

IEEE Trial-Use Standard Reference Model for Computing System Tool Interconnections

IEEE Standards Board

Approved December 5, 1991

Sponsor

Standards Coordinating Committee
of the
IEEE Computer Society

Abstract: Reference models for tool-to-organization interconnections, tool-to-platform interconnections, and information transfer among tools are provided. The purpose is to establish agreements for information transfer among tools in the contexts of human organization, a computer system platform, and a software development application. To make the transfer of semantic information among tools easier, a semantic transfer language (STL) is also provided. Interconnections that must be considered when buying, building, testing, or using computing system tools for specifying behavioral descriptions or requirements of system and software products are described.

Keywords: information transfer, reference model, semantic transfer language (STL), tool-to-organization interconnections, tool-to-platform interconnections.

The Institute of Electrical and Electronics Engineers, Inc.

New York, NY 10017-2394, USA

Copyright © 1992 by the Institute of Electrical and Electronics Engineers, Inc. All rights reserved. Published 1992 Printed in the United States of America

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE which have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old, and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08555-1331
USA

IEEE Standards documents are adopted by the Institute of Electrical and Electronics Engineers without regard to whether their adoption may involve patents on articles, materials, or processes. Such adoption does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the standards documents.

Publication of this trial-use standard for comment and criticism has been approved by the Institute of Electrical and Electronics Engineers. Trial-use standards are effective for 24 months from the date of publication. Comments for revision will be accepted for 18 months after publication. Suggestions for revision should be directed to the Secretary, IEEE Standards Board, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, and should be received no later than February 17, 1994. It is expected that following the 24-month period, this trial-use standard, revised as necessary, shall be submitted to the IEEE Standards Board for approval as a full-use standard.

Foreword

(This foreword is not a part of IEEE Std 1175, IEEE Trial-Use Standard Reference Model for Computing System Tool Interconnections.)

Professionals developing computing systems are always looking for ways to increase the quality of their products and the productivity of their development efforts. Many different quality and productivity improvement ideas appear in publications every day. These ideas include new (or revised) life cycle models, methodologies, standards, and tools. All these ideas have the potential to improve the quality of computing systems and the productivity of developing computing systems, but which ones should computing system professionals build, buy, or use?

This question is hard to answer. If the world could be frozen in time so nothing would change from the time the question is asked until a new idea was implemented, then the answer would be easy--select the new idea with the highest return on investment. Unfortunately, time is not frozen, and professionals must deal with the problems that come with constantly changing technology.

New ideas are seldom stand-alone solutions. Most ideas mean changes must be made in an organization. People in an organization may be required to learn and apply a new methodology. They may be required to convert all their old information files into new formats. Because these new ideas represent organizational change, the relationships between the organization and the new idea must be evaluated.

Another problem with new ideas is that they are often implemented in an organization through some form of software tool. That tools must run on some hardware/software platform, and that means that the platform must be changed to accommodate the new tool. This is another change that must be evaluated.

While the benefits from one tool may be substantial, the benefits from a number of tools working together should be even greater. In order for tools to work together, they must be able to transfer information to and from one another. That means changes to the interconnections among tools. A third type of change must then be evaluated.

The computing system industry is a competitive one that *dictates* constant improvements in quality and productivity. Constant change means that the builder, buyer, and user of the new tools will need to mix and match tools from different sources as the “best” tools become known and as new, innovative ideas and tools become available.

The purpose of this standard is to help builders, buyers, testers, and users of tools interconnect or integrate tools into families. If this standard were ideal, tools that follow this standard would “plug and play” into an organization, into a hardware/software platform, and into a family of tools. The interconnections would be “seamless” and easy to accomplish. While the goal of seamless interconnections may not be fully realized, tools that adhere to this standard should be easier to interconnect than tools that do not.

At the time the standard was balloted, the Tools Task Force Executive Committee had the following membership:

Cory S. Clinger
Francois Coallier
Peter L. Eirich
Paul C. Jorgensen

Robert M. Poston, *Chair*

Dwayne L. Knirk
Thomas M. Kurihara
Tomoo Matsubara
Masao Matsumoto

William David Sharon
Yuji Shinoki
Carl A. Singer
Tamera Terrell

The following persons were on the balloting committee that approved this trial-use standard for submission to the IEEE Standards Board:

Russel J. Abbott	Dwayne L. Knirk	Helmut Sandmayr
Mario R. Barbacci	Paul Kogut	Mark Sastry
David L. Barton	Thomas M. Kurihara	Larry Saunders
M. Ben-Menachem	John B. Lane	Max J. Schindler
Dennis Brandl	William P. LaPlant, Jr.	Paul E. Schmid, III
Alfred Brenner	Robert J. Lechner	Peter C. Scott
Fletcher J. Buckley	F. C. Lira	Richard L. Scott
Steven Carter	Lung-Lung Liu	R. J. Scott-Kerr
Elliot J. Chikofsky	Dieter Look	William Sharon
Cory S. Clinger	Alfred Lowenstein	Harry M. Siegel
Peter Coad	Philip C. Marriott	Patricia A. Simonson
Francois Coallier	David E. Martin	M. Cassandra Smith
William M. Corwin	Roger J. Martin	Robert N. Sulgrove
D. Vera Edelstein	Tomoo Matsubara	Hitoshi Tadaumi
Peter L. Eirich	Masao Matsumoto	Dave Thomas
Terence Fong	Michael Merriman	Kenneth J. Thurber
Gordon Force	Diane Mularz	George D. Tice, Jr.
Carl B. Friedlander	Brian Nejme	Leonard L. Tripp
Roger U. Fujii	William H. Nicholls	Philip Trudeau
Haydon D. Gaines	David Ornstein	Jon L. Turino
Jerrold M. Grochow	Robert R. Parker	Margaret Updike
John S. Grout	James M. Perry	Annelise von Mayrhauser
Matt Harris	Ken J. Peterson	John W. Walz
Derek J. Hatley	Robert M. Poston	Carl Warren
John L. Hill	John S. Quarterman	Ronald Waxman
John W. Horch	Samuel T. Redwine, Jr.	Richard Werling
Duane Hybertson	Donald J. Reifer	Camille S. White-Partain
Michael P. Imber	Gordon D. Robinson	R. P. Whitehurst
Hal Jespersen	Jorge E. Rodriguez	Andrew F. Wilson
Paul C. Jorgensen	David F. Rose	William Wong
Singh Kanwalinder	John Salasin	Tomoaki Yamazaki
Ken Kennedy	Marilyn E. Sander	Bruce Zemlin

At the time this standard was being developed, several standards-making organizations were developing complementary standards. Many of those organizations participated in an ad hoc standards harmonization and coordination effort. The Task Force greatly appreciates the services of the Special Representatives to the Task Force from those organizations.

Thomas G. Baker	Chair, IGES/PDES Organization Software Products Committee, and Chair, CALS Software Products Committee
Hugh Davis	Member, Portable Common Tool Environment (PCTE) Technical Committee
Mike Imber	Chair, EIA, CASE Data Interchange Format (CDIF) Technical Committee
Paul Jorgensen	Member, Extended Systems Modeling Language (ESML) Technical Committee
Tom Kurihara	Member, CBEMA, X3
Hironobu Nagano	Chair, SC7/WG1 of the Information Technology Standards Commission of Japan (ITSCJ)
Richard Werling	Chair, ISO/IEC JTC1/SC7/WG1: Symbols, Charts, and Diagrams

When the IEEE Standards Board approved this trial-use standard on December 5, 1991, it had the following membership:

Marco W. Migliaro, *Chair*
Donald C. Loughry, *Vice Chair*
Andrew G. Salem, *Secretary*

Dennis Bodson
Paul L. Borrill
Clyde Camp
James M. Daly
Donald C. Fleckenstein
Jay Forster*
David F. Franklin
Ingrid Fromm
Thomas L. Hannan

Donald N. Heirman
Kenneth D. Hendrix
John W. Horch
Ben C. Johnson
Ivor N. Knight
Joseph Koepfinger*
Irving Kelodny
Michael A. Lawler
John E. May, Jr.

Lawrence V. McCall
T. Don Michael*
Lloyd A. "Pete" Morley
Stig L. Nilsson
John L. Rankine
Ronald H. Reimer
Gary S. Robinson
Terrance R. Whittemore

*Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Fernando Aldana
Satish K. Aggarwal

James Beall

Richard B. Engelman
Stanley Warshaw

Kristin M. Dittmann, *Editor*

Contents

CLAUSE	PAGE
1 Introduction.....	1
2. Reference Model for Tool-to-Organization Interconnections	5
2.1 Organizational Context for Tools	5
2.2 Role or Job Function View of a Tool.....	6
2.3 Life Cycle View of a Tool	7
2.4 Support View of a Tool.....	8
2.5 Tool-to-Organization Interconnection Standard Profile	9
3. Reference Model for Tool-to-Platform Interconnections	10
3.1 Hardware-Software Platform Context for Tools.....	10
3.2 Platforms	12
3.3 Hardware Platforms	12
3.4 Software Platforms.....	13
3.5 Tool to Platform Interconnection Standard Profile	14
4. Reference Model for Information Transfer Among Tools	16
4.1 Information Transfer Context	16
4.2 Mechanisms for Information Transfer Among Tools	16
4.3 Processes of Information Transfer: Services for Information Transfer	18
4.4 Descriptions of Information Being Transferred.....	19
4.5 Information Transferred.....	20
4.6 Tool Interconnection Standard Profile.....	23
5. Semantic Transfer Language (STL) Overview and Syntax	24
5.1 STL Goals	24
5.2 STL Sentence Form	24
5.3 STL Notation	26
5.4 STL Information Packet.....	26
5.5 STL Sentences	27
5.6 STL Language Elements.....	30
5.7 Language Integrity	31
5.8 STL Syntax Summary	33
6. STL Concepts and Meanings	33
6.1 STL Concept Organization	33
6.2 Concept Definition Conventions	41
6.3 Concept Definition Sentence Templates.....	44
6.4 STL Summary	75
7. STL Conformance and Extensibility	102
7.1 STL Interconnection Profile	103
7.2 STL Extensibility	106
8. Bibliography	107
Appendix A (informative) Entity-Relationship Diagrams of STL	110
Appendix B (informative) STL Entity-Relationship Diagram Example	121
Appendix C (informative) STL DataFlow Diagram Diagram Example.....	127
Appendix D (informative) STL ESML and State Transition Diagram Example	133

IEEE Trial-Use Standard Reference Model for Computing System Tool Interconnections

Part 1 Description of this Standard

1 Introduction

1.1 Purpose

The purpose of this standard is to establish agreements for information transfer among tools in the contexts of human organization, a computer system platform, and a software development application.

When a tool is integrated into (or transferred into) an organization, some things usually change. Tool users may need to learn new methods or skills. Company policies or standards may need to be updated. Measurements of work products may need to be reconsidered. The changes that an organization must make to adopt a new tool often cost more than the tool itself. Knowing what changes an organization must make to use a tool should make integrating the tool easier. To help integrate tools into organizations this standard provides a **reference model for tool-to-organization interconnections**.

