) below) for the link attributes.

d) Start of any element

– Generic identifier;

– Attribute information for the start-tag;

– For each applicable link rule, attribute information for the link attributes;

– The application must inform the SGML parser which applicable link rule it chose;

– For the chosen link rule, the result GI and attribute information for the result element;

– If the element has an associated link set, the link set information.

e) End of any element, including elements declared to be empty

– Generic identifier

– Link set information for the link set that is current immediately after the element (including processing any relevant "#POSTLINK" parameter)

NOTE – If the element was empty, ESIS does not indicate why it was empty; that is, whether it was declared to be empty, or whether an explicit content reference occurred, or whether it just happened to contain no data characters, subelements, or other content.

f) End of document instance set

NOTE – Processing instructions could occur between the end of the document element and the end of the document instance set.

g) Processing instruction

– System data

h) Link set use declaration

– Link set information

i) Data

– Includes no ignored characters (e.g., record starts);

– Includes only significant record ends, with no indication of how significance was determined. Characters entered via character references are not distinguished in any way. Implementation-specific means can be used to represent bit combinations that the application cannot accept directly.

NOTES

1 Such bit combinations may be those of non-SGML characters entered via character references, but no significance is attached to this coincidence.

2 Bit combinations of non-SGML characters that occurred directly in the source text would have been flagged as errors, and would therefore never be treated as data.

j) Attribute information

– All attribute values must be reported and associated with their attribute names.

NOTES

1 For example, a parser could supply the attribute names with each value, or supply the values in an order that corresponds to a previously supplied list of names.

2 The order of the tokens in a tokenized attribute value shall be preserved as originally specified.

– Each unspecified impliable attribute must be identified;

NOTE – For example, a parser could identify such attributes explicitly, or it could allow the application to determine them by comparing the identified specified attribute values to a previously supplied list of attribute names.

– There shall be no indication of whether an attribute value was the default value;

– The order in which attributes are specified in the attribute specification list is not part of the ESIS;

– General entity name attribute values include the entity name and entity text. The entities themselves are not treated as having been referenced;

NOTE – An application can use system services to parse the entities, but such parsing is outside the context of the current document.

– For notation attributes, the attribute value includes the notation name and notation identifier;

– For CDATA attributes, references to SDATA entities in attribute value literals are resolved. The replacement text is distinguished from the surrounding text and identified as an individual SDATA entity;

– For CDATA attributes, references to CDATA entities in attribute value literals are resolved. The replacement text is not distinguished from the surrounding text.

k) References to internal entities

– The information passed to the application depends on the entity type:

SDATA: replacement text, identified as an individual SDATA entity.

PI: replacement text, identified as a processing instruction but not as an entity.

– For other references, nothing is passed to the application.

NOTE – The replacement text is parsed in the context in which the reference occurred, which can result in other ESIS information being passed.

l) References to external entities

The information passed to the application depends on the entity type:

– For data entities, the entity name and entity text are passed. If a notation is named, the notation name, notation identifier, and attribute information for the data attributes are also passed.

– For SGML text entities, nothing is passed to the application.

NOTE – The replacement text is parsed in the context in which the reference occurred, which can result in other ESIS information being passed.

– For SUBDOC entities, the entity name and entity text are passed. The application can require that the subdocument entity be parsed at the point at which the reference occurred.

NOTE – Parsing of the subdocument entity can result in other ESIS information being passed. The occurrence of the end of the document instance set of the subdocument entity will indicate that subsequent ESIS information applies to the element from which the subdocument entity was referenced.

m) Link set information

– All link rules whose source element specification is implied.

# Annex B
(informative)

## Sample tests and RAST results

This annex contains several typical test cases and their RAST results. These examples illustrate both ordinary tests and RAST output**.**

### B.1 A typical conforming document

The following is a typical test of a conforming document:

```
<!DOCTYPE g01b2404 [
<!--Categories:
element
markup declaration
prolog
-->
<!--
#PCDATA is a primitive content token (Clause 11.2.4,
Paragraph 4, Production 129).
-->
<!ELEMENT g01b2404 - - (g01-g1)>
<!ELEMENT g01-g1 - - (#PCDATA)>
]>
<g01b2404>
<g01-g1>
parsed character data
</g01-g1>
</g01b2404>
```

Its RAST result is:

```
[G01B2404]
[G01-G1]
|parsed character data|
[/G01-G1]
[/G01B2404]
```

### B.2 An erroneous prolog

The following is a typical test of a document with an erroneous prolog:

```
<!DOCTYPE p01b2201 [
<!--Categories:
element
markup declaration
prolog
-->
<!--
Omitted tag minimization includes start-tag minimization and
end-tag minimization (Clause 11.2.2, Paragraph 1,
Production 122).
-->
```