When a tool is integrated into a hardware-software platform, it must be made to operate with the components of the platform. The tool will need a physical context—the mechanisms that enable information transfer. The physical context or environment is provided by a platform. A platform consists of computer hardware (e.g., C.P.O.S., disk drives, networks) and computer software (e.g., operating, database, human interface systems). To help integrate tools into platforms, this standard provides a **reference model for tool-to-platform interconnections**.

When tools are integrated into a family of tools, they need to exchange information. There are many different *methods* or mechanisms for implementing information exchanges, *processes* for transferring information, and *kindsof information* to be transferred. To help tools exchange information, this standard provides a **reference model for information transfer among tools**.

When information is exchanged among tools, all the tools must know the format or syntax of the information and the meaning or semantics of the information. To help the integration of tools into families, and to make the transfer of semantic information among tools easier, this standard provides a **semantic transfer language (STL)**.

1.2 Scope

The full-use standard will describe interconnections that must be considered when buying, building, testing, or using computing system tools. Computing system tools are developed for use in creating computing systems.

This trial-use standard includes interconnections for tools that are used for specifying behavioral descriptions or requirements of system and software products. This trial-use standard includes a textual language for representing behavioral descriptions.

This trial-use standard excludes structural descriptions of software product designs or implementations. The language defined in this trial-use standard does not facilitate defining the structures and interconnections among systems, modules, or programs. Structural descriptions will be considered for inclusion in the full-use standard.

This trial-use standard focuses on the semantics of information concerning software and system behavior, not on graphical representations of that information. The STL defined in this standard includes clauses that refer to graphical symbols. Those references to graphical symbols are intended to provide a way of linking instances of STL sentences to graphical symbols that represent the STL sentences.

This trial-use standard excludes graphical information such as symbol descriptions (size, shape, font, etc.), symbol positions (orientation, location in diagrams, etc.), and diagram placements (orientation, location on page, etc.). Graphical information is not important when communicating tools do not need to reproduce the same drawing; therefore, graphical information is excluded from the trial-use standard. Graphical information is considered to be very important when communicating tools need to reproduce the same drawing. Therefore, the full-use standard will be expanded to include either references to standards that address graphic information or language expansions to describe graphical information.

This standard is not intended to apply to general purpose tools such as word processing systems and graphical drawing packages.

1.3 Audience

This standard is intended to help buyers, builders, testers, and users of professional tools.

1.4 Organization of this Document

This document can be divided into three parts. Part One contains Section 1, Introduction. Part Two contains Sections 2, 3, and 4, which describe the context for information transfer. Part Three contains Sections 5, 6, and 7, which describe the STL.

- 1) **Introduction**
- 2) **Reference Tool-to-Organization Interconnections**— This section helps establish the organizational context in which a tool must work. Questions such as “Will a tool work with the organization's methodology?” are addressed in this section.
- 3) **Reference Model for Tool-to-Platform Interconnections**— This section helps establish the hardware/software architectural context in which a tool must work. Questions such as “Will a tool work with the organization's equipment and software?” are addressed in this section.
- 4) **Reference Model for Information Transfer Among Tools**— This section helps establish the transfer or mechanistic context in which a tool must work. Questions such as “How is information exchanged among tools?” are addressed in this section.
- 5) **Semantic Transfer Language (STL) Syntax**— This section helps establish the form and format of the information that is transferred among tools. Questions such as “What is the structure of the information transferred among tools?” are addressed in this section.
- 6) **STL Semantics**— This section helps establish the meaning of the information that is transferred among tools. Questions such as “Now is the meaning of the information to be exchanged among tools represented?” are addressed in this section.
- 7) **STL Application**— This section helps apply the STL for information transfers among tools. Questions such as “How is conformance to the standard defined?” are addressed in this section.

The relationships between the reference models and the STL are illustrated in Figure 1 .

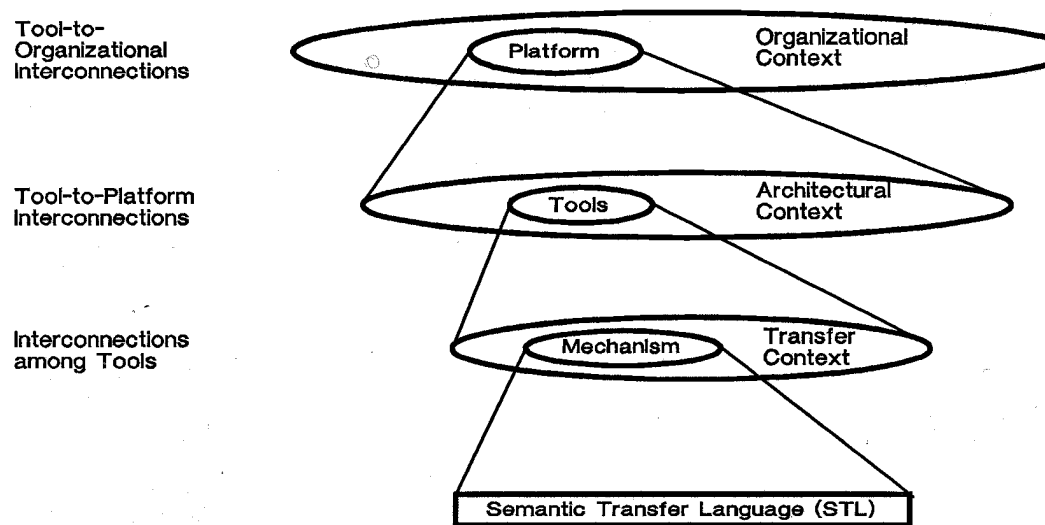


Figure 1 — Reference Models and STL Relationships

1.5 Definitions.

The definitions below describe specific terms as used within the context of the standard reference model. Additional relevant terms defined in IEEE Std 610.12-1990 [B13]¹ are as follows: **interface**, **semantics**, **software tool**, and **syntax**.

actions: Transformations of state, control, and data. See: **software concepts**.

events: Control, timing, and synchronization of actions. See: **software concepts**.

information and data: The primary object of software actions. See: **software concepts**.

logic: Logical restrictions (conditions) on actions. See: **software concepts**.

software concepts: Ideas about system and software. System and software behavior is often described by defining the associated actions, events, information, data, logic, and states of actions. These behavioral descriptors represent ideas about systems and software. Syn: See: **actions**; **events**; **information and data**; **logic**; **states**.

states: The context for and evolution of actions. See: **software concepts**.

1.6 Conformance

Conformance to the reference models described in Sections 2, 3, and 4 of this standard consists of completing the tool profiles provided at the end of those sections. The profiles serve to identify the context for tool interconnections.

¹The numbers in brackets correspond to those of the bibliography in Section 8 .

Conformance to the STL described in Sections 5 and 6 consists of completing the input and output profiles, called STL Interconnection Profiles, described in Section 7 and provided on the floppy disk that accompanies this standard. The STL Interconnection Profiles are considered to be “semantic fingerprints” because they identify every facet of the STL languages a tool uses (input profile) or produces (output profile). The STL Interconnection Profiles of different tools can be compared to determine if the tools share the same semantic intent for information and whether or not they can transfer information without conversion effort.

Conformance to this standard does not include validation that a tool does or does not process information as indicated in STL profiles. Specifications for conformance testing are beyond the scope of this standard.

Part 2

Context for Tool Interconnections

In natural languages such as French or English, each word has many meanings. In these languages the reader or receiver of communications selects one of the possible meanings based upon the context of the word. For example, consider the meaning of the word *hot* in the following sentence: “That car is hot.” *Hot* has a different meaning when the sentence is used in a desert radiator repair shop than it does when used in a beachside night club.

Professionals building computer software and systems also select word meanings based on the context in which a word or phrase is used. The term *real-time* implies seconds or tenths-of-a-second to a transaction-oriented banking system, but it implies nanoseconds to a missile range safety system. The term *process* implies different control issues to a chemical plant control system than it does to a payroll system. The term has radically different meanings for people building advanced computer architectures, code-coverage measurement systems, and sales order processing systems.

Communications among tools are also context-sensitive. The term *complete specification* has one meaning for an organization with an independent quality assurance team and a different meaning for an organization without a quality assurance team. The term *portable* has a totally different meaning in a shop using platforms (computers and operating systems) from only one product line model than it does in a shop using many different product lines. The term *transfer* means something different to tools sharing a common database (repository) than it does to tools exporting and importing files to share data.

Sections 2, 3, and 4 of this standard are included to help people working to interconnect tools identify the context for tool communications. Section 2 identifies an organizational context for a tool. Section 3 identifies the platform context for tool use. Section 4 identifies a transfer context for a tool.

2. Reference Model for Tool-to-Organization Interconnections

2.1 Organizational Context for Tools

The maximum effectiveness will be obtained from a tool when the tool works within the context of the organization where the tool is used. If a tool does not work within the context of the organization, then either (1) the tool will not be used, (2) the tool will be used in a less than optimal manner, (3) the tool will be modified, or (4) the organization will be modified. If a change to either the tool or the organization is required, then the amount of time and effort needed to make the change represents a cost that should be evaluated before the change is made. On the other hand, if a tool can work within the context of an existing organization, then change will not be needed. The potential costs of such organizational changes should be considered as part of the costs of acquiring a tool.

How well a tool interconnects with an organization may be considered from the following three perspectives:

- 1) *Role or job function*: Who uses the tool? What do people use the tool for?
- 2) *Life cycle*: In which project phases will the tool be used?
- 3) *Support elements*: How is a tool made productive?

An example of the points of interconnection within these perspectives is shown in Figure 2 .

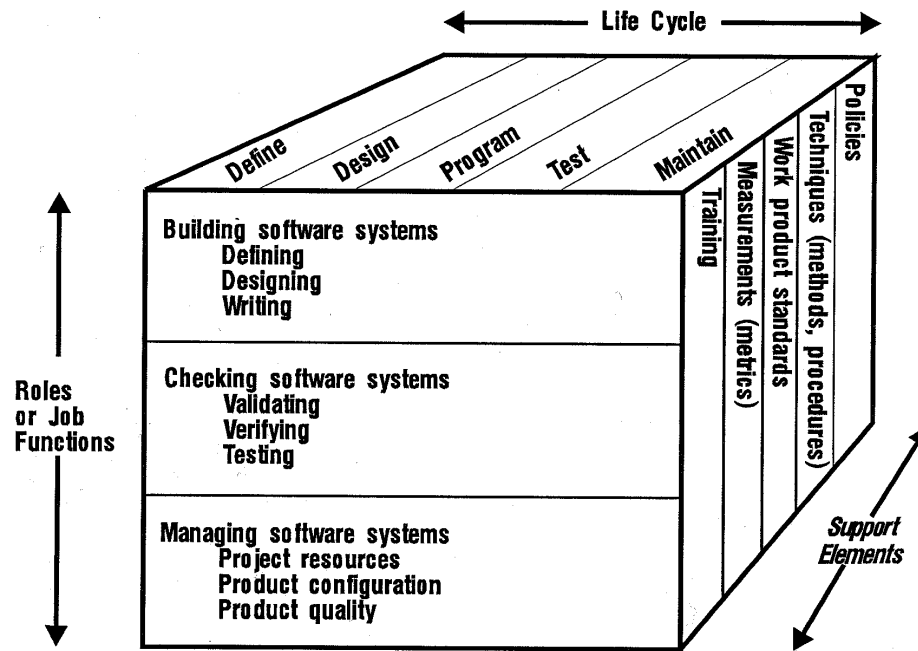


Figure 2 — Tool-to-Organization Interconnections

2.2 Role or Job Function View of a Tool

Who uses a tool? What do people use a tool for? The most direct interconnection between a tool and an organization is between the tool and its user(s). Who uses the tool can be specified by naming individuals or by naming generic roles. Roles may or may not be related to job titles or organization titles. In this standard, roles are not associated with any organization title. On a one-person project, a single individual may spend some percentage of available work time performing each of the roles named below. Each of the roles represents a job that a tool user performs. Examples of user roles and job functions are listed below. A set of software job functions is available in IEEE Std 1002-1987 [B19].

- 1) Building software systems: product engineering job functions
 - a) Defining requirements (requirements analysis)
 - b) Designing software (architectural and detailed design)
 - c) Writing programs (programming)
- 2) Checking software systems: verification, validation, and testing job functions
 - a) Validating software
 - b) Verifying software
 - c) Testing software
- 3) Managing software systems: technical management job functions
 - a) Managing project resources
 - b) Managing product configuration and change
 - c) Managing product quality

Some tools may be used for many different job functions. In those cases, the organization using the tool will probably want to identify different groups or classes of users, such as the following:

Primary user class. The people who are responsible for creating new data for entry into the tool. There is only one primary user class. *Examples:* requirements analyst, software designer.

Secondary user class. The people who reference and modify data in a tool. There are many secondary user classes. *Examples of secondary management users:* configuration control and quality control specialists. *Example of secondary technical user:* a designer or a programmer who uses the output of a requirements analysis tool. *Examples of secondary checking users:* auditors, reviewers, inspectors, and testers.

Final user class. The people who remove or retire project data from a tool. *Example:* a quality assurance specialist who extracts software metrics data from a tool, records the metrics in a corporate database, and stores project data in an off-site location.

Which job functions are defined for an organization is not the more important issue. The more important issue is that some job functions are defined and that the tool users performing those job functions are clearly identified.

2.3 Life Cycle View of a Tool

When is a tool used? A tool is directly interconnected with an organization when it is actually being used. The period of time when a tool is being used may be identified by naming the dates or by naming generic periods of time such as life cycle phases.

Examples of some life cycle phases are defined in the following list. A set of life cycle phase definitions is available in IEEE Std 1074-1991 [B24].

- 1) *Definition phase* . That period of time in a product's life cycle when the primary technical activities are concerned with defining the product's requirements.
- 2) *Design phase* . That period of time in a product's life cycle when the primary technical activities are concerned with designing the product's architecture and programs.
- 3) *Implementation phase* . That period of time in a product's life cycle when the primary technical activities are concerned with writing the programs.
- 4) *Test phase* . That period of time in a product's life cycle when the primary technical activities are concerned with testing the programs and the total software product.
- 5) *Maintenance phase* . That period of time in a product's life cycle when the primary technical activities are concerned with correcting, enhancing, and adapting the software.

Tools that are used in more than one life cycle phase may be grouped according to the following three main periods of use:

Phase of initial usage. A tool is first used when information is initially entered into it. Initial use can occur in only one product development life cycle phase.

Phases of intermediate usage. Additional uses of a tool occur when information captured by the tool is referenced or modified. These uses occur only after the initial use and may continue into a number of later life cycle phases.

Phase of final usage. Final use occurs when information is extracted from the tool for the last time on a computing system development project. Final use occurs in only one life cycle phase, usually the maintenance phase.

Which life cycle phases are defined for an organization is not the more important issue. The more important issue is that some life cycle phases are defined and that the periods of tool use are clearly identified.

2.4 Support View of a Tool

What does it take to make a tool successful in an organization? A tool's supporting infrastructure is the least obvious interconnection between an organization and a tool. The infrastructure consists of support elements. Support elements are those things that an organization must provide to maximize the benefits of a tool, as follows:

- 1) *Policies* . Policies are written descriptions of who must perform what activities in which life cycle phases. Policies may also be called directives, instructions, or methodologies.
- 2) *Techniques* . Techniques are written descriptions of how to perform an activity. Techniques may also be called methodologies, methods, or procedures. P1016.2, Guide to Software Design Descriptions, is an example.
- 3) *Work product standards* . Work product standards are written descriptions of the items (documents, code, or data) that must be produced in an activity. Work product standards may also be called documentation format standards. See the following standards as examples:
 - a) IEEE Std 830-1984, IEEE Guide for Software Requirements Specifications (ANSI)
 - b) IEEE Std 1016-1987, IEEE Recommended Practice for Software Design Descriptions (ANSI)
 - c) IEEE Std 829-1983 (Reaff 1991), IEEE Standard for Software Test Documentation (ANSI)
 - d) IEEE Std 828-1990, IEEE Standard Software Configuration Management Plans (ANSI)
 - e) IEEE Std 730-1989, IEEE Standard for Software Quality Assurance Plans (ANSI)
 - f) IEEE Std 1012-1986, IEEE Standard for Software Verification and Validation Plans (ANSI)
 - g) IEEE Std 1074-1991, IEEE Standard for Developing Software Life Cycle Processes (ANSI)
 - h) IEEE Std 1058.1-1987, IEEE Standard for Software Project Management Plans (ANSI)
- 4) *Measurements* . Measurements are written descriptions of how to evaluate work products quantitatively. (Measurements may also be called metrics.) See IEEE Std 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software (ANSI), and other ongoing standardization activities in this area.
- 5) *Training* . Training is experience in the application of support elements.
- 6) *Tools* . Tools are mechanizations that aid or replace human effort. Support elements aid a person who is new to a job or tool by providing the answers to the following questions:

Question

Answers are found in

What am I supposed to do? When do I do it?

Policies

How am I supposed to do it?

Techniques

What am I supposed to produce?

Work product standards

How will I know it is a good work product?

Measurements

Where do I get the answer to these questions?

Training

What is the easiest way to do the right thing?

Tools

(Tool administration issues such as licensing, backup/recovery, and data integrity protection are tool-specific and are not within the scope of this standard.)

2.5 Tool-to-Organization Interconnection Standard Profile

A context for designing, using, or testing a tool is established by the organization that will use the tool. By identifying each of the interconnections or relationships between an organization and its tools, knowledge of this form of context can be increased. The form entitled “Tool-to-Organization Interconnection Standard Profile,” illustrated on page 20, has been designed to help identify all of these interconnections. It is printed full size and without a figure label so it is easy to copy and use.

Tool-to-Organization Interconnection Standard Profile	
Tool Name: _____	Date: _____
Organization Interconnections	Name(s) of Applicable Standard(s)
Job Function	
Primary user	_____
Secondary user	_____

Final user	_____

Life Cycle	
Phase of initial usage	_____
Phases of intermediate usage	_____

Phase of final usage	_____

Support Elements	
Policies	_____
Techniques (methodologies)	_____

Work product standards	_____

Measurements	_____

Training courses	_____

Copyright © IEEE, 1992 — This form may be photocopied without restriction.

3. Reference Model for Tool-to-Platform Interconnections

3.1 Hardware-Software Platform Context for Tools

Just as tools are interconnected to the organizations that use them, tools are interconnected to the platforms where they operate. A platform is a collection of hardware and software components. The hardware compo-

nents include items such as computers, workstations, personal computers, terminals, disk drives, printers, and displays. The software components include items such as operating systems, database systems, communication systems, and human interface systems. A tool platform may also be called a software development environment or an operating environment.

An organization may use only one platform, and the tool will be operational on that platform. However, some organizations may use a variety of platforms and platform configurations, such as

- 1) Stand-alone work centers (workstations, personal computers, or both)
- 2) Individual work centers (such as terminals connected to a central host)
- 3) Individual work centers interconnected through communication networks

This variety presents two issues for interconnecting tools. The first is the ease of moving a tool from one platform to another platform—portability. The second is ease of using platform services to allow tools to exchange information within a platform or among platforms—connectability. How well tools interconnect with an organization's available platforms may be considered from the perspectives illustrated in Fig 3 and described in the following paragraphs.

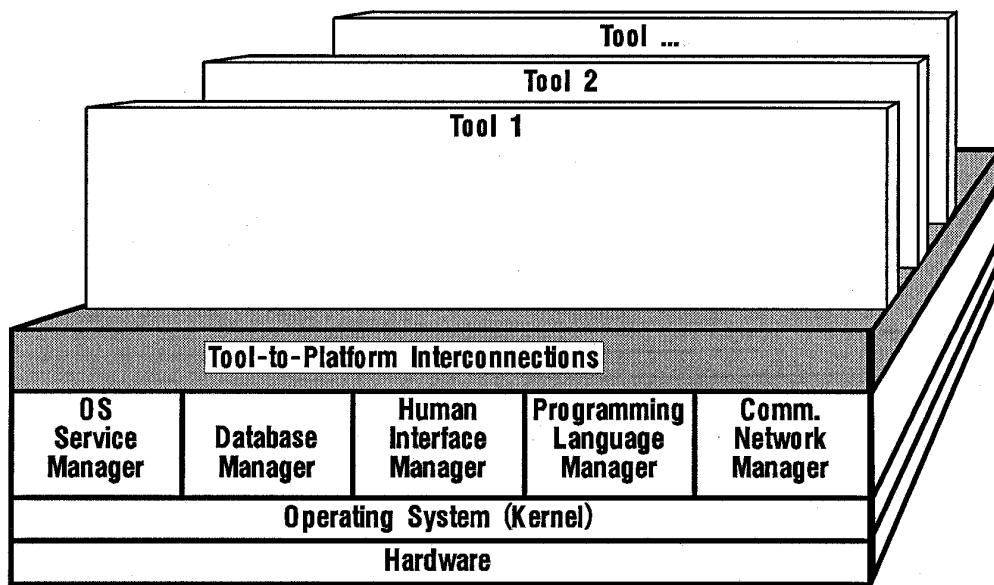


Figure 3 — Tool-to-Platform Interconnections

How well a tool interconnects with a specific platform may be considered from the perspective of how the tool uses the available platform services and controls, such as

- 1) *Operating system services:* Access, manipulate, and control hardware capabilities for all software programs on a platform
- 2) *Database system services:* Store and retrieve data, text, and graphics information for the application software programs (tools) on a platform
- 3) *Human interface system services:* Accept and display data, text, and graphical information on a platform for the human user of an application program (tool)
- 4) *Programming language systems:* Generate, control, compile, link, load, and debug programs

- 5) *Communication system services:* Send, receive, encode, decode, and route information and service requests between applications software programs (tools) running on different platforms

Tools that operate on a platform and use the services provided by the platform may also use three other types of standards to help describe or present information:

- 1) *Data file exchange system standards:* Either use standard file formats or convert from proprietary formats to standard file formats
- 2) *Document exchange system standards:* Transfer complete documents
- 3) *Description system standards:* File format for describing symbols and tables

The user's productivity and the effectiveness of a tool can be reduced if significant amounts of work are involved in moving data from a tool on one platform to a second tool on a different platform. If tools working on different platforms use special or proprietary services of the platforms, it may be especially difficult to get these tools to work together.

Attributes of a well-defined interface specification for any tool-to-platform interconnection are described in the POSIX system architecture. An interface specification should be open, independent, shared, documented, and maintained.

- 1) *Open:* The interface has
 - a) The capability of allowing a user or application operating on one platform to communicate with a user or application on another platform
 - b) Documented, nonproprietary interface specifications
 - c) Standardized connectivity and coupling for each mode of operation with implicit responsibilities to preserve interfaces
- 2) *Independent of other interfaces:* The interface supports
 - a) Software reuse, platform architecture, and application internals
 - b) Multiple releases
 - c) Recognition of authorized users
 - d) Functional recovery
- 3) *Shared:* The interface is designed for invocation by more than one application
- 4) *Documented:* The interface has
 - a) Well-defined syntax, semantics, and services
 - b) Well-defined set of parameters invoking equally well-defined sets of actions and responses
- 5) *Maintained:* The interface is updated regularly in an open consensus process

3.2 Platforms

Many platform standards need to be considered in evaluating a tool's ability to interconnect with a platform. Conversely, each platform must be evaluated to determine which standards the platform supports. The use of this standard does not require that all tools and platforms conform to the standards identified in the following sections. However, the ability to interconnect tools and freely relocate tools will probably be limited by the use of proprietary standards, tools, or platforms. A tool or platform that does not conform to the following standards is likely to require additional work for interconnections.

3.3 Hardware Platforms

There are no industry-level, consensus-based hardware platform standards. However, some organizations use only one manufacturer's product line. In effect, the product line becomes a de facto hardware platform standard.

3.4 Software Platforms

There are no industry-level, consensus-based software platform standards. The following subsections contain examples of software platform standards relevant to tool interconnections.

NOTE—For the convenience of the user of this standard, these standards are also listed in the bibliography in Section 8.

3.4.1 Operating Systems

ECMA Std 149-1990, Portable Common Tool Environment (PCTE) Abstract Specific.
ISO/IEC 9945-1:1990 (ANSI/IEEE Std 1003.1-1990), Information technology—Portable Operating System Interface (POSIX)—Part 1 : System Application Program Interface (API) [C Language] (DoD adopted).
X/OPEN Portability Guide, X/OPEN Group.

3.4.2 Database Systems

ANSI X3.133-1986, Database Language—NDL.
ANSI X3.135-1989, Information Systems—Database Language—SQL with Integrity Enhancement.
ANSI X3.138-1988, Information Systems—Information Resource Dictionary System (IRDS).

3.4.3 Human Interface Systems

X-window System, Massachusetts Institute of Technology (MIT), Version X.11.

3.4.4 Programming Language Systems

ANSI X3.9-1978 (Reaff 1989), Programming Language FORTRAN.
ANSI X3.23-1985, Programming Language COBOL.
ANSI X3.159-1989, Programming Language—C.
ANSI/MDC X11.1-1990, Information Systems Programming Language—MUMPS.
ANSI/IEEE 770X3.97-1983, IEEE Pascal Computer Programming Language (DoD adopted).
ANSI/MIL-STD-1815A-1983, Reference Manual for the Ada Programming Language.

3.4.5 Communication Systems

ISO 8571 (1988), Information processing systems—Open Systems Interconnection—File Transfer, Access and Management (Parts 1 and 2).
ISO 8649 : 1988, Information processing systems—Open Systems Interconnection—Protocol specification for Association Control Service Element—Service definition for the Association Control Service Element.
ISO 8650 : 1988, Information processing systems—Open Systems Interconnection—Protocol specification for Association Control Service Element.
ISO/IEC 8072 : 1986, Information processing systems—Open Systems Interconnection—Transport service definition.
ISO/IEC 8073 : 1988, Information processing systems—Open Systems Interconnection—Connection oriented transport protocol specification.
ISO/IEC 8802-3 : 1992 (ANSI/IEEE Std 802.3, 1992 Edition), Information technology—Local and metropolitan area networks—Part 3 : Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications.
ISO/IEC 8802-4 : 1990 (ANSI/IEEE Std 802.4-1990), Information technology—Local area networks—Part 4 : Token-passing bus access method and physical layer specifications.

ISO/IEC 8802-5 : 1992 (IEEE Std 802.5-1992), Information technology—Local and metropolitan area networks—Part 5 : Token ring access method and physical layer specifications.

3.4.6 Data File Exchange Formats

ANSI X3.122-1986, Computer Graphics Metafile for the Storage and Transfer of Picture Description Information.

ANSI X3.124-1985, Graphical Kernel System (GKS) Functional Description.

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

ISO 2022 : 1986, Information processing—8-bit single-byte coded graphic character sets.

ISO 6937 (1983), Information processing—Coded character sets for text communication (Parts 1 and 2).

ISO 8859, Information processing—8-bit single-byte coded graphic character sets (Parts 1 - 5).

ISO/IEC 8824 : 1990, Information technology—Open Systems Interconnection—Specification of Abstract Syntax Notation One (ASN.1).

ISO/IEC 8825 : 1990, Information technology—Open Systems Interconnection—Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).

3.4.7 Document Exchange Formats

ISO 8613 (1989), Information processing—Text and office systems—Office Document Architecture (ODA) and interchange format (Parts 1 - 8).

ISO 8879: 1986, Information processing—Text and office systems--Standard Generalized Markup Language (SGML).

3.4.8 Description Exchange Formats

ISO 5806 : 1986, Information processing—Specification of single-hit decision tables.

ISO 5807: 1985, Information processing—Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts.

ISO 8790: 1987, Information processing systems—Computer system configuration diagram symbols and conventions.

3.5 Tool to Platform Interconnection Standard Profile

A context for designing, using, or testing a tool is established by the platform where the tool will be used. By identifying each of the interconnections or relationships between a tool and its platform, knowledge of this form of context can be increased. The form “Tool-to-Platform Interconnection Standard Profile,” illustrated on page 20, has been designed to help identify all these interconnections. It is printed full size and without a figure label so it is easy to copy and use.

Tool-to-Platform Interconnection Standard Profile**Tool Name:** _____**Date:** _____

<u>Platform Interconnection</u>	<u>Name(s) of Applicable Standard(s)</u>
Hardware	_____ _____ _____
Operating Systems	_____ _____ _____
Database Systems	_____ _____ _____
Language Systems	_____ _____ _____
Communications Systems	_____ _____ _____
Human Interface Systems	_____ _____ _____
Data File Exchange Formats	_____ _____ _____
Document Exchange Formats	_____ _____ _____
Description Exchange Formats	_____ _____ _____

Copyright © IEEE, 1992—This form may be photocopied without restriction.

4. Reference Model for Information Transfer Among Tools

4.1 Information Transfer Context

Just as a tool-to-organization reference model helps establish the organizational context for a tool and a tool-to-platform reference model helps establish the architectural context for a tool, a reference model for information transfer among tools helps to establish the mechanistic context for information transfer among tools.

The reference model for information transfer among tools is described in the following four parts:

- 1) *Mechanisms for transferring information:* A mechanism is some combination of hardware devices and computer programs that perform the transfer of information. Reading and writing programs that process file formats are an example of a transfer mechanism. The mechanisms control how the information transfer will take place.
- 2) *Processes of transferring information:* The process of information transfer is a step-by-step procedure used by the mechanisms for transferring information. The process of transfer is dependent on the mechanisms for transferring information and the amount of error protection needed in the transfer.
- 3) *Descriptions of the information transferred:* This is a written description of what the information to be transferred looks like (its syntax) and what the information means (its semantics). The description of information transferred depends on the process of transfer and the interpretations of the information shared by the communicating tools.
- 4) *Information transferred:* This is the information that is physically moved among tools.

4.2 Mechanisms for Information Transfer Among Tools

Information transfer among tools may be performed by many different transfer mechanisms, as illustrated in Figure 4. Some implementations make these mechanisms invisible to tool users.

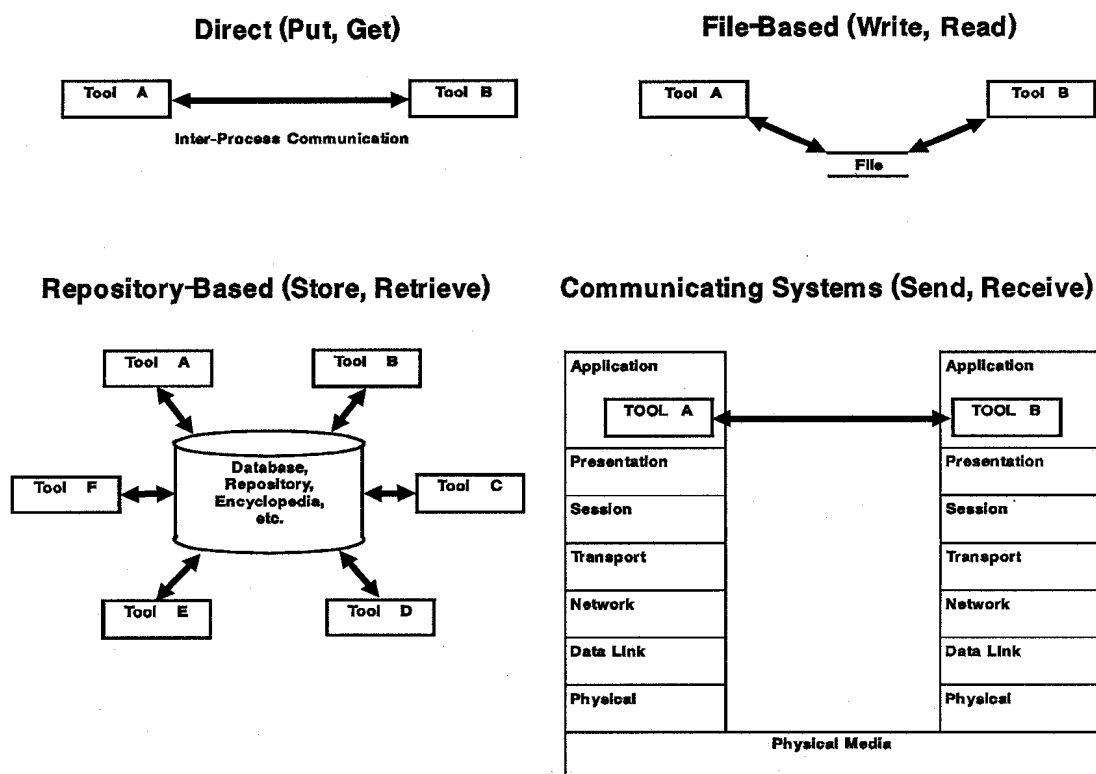


Figure 4 — Example of Information Transfer Mechanisms

Each of the mechanisms serves a particular need, and each might be selected for some particular situation. The direct transfer mechanism is the most efficient from a response time perspective. The file transfer mechanism is best from a simplicity of implementation perspective. The central database or repository mechanism is best from a tools integration perspective. An in-memory shared working area mechanism fits somewhere between direct and repository-based transfer mechanisms. The communicating system mechanism is best from an open system perspective.

Every transfer mechanism is implemented to optimize some operational characteristic. For example,

- The direct mechanism maximizes the speed of transfer.
- The file-based mechanism minimizes the cost of implementation.
- The central repository minimizes the volume of data storage.

Every mechanism optimization requires some constraints on the process of transferring information. For example,

- The speed of information transferred can be maximized if error checking is minimized.
- The volume of information transferred can be minimized if data is compressed.

Since every transfer mechanism is best for some situation and optimizes some characteristic, every mechanism will be needed at some time. Because many different mechanisms are needed, the language for transferring information defined in Section 5 of this standard has been designed to be independent of transfer mechanisms. A process-independent information transfer mechanism may be represented by the model illustrated in Figure 5.

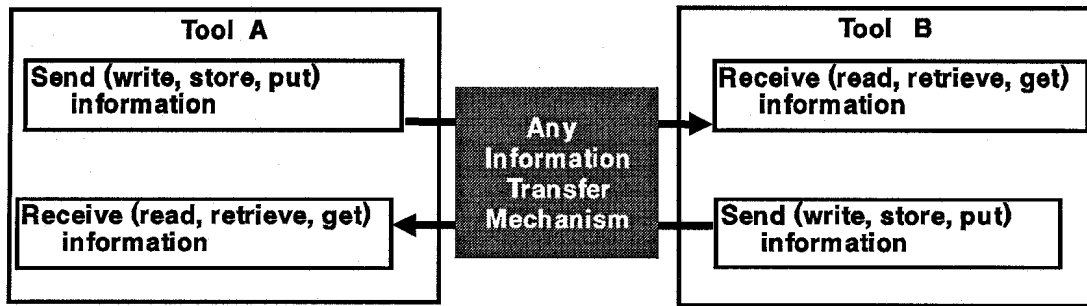


Figure 5 — Tool Interconnection Process Model

4.3 Processes of Information Transfer: Services for Information Transfer

Each transfer of information is composed of a send (write, put, or export) and a receive (read, get, or import) process. It may not be possible, however, for the makers of the sending and receiving tools to establish the details of the process before the tools are created. New tools are being announced daily. New, as yet undefined, tools will be created tomorrow. In order to accommodate this continually changing environment, the following transfer policy is endorsed by this standard:

- 1) The sending tool sends whatever information it deems necessary.
- 2) The receiving tool receives whatever information is available, verifies the information, and discards unneeded or questionable information.
- 3) At the option of the user, the receiving tool should provide diagnostic reports describing the information discarded and produce an STL file containing the discarded information.

This “receiver beware” policy is a compromise between efficient transfers and flexible transfers. The transfer could be more efficient if the quality and quantity of information were known. Unfortunately, the transfer process must accommodate transfers of unknown quantity and unknown quality.

4.3.1 Send Information

The send process can be described by defining the services it provides, as follows:

- 1) *Extract information from a tool-specific form*. In this process, the sending tool extracts information from its storage in its stored form. The design and operation of this process are affected by the information available in the sending tool and are constrained by the implementation of the sending tool.
- 2) *Encode or format the information into a standard form*. In this process, the sending tool converts the information from its internal form to the standard form. The design and operation of this process are affected by both the information from the sending tool and the standard format for information.
- 3) *Transmit the information in the standard form*. In this process, the sending tool delivers the transmitted information to an interconnection method (mechanism) such as the receiving tool, a file handler, a database system, or a communication system. The design and operation of this process are affected by both the information to be transmitted in the standard format and the transfer mechanism.

4.3.2 Receive Information

The receive function can be described by defining the services it provides, as follows:

- 1) *Recognize the smallest units of information* . In this process, the receiving tool accepts information in the standard form from the interconnection mechanism and identifies separate units of information. This process is well understood. The software for this function is commonly called a lexical scanner. If the language for information transfer is described in a standard form, available tools can generate the software for scanners.
- 2) *Recognize the structures of information* . In this process, the receiving tool recognizes collections of information. This process is well understood. The software for this function is commonly called a parser. If the language for information transfer is described in a standard form, available tools can generate the software for parsers.
- 3) *Format information from the standard form into the receiving-tool-specific form* . In this process, the receiving tool converts standard form information into its own internal (tool-specific) form.
- 4) *Map semantic information provided by the sending tool to semantic information usable by the receiving tool* . This process is not well understood. It directly depends on the agreement about the semantic interpretation of transferred information between sending and receiving tools.
- 5) *Check the information needed by the receiving tool* . Identify and resolve all conflicts between available information and needed information. Resolve all conflicts of missing, incorrect, and extra information. This process is not well understood and must be developed for each specific receiving tool.

4.4 Descriptions of Information Being Transferred

If information is to be transferred among tools and used by different tools, information must be interpreted the same way by all communicating tools. Each unit of transferred information must have its syntax (form) and its semantics (meaning) defined.

4.4.1 Syntactic (Format) Information

Syntactic information describes the physical structure or form of subject information. Syntactic information may or may not be transferred among tools, but it will be used in the sending and receiving processes.

If the communicating tools interpret syntax in one and only one way, syntax information does not need to be transferred among tools. If the communicating tools must transfer information that describes the syntax, the information is said to have self-describing syntax. ISO/IEC 8824m : 1990 [B43] and ISO/IEC 8825 : 1990 [B44] provide the standards necessary for defining information with self-describing syntax. Together ISO/IEC 8824 and ISO/IEC 8825 describe the following information:

- Atomic data elements (bits) and how those atomic data elements will be grouped for interpretation (coded) into primitive data values such as characters, character strings, and numbers.
- How primitive (unstructured) data items may be grouped together into structured data items such as words or fields.
- The delimiters (characters, symbols, or counts of characters) that separate primitive data items.
- How structured data items may be grouped together to form data items of larger structures such as lists, records, files, tables, and databases.

The syntax for information transfer described in Section 5 is designed so that it is compatible with ASN.1 (ISO/IEC 8824 and ISO/IEC 8825) but does not contain the self-describing information.

4.4.2 Semantic (Meaning) Information

Semantic information is the information that represents the meaning of transferred information. Communicating tools cannot process information the same way unless they have exactly the same interpretation of the information. They must share one, and only one, interpretation.

Communicating tools may transfer information that is syntactically correct and interpret the information in totally different ways. For example, many tools have a concept of “states.” But what is the meaning behind a “state”? Is it a “action state” or a “data state”? Both are commonly used in the industry.

- “Action state” or “functional state” both mean a collection of operations available for execution at an instant of time.
- “Data state” means a collection of data values existing at an instant of time.

Answers to these questions depend on how different tool producers interpret the different methodologies. The answers depend upon semantic information. Because there are many methodologies and their notations are not mathematically precise, different interpretations of their meanings are possible. Since different interpretations are possible, there is no way to determine which interpretation is “correct,” and tool producers are free to build tools using whatever interpretation they choose.

If different interpretations are possible and tools must transfer information with one, and only one interpretation, then

- One tool must process multiple interpretations, or
- Both tools must process multiple interpretations, and choose the right interpretation for each transfer, or
- Both tools must be able to transform their information to a common interpretation.

The STL presented in Sections 5 and 6 has been designed so that any tool can map or transform its own interpretations to a standard interpretation, and thereby establish meaningful interconnections with other tools.

4.5 Information Transferred

Transferred information may be classified as control, management, or subject information.

4.5.1 Control Information

Control information is the information that allows the communicating tools to start, stop, send, receive, and resend information. Differences in control information and the control of information transfer are what separates methods of transfer such as direct transfer systems (puts, gets), repository-based systems (writes, reads), file-based systems (writes, reads), communicating systems (sends, receives). Control information is typically defined by platform standards (operating, database, and/or communication system) and increasingly, with tools and applications. Control information will not be included in this document.

4.5.2 Management Information

Management information is information that helps support subject information. Each instance of information about a concept needs to be managed and controlled. If management and control information is transferred via a tool interconnection, the information should be in a standard format.

4.5.2.1 Quality Information

What information does the tool transfer for evaluating the quality of the hardware or software being developed? An example of this kind of information would be the data structure complexity for a data item such as a database.

4.5.2.2 Configuration and Change Information

What information does the tool transfer for controlling the hardware or software being developed? An example of this kind of information would be the version number of an instance of a concept or the release number for a software program. See IEEE Std 828-1990 [B15].

4.5.2.3 Project Management Information

What information does the tool transfer for controlling resources allocated to developing the hardware or software? An example of this kind of information would be the date on which an instance of the concept was approved. See IEEE Std 1058.1-1987 [B23].

4.5.2.4 Measurement (Metrics) Information

Measurement information is used to quantify products and processes for all types of management including quality, configuration, and project management. An example of this kind of information would be the count of data items included in a data dictionary. See 2.4, item 4 of this standard for a list of standards relating to metrics.

4.5.3 Subject Information

Subject information is the actual information being transferred. Subject information includes available information about software requirements, designs, programs, and tests. Subject information is described in terms of software concepts such as actions (e.g., transactions, processes, etc.), data (e.g., information, data relationships, etc.), events (e.g., times, interrupts, etc.), constraints (e.g., restrictions, limits, etc.), and states (e.g., context, situations, etc.). At any point in time, most developers and tools view subject information from one viewpoint or perspective, such as data flow, control flow, or performance. Once a perspective is selected, the information can be presented in either graphical, tabular, or natural language, or in the semantic transfer language.

The separation of presentation, perspective, and concept information allows a tool user to work with whatever perspective and presentation are most appropriate for a job. The separation also simplifies transfer of information among tools. For example, tools that provide their own presentation forms need to receive only concept and perspective information. The separation is illustrated in Figure 6.

- *Presentation information:* Presentation information is information about how perspective information is presented on screens, on paper, or in files. Presentation forms for perspective information may be for graphical, tabular, or natural language, or for the semantic transfer language. For example, graphical presentation information describes
 - Symbols, such as letters, numbers, circles, boxes, arrows, icons
 - Symbol characteristics (attributes, properties) such as color, size, intensity
 - Symbol organizations or layouts, such as displays, pages, diagrams, figures
 - Symbol location or placement in a presentation organization, such as point (x, y)
- *Perspective information:* Perspective information is an organization of a subset of the software conceptual information. Perspective information is selected to satisfy a particular need, such as a need to analyze a state transition view, an entity-relationship view, a data flow view, a performance view, etc.
- *Conceptual information:* Software behavior is often described by defining the software's

- *Actions* , including transformations of state, control, and data
- *Information and data* , the object of software actions
- *Events* , control timing and synchronization of actions
- *Logic* , the conditions, constraints on actions
- *States* , the evolution of and context for actions

These behavioral descriptors represent ideas about systems and software, and are called behavioral concepts, software concepts, or simply “concepts.”

Relationship information is a different kind of information. Relationship information is information about how different concepts are related to one other, for a particular instance of the software.

- Software concepts are related to other software concepts, such as “Action X has component actions Y and Z.”
- Software concepts are related to representation concepts, such as “Action X is pictured with graphic symbol circle2.”

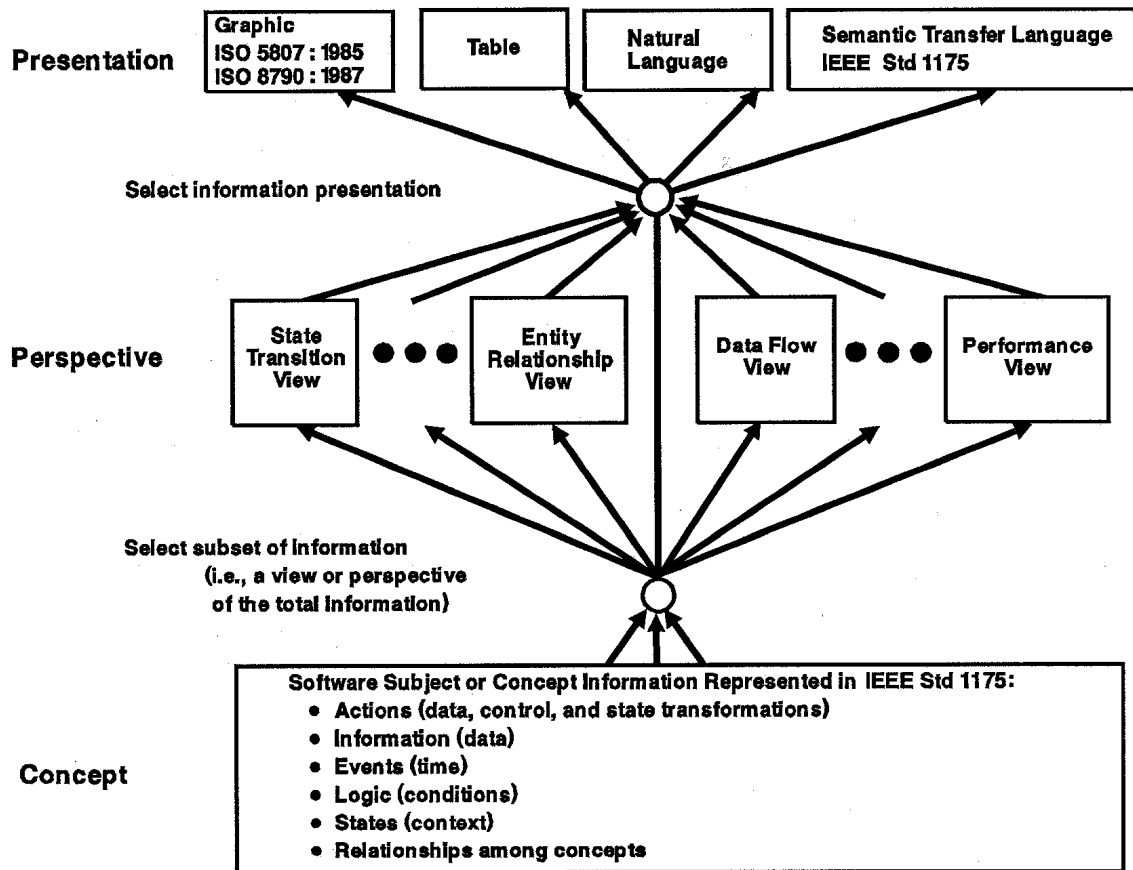


Figure 6 — Presentations, Perspectives, and Concepts

4.6 Tool Interconnection Standard Profile

A context for transferring information among tools is established by several items: (1) the mechanisms that perform the transfer, (2) the process of the transfer, and (3) the descriptions of the syntax and semantics for information to be transferred. Identifying each of these items allows knowledge of the transfer context to be increased. The form entitled "Tool Interconnection Standard Profile," illustrated on page 29, has been designed to help identify all these interconnections. It has been printed full size and without a figure label so that it is easy to copy and use.

Tool Interconnection Standard Profile	
Tool Name: _____	Date: _____
<u>Tool Interconnections</u>	<u>Name(s) of Applicable Standard(s)</u>
Mechanisms for Transfer	
Direct	_____
File-Based	_____
Central Repository	_____
Communications Systems	_____
Other	_____
Processes of Transfer	
Send	_____
Receive	_____
Information Descriptions	
Syntax	_____
Semantics	_____
Information Purpose	
Control	_____
Management	_____
Quality	_____
Configuration/Change	_____
Project Management	_____
Measurement (Metrics)	_____
Subject	_____
Presentation	_____
Perspective	_____
Concept	_____
Other	_____

Copyright © IEEE, 1992—This form may be photocopied without restriction.

Part 3 Interconnection Language

In Part 1 of this standard, this standard was described. In Part 2 the context for the transfer of information among tools was identified. Part 3 provides a description of a language that enables information transfer among tools. Section 5 provides a description of the syntax for language. Section 6 provides a description of the semantics for the language and templates for the sentences of the language. Section 7 provides a description of how to describe conformance to the language and how to extend the language.

5. Semantic Transfer Language (STL) Overview and Syntax

This section contains an overview of the STL and a summary of its syntactic elements. The STL is presented in top-down order. First, a complete information transfer is described. Next, the language structures are described. Finally, the language primitives are defined.

5.1 STL Goals

Design decisions for the STL were based on satisfying the following goals, in order of decreasing priority:

- 1) The STL must be parsable. This goal is the minimum requirement for allowing tool interconnection. If information is to be transferred among tools, the tools must be able to read the information. This absolute requirement does not require that the STL be easy to parse, only that it be parsable.
- 2) The STL must be easy to read by programmers who do not have special training in the STL and do not have special tools. The goal is to express information about software in a form that requires the least amount of specialized knowledge to read. In particular, a software description written in the STL should be viewable without needing a special tool. A text editor should be sufficient for viewing an STL file.
- 3) The STL must be easy to write. Programmers who have limited training in STL and no access to special tools must be able to write a description in STL easily. A text editor should be sufficient for preparing an STL file.
- 4) It should be possible to convert the STL into a compact transfer form to make efficient use of machine resources.

Goals 2 and 3 imply that the STL should be a language that is close to natural language. Goal 4 implies that the STL should be a sparse, machine-like language. To meet these diverse goals, the STL is provided in a human-readable, highly stylized sentence form.

5.2 STL Sentence Form

Figure 7 contains a sequence of special natural language sentences. They are special because

- 1) Every sentence is about one, and only one, subject (i.e., Action A01).
- 2) Every sentence contains a noncompound verb or verb phrase.
- 3) Every sentence expresses only one relation to or attribute of the subject.
- 4) Every sentence has the same order of words:
 - a) a subject with a descriptor
 - b) a verb or verb phrase
 - c) a direct object with a descriptor or a prepositional or an adverbial phrase

Figure 8 contains an STL sentence that contains the same information as the natural language sentences presented in Figure 7.

Action A01 is allowed in state normal.
Action A01 receives eventitem warning_interrupt.
Action A01 transmits eventitem safety_action1.
Action A01 uses dataitem water temperature reading.
Action A01 produces dataitem safety_action_report.
Action A01 has duration time maximum 3.
Action A01 has duration time units "second".

Figure 7 — Natural Language Sentences

Action A01
is allowed in state normal ;
receives eventitem warning_interrupt ;
transmits eventitem safety_action1 ;
uses dataitem water temperature reading ;
produces dataitem safety_action_report ;
has duration time maximum 3 ;
has duration time units "second".

Figure 8 — STL Sentence Example 1

The STL sentence is different from the natural language sentences in two ways:

- 1) All the natural language sentences about one subject are merged into one sentence. Several simple sentences have become several simple clauses in one sentence.
- 2) Where the sentences are separated by periods, the clauses are separated by semi-colons.

The STL sentence is presented in Figure 8 with one clause per line. However, the STL language has a free form, so any number of words can be placed on a line. The convention of placing one clause on a line is used, because it focuses attention on one clause (either a relation or attribute) at a time. The example STL sentence in Figure 8 is used in the rest of this overview.

5.3 STL Notation

STL syntax is described using a modified version of the Backus-Naur Form (BNF).² The BNF symbols and their meaning are defined in Table 1. Sections 5.5 through 5.7 provide a narrative description of the BNF syntax for the STL. The complete BNF for STL is presented in Figure 12.

Table 1 — BNF Notation Symbols

Symbol	Meaning
::=	is defined as
white spaces	spaces, blank lines, or tabs between terms (White space means “is followed by”.)
x y	choice — either x or y
{x}	0 or more repetitions of x
n{x}m	at least n and as many as m repetitions of x
[x]	optional — 0 or 1 occurrences of x
<x>	x must be replaced by other language constructs: x is not a terminal language element.
x	x cannot be replaced by other language elements: x is a terminal language element.
NULL	the empty expression
(text)	The text is a comment and not part of the syntax.

5.4 STL Information Packet

The information content of an information transfer shall be called a software description packet, or simply S_Packet. Many S_Packets may describe the same system, hardware, or software product, but each S_Packet contains information about one, and only one, system, hardware, or software product. The form and rules for interpreting S_Packets are described in the following paragraphs. In BNF the S_Packet syntax is described as follows:

<S_Packet> ::= <identification_sentence>
 {<STL_sentence>}
 pc_mark

The BNF definition is read as follows: an S Packet is defined as an identification_sentence followed by any number of STL_sentences followed by the seven characters pc_mark. The S Packet identification sentence

²Naur, P. (ed.) “Revised Report on the Algorithmic Language Algol 60,” *Communications of the ACM*, vol. 6, no. 1, Jan. 1960, pp. 1–17.

is a special STL_sentence intended to support the configuration management of information transfer using STL. It is described in 6.3.19. The characters pe_mark are the “packet end mark” and must be the last characters in the packet. The STL_sentences may be in any order.

5.5 STL Sentences

All STL_sentences are constructed with the same rules. STL_sentences are composed of other language elements: words, clauses, and punctuation. STL_sentences have a restricted vocabulary, a small number of well-defined clauses, and a few punctuation marks. An STL sentence in BNF syntax is

```
<STL_sentence> ::=
    <sentence_keyword> <sentence_identifier>
    <STL_clause> { ; <STL_clause> }.
```

The BNF definition is read as follows: an STL_sentence is defined as a sentence_keyword, followed by a sentence_identifier, followed by one or more STL_clauses, each of which is separated by a semicolon, and with the final clause followed by a period. The STL_clauses may be in any order.

In the example STL sentence presented in Figure 8 , the sentence_keyword is “Action,” which is a noun. The sentence_identifier, A01, is an appositive that makes the noun specific. The example STL sentence contains seven clauses and is terminated with a period.

5.5.1 Sentence_keywords

A sentence_keyword must be the first word in every STL_sentence. Sentence_keywords identify the software concept being described in the STL_sentence. The BNF syntax for a sentence_keyword is

```
<sentence_keyword> ::= <concept_name>
```

The BNF definition is read as follows: a sentence_keyword is defined as a concept_name. Concept_names are itemized in Figure 9. An STL_sentence is defined in Section 6 for every concept named in Figure 9 . Additionally, concept_names may be defined through the use of the STL extensibility features as described in 7.2 . Stylistically, the forms in Figure 9 are preferred, but STL processing shall be case-insensitive.

Action	DataKey	EventItem
Collection	DataPart	EventType
Condition	DataRole	GraphicSymbol
ConnectionPath	DataStore	Object
Constant	DataType	S_Packet
DataItem	DataView	State
		StateTransition

Figure 9 — STL Concept Names

5.5.2 Sentence_identifier

In each STL_sentence, the sentence_identifier is a symbolic term used as the unique identifier of the sentence. Sentence_identifiers may be entered in a tool by a tool user or may be created by the tool. Ensuring uniqueness is a responsibility of the S_Packet creator.

The sentence_identifier may be used in two ways. First, it may be used to identify the sentence where a concept is being defined, as in the “A01” expression in Figure 10. Second, it may be used as a reference to another sentence, as in the expression “normal” in Figure 10.

The sentence_identifier is a language element shown in the BNF as

`<sentence_identifier> ::= <symbolic term> (See 5.6.2 )`

5.5.3 STL_Clauses

In Figure 10 the first clause in the STL example sentence describes a relation between Action A01 and state normal. It is a relation_clause. Each of the last two clauses in the example sentence describes one measurable fact or attribute of Action A01. The last two clauses are attribute clauses. Each STL_clause describes either one relationship for or one attribute of the subject of the sentence where the clause appears. The BNF syntax for an STL_clause is described as follows:

`<STL_clause> ::= <relation_clause> | <attribute_clause> | NULL`

Clauses may or may not be present. When clauses are present, they may appear in any order. In any STL_sentence a particular clause may be present once and only once. STL_clauses must begin with a keyphrase and are separated by a semicolon. The last clause in a sentence is properly followed by a period, not a semicolon. However, for the convenience of STL users, the construct “;.” is allowed at the end of a sentence by the NULL alternative above. The NULL alternative stands for no characters.

Action A01

is allowed in state normal ;

receives eventitem warning_interrupt ;

transmits eventitem safety_action1 ;

uses dataitem water temperature reading ;

produces dataitem safety_action_report ;

has duration time maximum 3 ;

has duration time units "second".

Figure 10 — STL Sentence Example 2

Action A01

is allowed in state normal ;
receives eventitem warning_interrupt ;
transmits eventitem safety_action1 ;
uses dataitem water_temperature_reading ;
produces dataitem safety_action_report ;
has duration time maximum 3 ;
has duration time units "second".

Figure 11 — STL Sentence Example 3**5.5.3.1 Relation_clauses**

A relation_clause defines a relationship between the system or software concept instance being defined in the STL_sentence and one or more concept instances defined in other STL_sentences. Relation_clauses may describe relations such as abstraction, aggregation, connection, presentation, and restriction. The set of relation_clauses is unique for each pair of software concepts, so the relation_clauses are defined with the STL_sentences in Section 6. The BNF form of the relation clause is as follows:

```

<relation_clause> ::=
    <relation_keywords>
    <relation_list>

```

The BNF sentence is read as follows: a relation clause is defined as relation_keywords followed by a relation_list. In Figure 11 the example STL sentence is presented with relation_keywords underlined and relation_lists boldfaced.

The last word in each set of relation_keywords is the name of a concept whose instances can appear in a relation_list. This word functions as a classifier and allows type checking within the clause. When the concept name appears as a classifier it is expected to be in lowercase letters as illustrated in Figure 11. The BNF form of the relation_clause is as follows:

```

<relation_keyword> ::= <relation_terms><relation_classifier>
<relation_classifier> ::= <concept_name>

```

The relation_list references (names) the concept instances that participate in the relationship. The BNF form of the relation_list is as follows:

```

<relation_list> ::= <relation_member> { , <relation_member> }
<relation_member> ::= <sentence_identifier> | TBD

```

A relation_list contains one or more relation members. A relation_member may be either a sentence_identifier or the special keyword “TBD.” “TBD” stands for “to be determined” and is used to enable the transfer of specifications that are known to be incomplete at their time of transfer. If the relation_list contains more than one sentence_identifier, the identifiers are separated by commas. The last or only sentence_identifier is not followed by a comma. Every sentence_identifier named (referenced) in a

relation list must be defined in an STL_sentence that is included in the S_Packet. This is the responsibility of the S_Packet creator.

Most relations can be expressed in two complementary forms. For example,

```
Dataitem    x    is used by action    y    .
Action      y    uses dataitem        x    .
```

The relation clauses “is used by” and “uses” express the input relation between x and y. To conserve storage, some tools will store the relationship in one direction and then derive the other direction when it is needed. In the STL, relationships may be expressed in either or both directions. Relation clauses and keywords are defined with the STL sentences in Section 6.

5.5.3.2 Attribute_clauses

The last two clauses in the example STL_sentence of Figure 10 are attribute_clauses. Attribute_clauses do not contain concept names. Instead, attribute_clauses specify intrinsic properties or measurable characteristics of the software concept instance being defined in an STL_sentence. The BNF syntax for an attribute_clause is as follows:

```
<attribute_clause> ::= <attribute_keywords> <attribute_value>
```

That is, an attribute_clause is defined as attribute_keywords followed by an attribute_value. The attribute_keywords identify what type of attribute is being defined. Attribute_keywords are defined with the STL_sentences later in Section 6. When attribute_keywords are present, all terms of the attribute_clause must be present and in order.

An attribute_value is one or more literals or constants. In the last clause of Figure 10, the word “second” is in quotes to indicate a string value, while the number “3” in the penultimate clause is not in quotes to indicate a numeric value. In some cases, specific keywords are provided for the possible attribute_values. Attribute_values are defined with the STL_sentences in Section 6.

5.6 STL Language Elements

Attribute_values, keywords, sentence_identifiers, periods, and semicolons are examples of the basic language elements of the STL. They are called tokens and are composed of one or more characters. Characters are the most primitive language elements of the STL. In this section the characters and tokens of the STL are defined.

5.6.1 Character Set

Character set encoding and decoding are functions of the sending and receiving tools, networks, or system utilities and are not described in this standard. Any standard character set encoding that supports the characters itemized below is acceptable for use with the STL.

character. For the STL, the character set is divided into white_space, digits, letters, special_characters, and other_characters, as follows:

```
<character> ::= <white_space> | <letter> | <digit>
               | <special_character> | <other_character>
```

white_space. White_space characters are characters that are printable as blank space, as follows:

```
<white_space> ::= <tab> | <new line> | <carriage return> | <space>
```

letter. The letter characters are lowercase alphabetic characters, a through z, and the uppercase alphabetic characters, A through Z, as follows:

<letter> ::= a-z | A-Z

digit. The numerical characters are the digits 0 through 9, as follows:

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

special_character. The special_characters are the STL punctuation characters, as follows:

<special_character> ::= , | . | ; | _ | " | + | - | (|) | / | *

other_character. No other characters are recognized in the STL. However, any characters may be transferred in an S_Packet as strings (see Section 6).

5.6.2 Token

A token is a special_character or a group of characters. The allowed groups of characters are called character strings, numbers, keywords, keyphrases, symbolic terms, or comments. Tokens are as follows:

**special_character | string | number | keyword | keyphrase | symbolic_term
| comment**

Strings, numbers, keywords, and keyphrases are described in Section 6. Symbolic_terms and comments are described in the following sections.

5.6.3 Symbolic Terms

User-created terms, such as sentence_identifiers (e.g., “A01” and “normal” in Fig 10), are called symbolic terms. In the STL, symbolic terms uniquely identify every occurrence of an STL_sentence. Symbolic terms may contain any of the letters, A-Z and a-z, the connective underscore “_”, and the digits 0-9. Symbolic terms shall begin with a letter. They shall not exceed 64 characters in length.

5.6.4 Comments

Comments may be used freely in a description. A comment begins with the two characters “/*” and continues to the first occurrence of the two characters “*/”. All contents of a comment are uninterpreted semantically. Comments are tokens that are ignored. Comments are as follows:

comment ::= /* {<character>} */

5.7 Language Integrity

An STL S_Packet is “correct” if it is unambiguous, consistent, and complete, as defined in the following paragraphs.

5.7.1 Unambiguous

An instance of a software concept is unambiguous if it has one, and only one, defining sentence. All concept instances shall have unique sentence_identifiers to prevent misunderstandings of the description. Each instance of a software concept can be defined in one, and only one, sentence. When two or more sentences for the same instance are encountered, the policy for handling the duplications shall be as follows:

- 1) Keep the first one.
- 2) Ignore subsequent ones.
- 3) Advise the user of the duplication.

5.7.2 Consistent

Consistency of software concept information can be evaluated from different levels of S_Packet and sentence. In addition, complementary relationships, such as “carries” and “is carried by,” require special care in dealing with an S_Packet.

s_packet consistency. An S_Packet is said to be consistent if each term in the S_Packet is used in one and only one way. For example, if a data item has the sentence_identifier X, then a data state cannot have the sentence_identifier X.

sentence consistency. An STL_sentence is said to be consistent if every concept name mentioned in a relation_list is of the same type as identified by the *classifier* in the sentence.

complementary relationships. Most relationships may be stated in either of two forms. For example, a definition sentence for data item X may have the clause stating it “is carried by” connection path Z, while the definition sentence for connection path Z may have the clause stating it “carries” data item X. The presence of either clause in its appropriate definition sentence records the relationship. It is also acceptable for both clauses to be present in their appropriate definition sentences, but they must both record the same relationship. If they are both present but indicate different relationships, the software description is inconsistent. This rule allows easy expression of many-to-many relations.

5.7.3 Complete

Completeness of software concept information can be evaluated from different levels of S_Packet, sentence, and clause. A fourth type or level of completeness, information completeness, cannot be evaluated using this standard.

5.7.3.1 S_Packet Completeness

An S_Packet is complete if two conditions are satisfied. First, the S_Packet must begin with an identification_sentence and end with the “pe_mark” keyword. Second, every sentence_identifier named (referenced) in an STL_clause must be defined in an STL_sentence. The first condition requires that text boundaries are identified. The second condition requires that referential integrity is satisfied.

5.7.3.2 Sentence Completeness

An STL_sentence is complete if it has a sentence_keyword, a sentence_identifier, at least one STL_clause, and a period. Sentence completeness does not imply sufficiency for the receiver of the S_packet.

5.7.3.3 Clause Completeness

A clause is complete if it has every required language element.

5.7.3.4 S_Packet Completeness versus Model Completeness

S_packet completeness and model completeness are two different subjects. For example, a state transition *model* cannot be called complete unless it includes at least one initial state and at least one final state. Yet a tool user may want to transfer one day's work such as an incomplete state transition model from one tool to another. To accommodate this everyday need of a tool user, the S_packet or transfer must allow for transferring incomplete models. S_Packet completeness is defined in 5.7.3.1. The subject of information or model completeness is beyond the scope of this standard.

5.8 STL Syntax Summary

The form and punctuation for the STL in BNF notation is presented in Figure 12 .

<code><S_Packet> ::=</code>	<code><identification_sentence></code> <code>{ <STL_sentence> }</code> <code>pe_mark</code>
<code><identification_sentence> ::=</code>	(See 6.3.19.)
<code><STL_sentence> ::=</code>	<code><sentence_keyword> <sentence_identifier></code> <code><STL_clause> { ; <STL_clause> } .</code>
<code><sentence_keyword> ::=</code>	<code><concept_name></code>
<code><concept_name> ::=</code>	<code><symbolic term></code> (See Fig 9 and Sections 6.3 and 7.2.)
<code><sentence_identifier> ::=</code>	<code><symbolic term></code> (User defined, see 5.6.3.)
<code><STL_clause> ::=</code>	<code><relation_clause> <attribute_clause> NULL</code>
<code><relation_clause> ::=</code>	<code><relation_keywords> <relation_list></code>
<code><relation_keywords> ::=</code>	<code><relation_terms> <relation_classifier></code>
<code><relation_terms> ::=</code>	(See Section 6, Table 5 – center column.)
<code><relation_classifier> ::=</code>	<code><concept_name></code>
<code><relation_list> ::=</code>	<code><relation_member> { , <relation_member> }</code>
<code><relation_member> ::=</code>	<code><sentence_identifier> TBD</code> (The special keyword TBD means “to be determined” and is used to indicate that a sentence_identifier is unknown, needed, and should be defined at a later date.)
<code><attribute_clause> ::=</code>	<code><attribute_keywords> <attribute_value></code>
<code><attribute_keywords> ::=</code>	(See Section 6.)
<code><attribute_value> ::=</code>	(See Section 6.)
<code><symbolic_term> ::=</code>	(See 5.6.3.)

Figure 12 — STL Syntax

6. STL Concepts and Meanings

This section is organized as follows: 6.1 and 6.2 form an introductory overview; 6.3 is in the form of a reference handbook, with each STL concept presented in alphabetical order.

6.1 STL Concept Organization

STL concepts are logically organized according to subject, purpose, and type. This logical organization gives a classification hierarchy with three levels. The classification shows the commonality that exists among the concepts and the differences that distinguish among them.

The classification is exhaustive, and the distinctions make them mutually exclusive as well. Thus, each concept fills a well-defined information transfer need. Table 2 illustrates the classification hierarchy.

Table 2 — Classification Hierarchy for Concepts

Category, Group, Concept	Subject, Purpose, Type of Information
Identification Category	Subject: information transfer unit
Identification Group	Purpose: Identification of Information
S_Packet	-information labeling and references
Interpretive Support Category	Subject: elements of description
Value Group	Purpose: value instance representations
literals	-quantification of properties
Constant	-reference names for literals
Organization Group	Purpose: user focus
Collection	- groupings of software description elements
Object	- combine and encapsulate properties and transforms
Image Group	Purpose: visual presentations indrawings
GraphicSymbol	- references to named image instances
Software Behavior Category	Subject: operations of software
Data Group	Purpose: property representation
DataItem	- labeled instances of datatypes in transforms
DataKey	- identifier property representation
DataType	- property representation abstractions
DataPart	- property component structuring
DataRole	- rules for interaction instance
MapView	- entity subtype structure
Time Group	Purpose: marking of time
EventItem	-labeled appearances of recognized time
EventType	- time recognition abstractions
Logic Group	Purpose: decisions
condexpressions	- property value instance discriminator
Condition	- named property value instance discriminators
Transform Group	Purpose: changes of property representation
Action	- observable changes of properties
DataStore	- retention of properties between actions
Coupling Group	Purpose: propagation of changes
ConnectionPath	- information propagator to transforms
Evolution Group	Purpose: sequencing of changes
State	- property characterization and transform access
StateTransition	- allowed changes of state

This classification is described in the remainder of this section. Appendix A contains entity-relationship dia-

grams for the STL that illustrate the interrelationships between the concepts. The set of diagrams is organized by subtypes and interrelationships.

Each STL concept instance has a subject about which it provides information. Three categories of STL concepts are distinguished:

- The identification category
- The interpretive support category
- The software or system behavior category

The identification category contains a concept whose instances provide facts about the S_Packet itself. These facts must be known for the S_Packet to be used. Examples are the subject of the S_Packet and the date of S_Packet creation.

The support category contains concepts whose instances provide facts about the quantification, organization, and presentation of other concept instances in an S_Packet. Instances of concepts in the support category provide no direct facts about the software behavior being described. They facilitate the human interpretation and application of the information provided by other concept instances.

The behavior category contains the concepts whose instances provide facts about the behavior of the subject software or system being represented with an STL description. These are the core concepts of the STL. They are used to express the specific observable characteristics of the software in its application. From the instances of these concepts in an S_Packet, various models may be abstracted and various diagrams may be made.

The following sections describe the organization of concepts within each of these categories.

6.1.1 Identification Category

There is only one purpose for information in the identification category, namely to identify the information in an S_Packet. Only one concept sentence, the S_Packet Identification sentence, is defined for this purpose. Only one instance of this sentence appears in an S_Packet. The sentence keyword for this sentence must be "S_Packet".

6.1.2 Interpretive Support Category

The interpretive support category is divided into three groups that serve to quantify, scope, or map the software behavior concepts.

- 1) *Quantifying concept instances.* All concept instances in an S_Packet are quantified by value concept instances. The STL provides two value concepts. They are the literal and the Constant. There is no concept definition sentence for literals. Several subtypes of value concepts are provided in STL that, together with the STL data concepts, are used to quantify application-specific facts.
- 2) *Scoping intellectual viewpoints on the concept instances.* Much information is needed to specify adequately the required behavior of modern software. Many different approaches have been developed to focus on selected parts of the information. These approaches normally adopt a particular viewpoint for selecting the information to consider. They also include hierarchical partitioning of the selected information into intellectually manageable chunks that are easier to reason about. The STL provides two organization concepts that are used to express these subsets and their hierarchical structures. They are the Collection and the Object concepts.
- 3) *Mapping concept instances to diagram images.* Many current organizational approaches also have diagrammatic representations for the information in the hierarchical structures. The STL does not include transfer of graphical information directly. It does have one additional concept for mapping

between software category concept instances and instances of externally defined image type instances. That concept is the GraphicSymbol.

6.1.2.1 Quantifying: Value Concepts

The STL has two concepts for values: the literal and the Constant.

For the literal, there is no concept definition sentence. The definition of a literal instance is neither more nor less than its appearance. Literal numbers are recognized from a standard syntactic form of expression. Literal strings are recognized as quoted text. Literals give an immediate and direct quantification of concept instance properties.

The STL provides a concept definition sentence for defining instances of the Constant. The principal distinction between a constant and a literal is the identifier. A constant has a symbolic name by which its defining value can be referenced; a literal has no name and cannot be referenced. Constants give an indirect quantification of concept instance properties.

Values in the STL have the following subtypes:

```
<value> :: =  
    <integerValue> | <realValue> | <characterValue>  
    | <stringValue> | <stringValueShort> | <stringValueLong>  
    | <structuredValue> | <externalValue>
```

All subtypes except the last can appear in an S_Packet as either literals or constants. The last value subtype, externalValue, can appear only as a constant. See 6.3.17 for the definitions of literals.

Attributes in STL concept instances are quantified by supplying either specified keywords or instances of the value subtypes. In the concept definition templates, a mnemonic name containing an underscore marks where an instance value is to be supplied in an attribute clause. A note below the clause list(s) indicates which value subtype is allowed in place of the mnemonic.

6.1.2.2 Scoping: Organization Concepts

The STL has two concepts for organization: the Collection and the Object.

Collection concept instances are used to group software behavior concept instances into named sets. In addition, collections can be arranged into hierarchical structures. Collections correspond to the Perspective level in Figure 6. A named collection may represent a complete or partial model of the software behavior.

The Object concept expresses the aggregation and encapsulation of application properties and transforms. Object instances are defined in terms of software behavior concept instances. Objects differ from collections in that objects may be related through inheritance and specialization, while collections may not.

6.1.2.3 Mapping: Image Concepts

The STL has one concept for mapping an instance of a software behavior concept to an externally defined image instance: the GraphicSymbol.

A GraphicSymbol definition does not define a graphical rendition of a symbol. Rather, it links the image instance name attached to a concept instance to an externally supplied type definition for the image instance.

6.1.3 Software Behavior Category

STL concepts in the software behavior category express the ideas or thoughts that a product developer may want to record about the operation of a system of hardware, software, or both. Behavior category concepts correspond to the Subject level in Figure 6. Six groups of STL description concepts are distinguished on the basis of the following purposes:

- 1) *Representing application properties.* Software manipulates representations of application properties. These representations are called “data.” The specification of software behavior must therefore refer to such data representations. The STL provides six data concepts whose instances are used to describe those representations. They are the DataItem, the DataType, the DataPart, the DataKey, the DataRole, and the DataView concepts.
- 2) *Representing times.* Some software behavior requires explicit response (synchronization) to things that occur external to it. These things are occurrences that affect the operation of the software but that are not dictated by the software's own flow of control. They are called “events.” A description of software response to events needs the notion of time points and intervals. The STL provides two event concepts whose instances are used to describe the occurrences of such time points. They are the EventType and the EventItem concepts.
- 3) *Expressing decisions and assertions governing transforms.* The ability of software to deal differently with different situations is a result of its ability to compare things and make decisions. The same kind of logic with which those decisions are expressed is also used to express assertions on the results of the software behavior. The STL provides two logic concepts. They are the Condition and the Condexpression. There is no concept definition sentence for Condexpression. Instances of these concepts are used to describe decisions and assertions.
- 4) *Transforming property representations.* The primary accomplishments of software operation are calculating upon and transforming the data representations of application properties. Software operation may result in changes from inputs at one time to outputs at another time. In the important special case of data storage, operations may result in *no* change from one time to another time (an identity transform). The STL provides two transform concepts whose instances are used to describe such effects of software operation. They are the Action and the DataStore concepts.
- 5) *Propagating transform effects.* For the operation of software to have any useful effect in the external world, it must be connected to a source for input and a destination for output. Likewise, for the effects of an operation at one time to be referenced by an operation at a later time, both must be connected to some store into which the effect of the first can be deposited and retrieved. This description of software behavior in transformation and propagation terms is common in current practice. The STL therefore provides a concept whose instances are used to describe the propagation of information from source to destination. It is the ConnectionPath concept.
- 6) *Sequencing transforms.* Associated with the monotonic increase of world time is the notion of a coordinated evolution of the software and its external world through a succession of operations. The description of this evolution may involve both sequences of transforms and sequences of property representations. The STL provides a pair of evolution concepts. Their instances are used to describe individual sets of transforms and property representations, and to describe the allowed sequences in which those sets can occur. They are the State and the StateTransition concepts.

6.1.3.1 Data Concepts

The STL has six concepts for data: the DataItem, the DataType, the DataPart, the DataKey, the DataRole, and the DataView.

A DataItem concept instance describes a single occurrence of data at one place and time in the subject application. A dataitem labels the occurrence so it can be referenced in connection, evolution, logic, or transform concept definitions.